HW3 Q7,8

Muhammad Ahmed Chaudhry, Isaac Kleisle-Murphy, Kai Okada

5/31/2021

Problem 7

a) Here we code the function to implement SGD with mini-batches:

```
stochastic_gd <- function(X,y,epsilon,batch_size,epochs){
  #' Function written as per required inputs and outputs
  #' @param X: matrix, data-set of input variable values of size N*n
  #' @param y: vector of responses for each given set of input variable values, size N*1
  #' @param epsilon: numeric, value for the learning rate at each step
  #' @param B: batch size, number of observations to use at each step when computing gradient
  #' @param epochs: integer (positive), number of passes through the entire data-set
  #' @return squared_loss: matrix, of squared error loss values, one for each step, size epochs*N/B

  # Setting convenient variable names for some problem parameters
  B<-batch_size
  N<-dim(X)[1]
  n<-dim(X)[2]

  # Initializing the solution beta vector to 0
  beta<-matrix(0,nrow=n,ncol=1)

  # Initializing the loss to 0
  squared_loss<-matrix(NA,nrow=epochs,ncol=N/B)
  loss<-c(rep(0,N/B))

  # Outer loop for epochs
  for (j in 1:epochs){
    # Inner loop for batches
    for (i in 1:(N/B)){
      start<-(i-1)*B+1
      end<-start+B-1
      predictors<-as.matrix(X[start:end,])
      response<-y[start:end]
      if (B==1){
        # R treats matrix dimensions differntly if there's just one row
        # so we add an if case to be careful

        # Gradient calculation on the batch
        gradient<-2/B*(as.matrix(predictors)%*%(response-t(as.matrix(predictors))%*%beta))

        # Updating beta
        beta<-beta+epsilon*gradient
        # Calculating loss on the entire dataset for current beta
        loss_iter<-1/N*(t(y-as.matrix(X)%*%beta)%*%(y-as.matrix(X)%*%beta))

        # Storing loss value
        loss[i]<-loss_iter
      }

      else{
        # Gradient calculation on the batch
        gradient<-2/B*(t(as.matrix(predictors))%*%(response-as.matrix(predictors)%*%beta))

        # Updating beta
        beta<-beta+epsilon*gradient

        # Calculating loss on the entire dataset for current beta
        loss_iter<-1/N*(t(y-as.matrix(X)%*%beta)%*%(y-as.matrix(X)%*%beta))

        # Storing loss value
        loss[i]<-loss_iter
      }
    }
    # Getting loss for each step and for whole epoch
    squared_loss[j,]<-loss

  }

#x<-list(squared_loss, beta)

return(squared_loss)
}
```

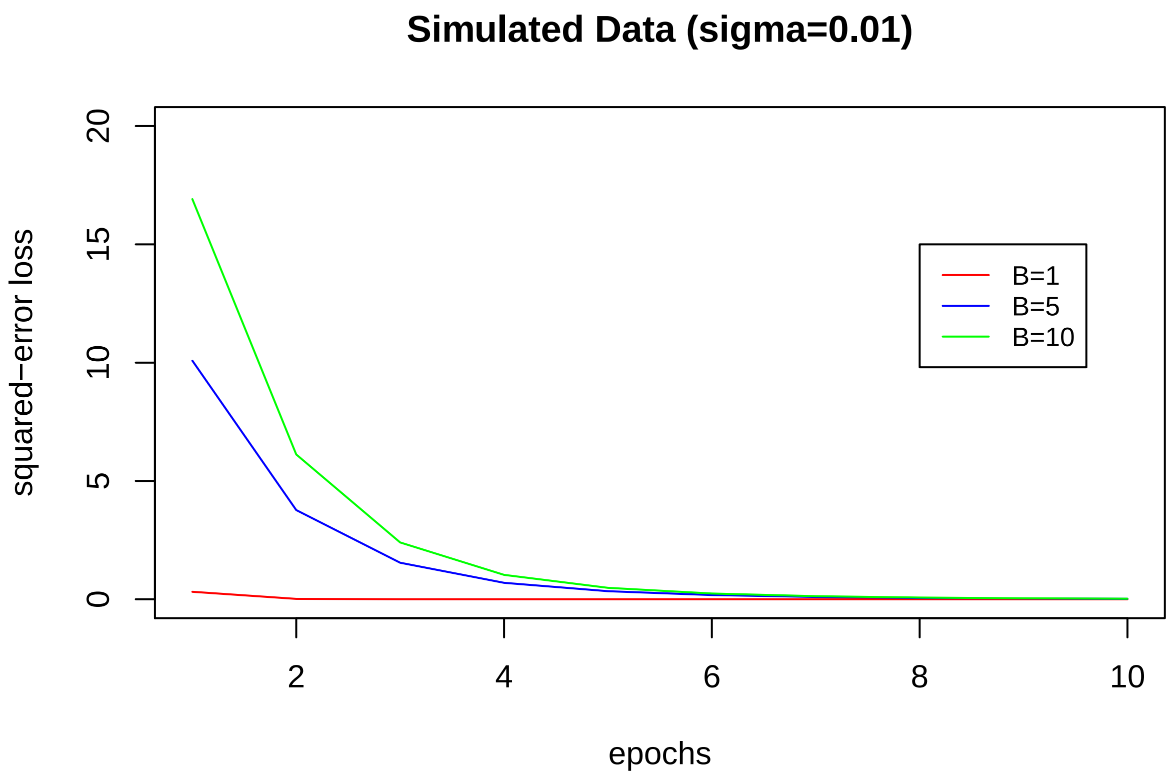
b) Now we test the function with the data-set that we generate ($\sigma = 0.01$ case):

```
# Generating our predictors, coefficients and response
set.seed(2021)
sigma<-0.01
X<-mvrnorm(n=100, mu=rep(0,10),Sigma=diag(rep(1,10)))
a_star<-mvrnorm(n=1, mu=rep(0,10),Sigma=diag(rep(1,10)))
delta<-mvrnorm(n=1, mu=rep(0,100),Sigma=diag(rep(sigma^2,100)))

y<-X%*%a_star+delta


# Running the function, once for each batch size
result_1<-stochastic_gd(X,y,0.01,1,10) # B=1
result_5<-stochastic_gd(X,y,0.01,5,10) # B=5
result_10<-stochastic_gd(X,y,0.01,10,10) # B=10
```

Then we plot the loss value obtained after each epoch for each of the batch sizes:

```
epochs<-1:10
loss_epochs_1<-result_1[,dim(result_1)[2]]
loss_epochs_5<-result_5[,dim(result_5)[2]]
loss_epochs_10<-result_5[,dim(result_10)[2]]
```

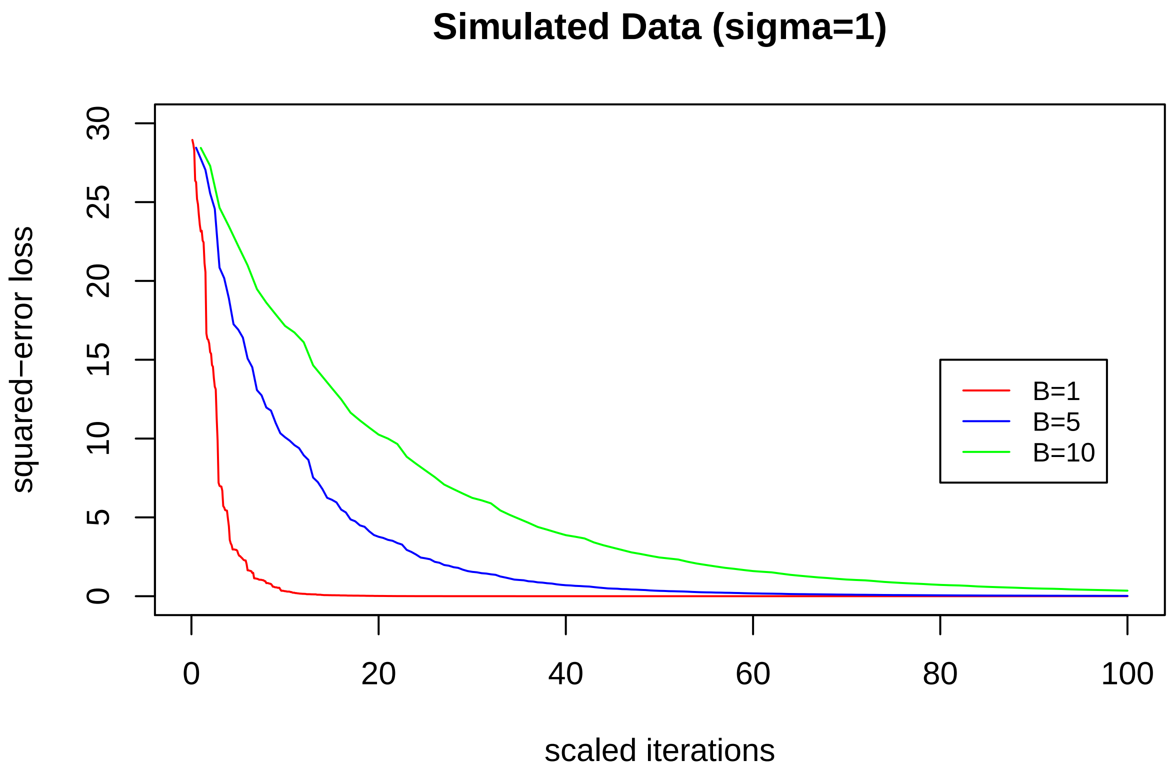
```
plot(x=epochs,y=loss_epochs_1,type="l",col="red",ylim=(c(0,20)),
     ylab="squared-error loss",main="Simulated Data (sigma=0.01)")
points(x=epochs,y=loss_epochs_5,type="l",col="blue")
points(x=epochs,y=loss_epochs_10,type="l",col="green")
legend(8,15, legend=c("B=1", "B=5", "B=10"),
       col=c("red", "blue", "green"), lty=1, cex=0.8)
```



As per the plot above, we see that the method with Batch size=1 performs the best in terms of fastest convergence, followed by a batch size of 5, and then a batch size of 10. This is primarily because of the massive difference in terms of the number of steps taken per epoch between the different batch sizes. For a batch size of 1, the algorithm takes a 100 update steps in 1 epoch, whereas a batch size of 5 takes 20 steps, and a batch size of 10 takes 10 steps. Even though with a batch size of 1, the gradient can be very noisy since it is calculated using only one observation, and due to that noise the algorithm has a high chance of moving away from the global minimum, since the learning rate or step size (ε) is very small (0.01), such a situation does not arise, and the affect of greater number of steps than with a batch size of 5 or 10 dominates. The plot below where we present the loss at each step between the 3 approaches presents a fuller picture:

```
iters_1<-(1:1000)/10
loss_epochs1_full<-as.vector(t(result_1))
# plot(x=iters_1, y=loss_epochs1_full, type="l")

iters_2<-(1:200)/2
loss_epochs5_full<-as.vector(t(result_5))
# plot(x=iters_2, y=loss_epochs5_full, type="l")

iters_3<-(1:100)/1
loss_epochs10_full<-as.vector(t(result_10))
```

```
# plot(x=iters_3, y=loss_epochs10_full, type="l")


plot(x=iters_1,y=loss_epochs1_full,type="l",col="red",ylim=c(0,30),
     ylab="squared-error loss", xlab="scaled iterations",
     main="Simulated Data (sigma=1)")
points(x=iters_2,y=loss_epochs5_full,type="l",col="blue")
points(x=iters_3,y=loss_epochs10_full,type="l",col="green")
legend(80,15, legend=c("B=1", "B=5", "B=10"),
       col=c("red", "blue", "green"), lty=1, cex=0.8)
```

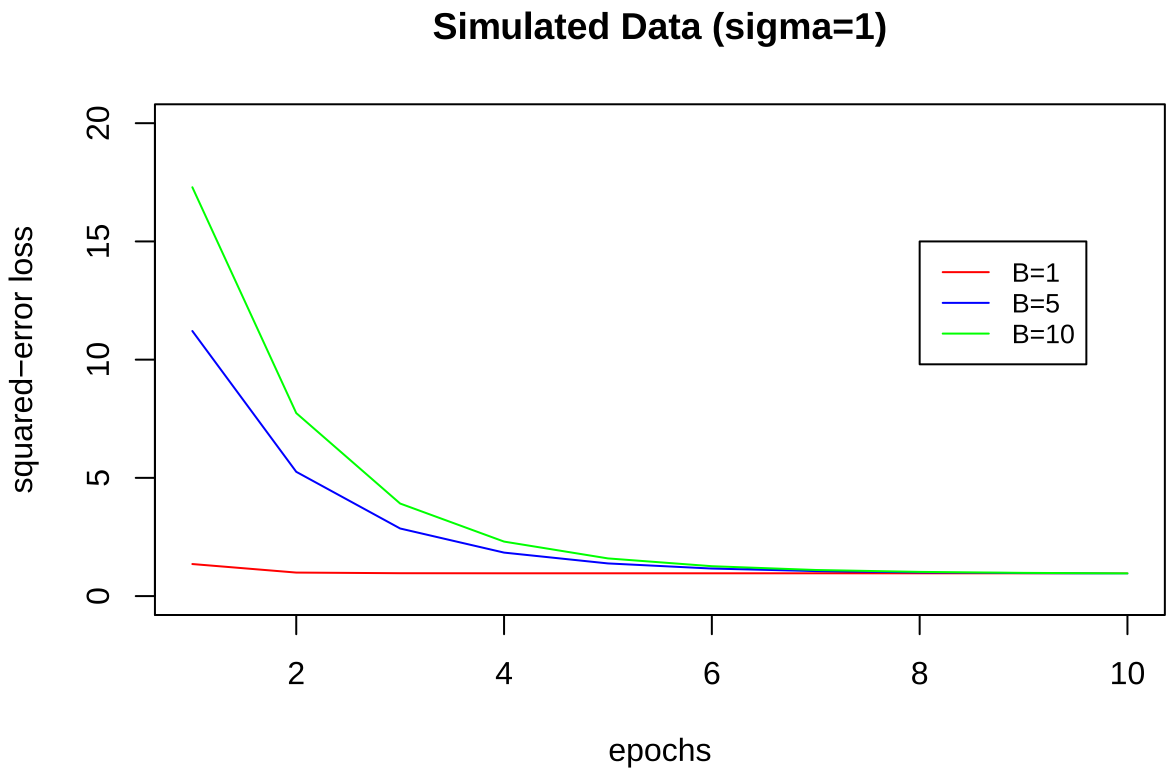


We can very clearly see the effect of the noisier gradient with a batch size of 1, and then the gradient becomes less noise going from a batch size of 1, to 5, to 10, to give us the smoothest descent with a batch size of 10. However, the number of steps affect dominates (due to a very small learning rate, ε) to give the fastest convergence for batch size 1 case, followed by batch size 5 and then 10 in that order.

c) Now we change the value of σ to 1 to make the a lot more noisy:

```
# Generating our predictors, coefficients and response
set.seed(2022)
sigma<-1
X<-mvrnorm(n=100, mu=rep(0,10),Sigma=diag(rep(1,10)))
a_star<-mvrnorm(n=1, mu=rep(0,10),Sigma=diag(rep(1,10)))
delta<-mvrnorm(n=1, mu=rep(0,100),Sigma=diag(rep(sigma^2,100)))

y<-X%*%a_star+delta

# Running the function, once for each batch size
result_1<-stochastic_gd(X,y,0.01,1,10) # B=1
result_5<-stochastic_gd(X,y,0.01,5,10) # B=5
```

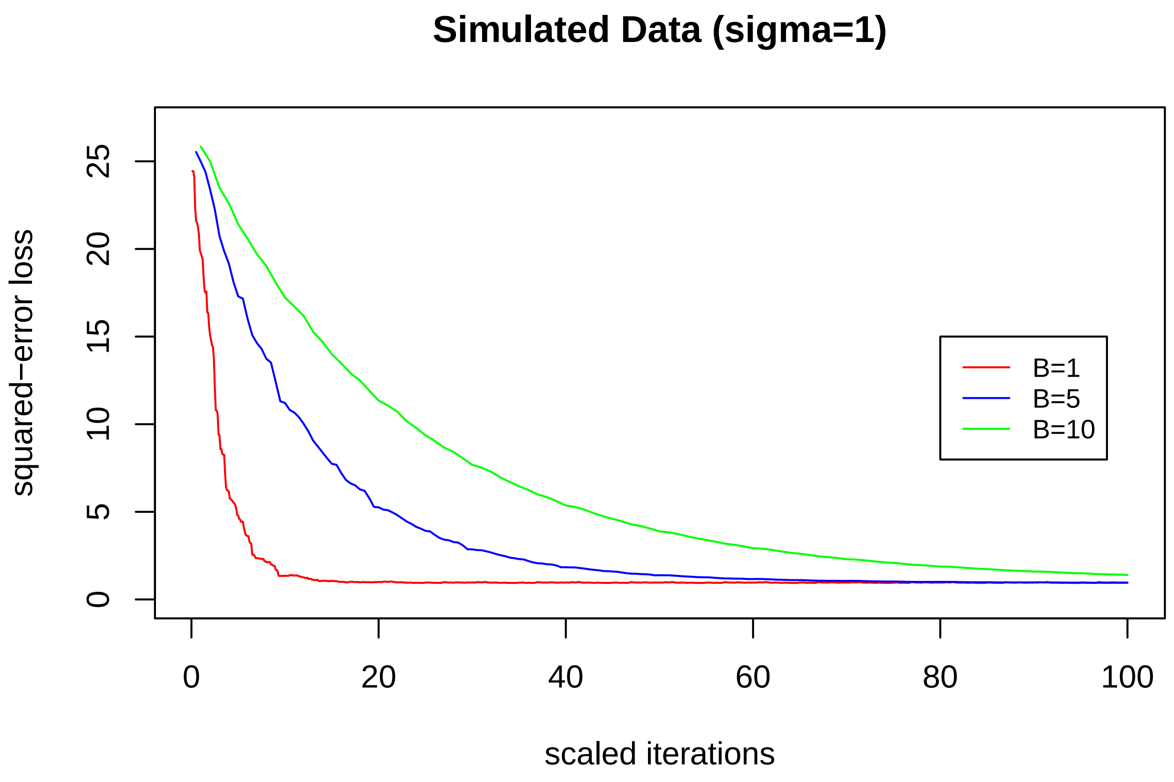
```
result_10<-stochastic_gd(X,y,0.01,10,10) # B=10


epochs<-1:10
loss_epochs_1<-result_1[,dim(result_1)[2]]
loss_epochs_5<-result_5[,dim(result_5)[2]]
loss_epochs_10<-result_5[,dim(result_10)[2]]

# Plotting the results
plot(x=epochs,y=loss_epochs_1,type="l",col="red",ylim=c(0,20),
     ylab="squared-error loss", main="Simulated Data (sigma=1)")
points(x=epochs,y=loss_epochs_5,type="l",col="blue")
points(x=epochs,y=loss_epochs_10,type="l",col="green")
legend(8,15, legend=c("B=1", "B=5", "B=10"),
       col=c("red", "blue", "green"), lty=1, cex=0.8)
```



Again, we see a very similar plot to part b). The number of steps affect dominates to give the fastest rate of convergence for a batch size of 1, followed by batch size of 5 and then batch size of 10 (even though here the data is a lot noisier, the small enough learning rate ε more than compnensates for it, at least uptil 10 epochs). We plot the complete trajectory below:

```
iters_1<-(1:1000)/10
loss_epochs1_full<-as.vector(t(result_1))
#plot(x=iters_1, y=loss_epochs1_full, type="l")

iters_2<-(1:200)/2
loss_epochs5_full<-as.vector(t(result_5))
#plot(x=iters_2, y=loss_epochs5_full, type="l")

iters_3<-(1:100)/1
```

```r
loss_epochs10_full<-as.vector(t(result_10))
#plot(x=iters_3, y=loss_epochs10_full, type="l")


plot(x=iters_1,y=loss_epochs1_full,type="l",col="red",ylim=c(0,27),
     ylab="squared-error loss", xlab="scaled iterations",
     main="Simulated Data (sigma=1)")
points(x=iters_2,y=loss_epochs5_full,type="l",col="blue")
points(x=iters_3,y=loss_epochs10_full,type="l",col="green")
legend(80,15, legend=c("B=1", "B=5", "B=10"),
       col=c("red", "blue", "green"), lty=1, cex=0.8)
```



The plot above shows that the difference between the smoothness of the paths for batch sizes 1 to 5 to 10 is a lot more apparent, with the batch size equals 1 case being the noisiest, and batch size equals 10 being the smoothest. Based on this plot, we conjecture that if the algorithm was run for a greater number of epochs, the batch size 10 approach, due to the fact that it is more stable, may actually converge to a slightly lower eventual loss value than the other batch size algorithms (so estimating the coefficients vector better).

Problem 8

a) A larger neighborhood means that more observations will be averaged to get an estimate of the response variable for a particular set of values for the inputs. This will give us lower variance for the prediction estimate since a small change in the input values (slight increase in the dissimilarity) will not change the estimate of the response very much as the observations in the neighborhoods will more or less be the same between the two sets of input values. This is an advantage of using a larger neighborhood. However, there will potentially be higher bias because points slightly further off (more dissimilar) will affect the estimate of the response variable since the neighborhood size will be large enough to capture those. This is a disadvantage of using a larger neighborhood size. On the other hand, using a smaller neighborhood induces higher variance for the estimate of the response variable, which is a disadvantage. A small change in the input values slight increase in dissimilarity) could significantly change the estimate of the response because the neighborhood size is relatively small, and so the observations included in the average could be very different between two slightly different sets of input values. However, the estimate will have lower bias since only points very close to a particular test point (very similar) in input variable space will be avergaed to make an estimate for the response for those inputs. This is an advantage of using a smaller neighborhood.

Which of the two approaches performs better will depend on the extent to which the function we're estimating meets one of the fundamental assumptions of near neighbor regression that points closer together (having low dissimilarity) in input space will have similar response values. This will determine which of the bias or variance affects will dominate.

b) Since the near neighbor regression method (at least how it is implemented in R) calculates dissimilarity based on Euclidean distance, an advantage of standardizing all the variables to have the same variance before training is that the model will not be adversely affected simply because the input variables have different scales or units (different variances), which will pull the prediction estimates in different directions due to scale differences. However, a potential disadvantage of standardizing all variables to have the same variance is that if the data-set includes input variables which are purely noise and carry no signal but are very low variance, standardizing (to make all variables have the same variance) could scale these variables up to have a much higher variance than originally. This up-scaling of the noise variables will in turn harm model performance by clouding the signal from other input variables. We demonstrate this precisely in the simulation we run for part c).

c) Here we create a small simulation to show that in near neighbor regression, standardizing variables is not always a good idea:

```r
# Generating bi-variate normal (0,1) data
set.seed(2021)
X<-mvrnorm(n=100, mu=rep(0,2),Sigma=diag(rep(1,2)))

# arbitrary choice for true coefficient vector
beta<-c(1,2)

# irreducible noise which is normal(0,1)
epsilon<-rnorm(n=100, mean=0, sd=1)

# True function is y=x_1+2*x_2+epsilon
y<-X%*%beta+epsilon


# Here we generate the noise variable which we will include in our dataset
noise<-rnorm(n=100, mean=0, sd=0.01)

# Standardizing all variables to have the same variance (1) in a separate dataset
noise_standardized<-noise/sd(noise)

X_train<-as.data.frame(cbind(X,noise))
X_train_standardized<-as.data.frame(cbind(X,noise_standardized))
X_train_standardized$V1<-X_train_standardized$V1/sd(X_train_standardized$V1)
X_train_standardized$V2<-X_train_standardized$V2/sd(X_train_standardized$V2)


# Now we randomly split the data into train and test set and then run knn with
# noise standardized versus not standardized
# split the dataset
set.seed(2021)
# 80% training
train_rows <- sample(1:nrow(X_train), size = as.integer(nrow(X_train) * 0.8))
test_rows <- setdiff(1:nrow(X_train),train_rows) # 20% test

train_data <- X_train[train_rows,]
test_data <- X_train[test_rows,]

train_data_standardized<-X_train_standardized[train_rows,]
test_data_standardized<-X_train_standardized[test_rows,]

# Splitting the response into train and test sets based on the same indices
y_train<-y[train_rows,]
y_test<-y[test_rows,]

# Training the standardized and non-standardized data models
# We don't perform tuning on the parameter k for simplicity and just run
# the model for k=3
model<-knn.reg(train_data,test = test_data,y=y_train,k=3)
model_standardized<-knn.reg(train_data_standardized,
                            test = test_data_standardized,y=y_train,k=3)

# Calculating the MSE in each case
model_mse<-mean((y_test-model$pred)^2)
model_standardized_mse<-mean((y_test-model_standardized$pred)^2)

print(paste0("The model MSE with non-standardized variables is ", model_mse))
```

```
## [1] "The model MSE with non-standardized variables is 1.56052081184031"
```

```r
print(paste0("The model MSE with standardized variables is ", model_standardized_mse))
```

```
## [1] "The model MSE with standardized variables is 3.09219486094146"
```

```r
print(paste0("The standardized variables model MSE is ",
             round((model_standardized_mse/model_mse-1)*100,3),
             "% higher"))
```

```
## [1] "The standardized variables model MSE is 98.151% higher"
```

```r
#model_standardized_mse/model_mse-1
```

And so we see that with the standardized data, since the noise variable was up-scaled quite a bit, the MSE obtained from the model trained on the standardized data is 98.151% higher than the MSE obtained from the model trained on non-standardized data. This is manifestation of the idea that standardizing all variables to have the same variance is not always a good idea. While practically speaking, we will never actually know if a variable in the data-set is purely low variance noise, this example illustrates that we run the risk of up-scaling noise variables through standardization, and the approach used works to manifest this idea as discussed and confirmed in office hours.