# STAT 542: Homework 4

Sharvi Tomar (stomar2)

## Contents

Question 1 [35 Points] Regression and Optimization with Huber Loss . . . . . . . . . . . . . . . . . . 1
Question 2 [65 Points] Scaling and Coordinate Descent for Linear Regression . . . . . . . . . . . . 4

```
seed_value = 24
```

### Question 1 [35 Points] Regression and Optimization with Huber Loss

When fitting linear regressions, outliers could significantly affect the fitting results. However, manually checking and removing outliers can be tricky and time consuming. Some regression methods address this problem by using a more robust loss function. For example, one such regression is to minimize the objective function

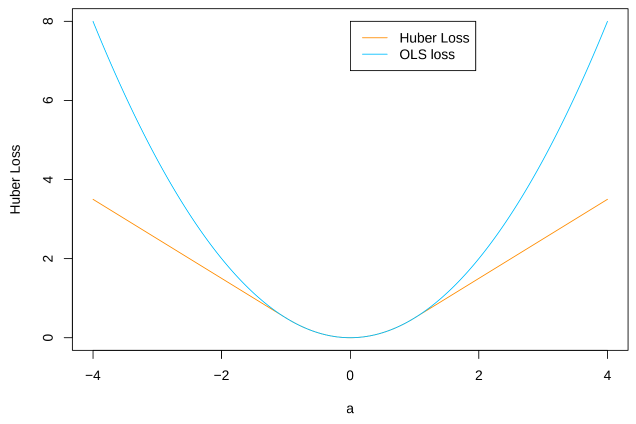
$$\frac{1}{n}\sum_{i=1}^{n} \ell_\delta(y_i - x_i^{\mathrm{T}}\beta),$$

where the loss function $\ell_\delta$ is the **Huber Loss**, defined as

$$\ell_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{if } |a| \le \delta \\ \delta(|a| - \frac{1}{2}\delta) & \text{o.w.} \end{cases}$$

Here is a visualization that compares Huber loss with the $\ell_2$ loss. We can see that the Huber loss assigns much less value when $y_i - x_i^{\mathrm{T}}\beta$ is more extreme (outliers).

```
  # define the Huber loss function
  Huber <- function(a, delta = 1) ifelse(abs(a) <= delta, 0.5*a^2, delta*( abs(a) - 0.5*delta))

  # plot against L2
  x = seq(-4, 4, 0.01)
  plot(x, Huber(x), type = "l",
       xlab = "a", ylab = "Huber Loss",
       col = "darkorange", ylim = c(0, 8))
  lines(x, 0.5*x^2, col = "deepskyblue")
  legend(x = 0, y = 8, legend = c("Huber Loss", "OLS loss"),
         col = c("darkorange", "deepskyblue"), lty = 1)
```



Use the following code to generate

```r
# generate data from a simple linear model
set.seed(542)
n = 150
x = runif(n)
X = cbind(1, x)
y = X %*% c(0.5, 1) + rnorm(n)

# create an outlier
y[which.min(X[, 2])] = -30
```

a) [5 pts] Fit an OLS model with the regular $\ell_2$ loss. Report your coefficients (do not report other information). Although this is only one set of samples, but do you expect this estimator to be biased based on how we set up the observed data? Do you expect the parameter $\beta_1$ to bias upwards or downwards? Explain your reason. Hint: is the outlier pulling the regression line slope up or down?

```r
model = lm(y ~ X - 1) #Create the linear regression
model$coefficients
```

```
##          X          Xx
## -0.1253725  1.9036546
```

b) [10 pts] Define your own Huber loss function `huberLoss(b, trainX, trainY)` given a set of observed data with tuning parameter $\delta = 1$. Here, `b` is a $p$-dim parameter vector, `trainX` is a $n \times p$ design matrix and *trainY* is the outcome. This function should return a scalar as the empirical loss. You can use our `Huber` function in your own code. After defining this loss function, use the `optim()` function to solve the parameter estimates. Finally, report your coefficients.

- Use `b = (0, 0)` as the initial value.
- Use `BFGS` as the optimization method.

```r
# function definition
huberLoss = function(b, trainx, trainy) mean(Huber(trainy - trainx %*% b,
                                                   delta = 1))
```

```r
  # best set of parameters using optim function
  b = c(0, 0)
  huber_loss_optim = optim(par = b, fn = huberLoss, method = "BFGS",
                           trainx = X, trainy = y)
  huber_loss_optim$par
```

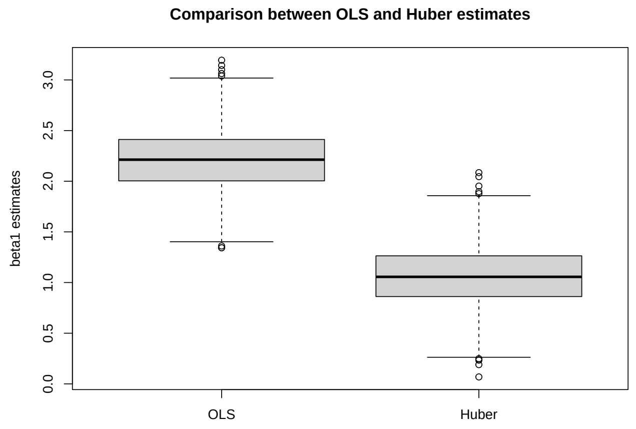
```
## [1] 0.7545940 0.6223551
```

c) [20 pts] We still do not know which method performs better in this case. Let's use a simulation study to compare the two methods. Complete the following

   - Set up a simulation for 1000 times. At each time, randomly generate a set of observed data, but also force the outlier with our code `y[which.min(X[, 2])] = -30`.
   - Fit the regression model with $\ell_2$ loss and Huber loss, and record the slope variable estimates.
   - Make a side-by-side boxplot to show how these two methods differ in terms of the estimations. Which method seem to have more bias? and report the amount of bias based on your simulation. What can you conclude from the results? Does this match your expectation in part a)? Can you explain this (both OLS and Huber) with the form of loss function, in terms of what their effects are?

```r
set.seed(24)
nsim = 1000

ols_slopes = rep(0, nsim)
huber_slopes = rep(0, nsim)

for (l in 1:nsim){
  n = 150
  x = runif(n)
  X = cbind(1, x)
  y = X %*% c(0.5, 1) + rnorm(n)
  # create an outlier
  y[which.min(X[, 2])] = -30

  lm.fit = lm(y ~ X - 1)
  ols_slopes[l] = lm.fit$coefficients[2]

  b = c(0,0)
  hL.optim = optim(par = b, fn = huberLoss, method = "BFGS",
                           trainx = X, trainy = y)
  huber_slopes[l] = hL.optim$par[2]
}
```

```r
# side-by-side box plot
boxplot(ols_slopes, huber_slopes, main = "Comparison between OLS and Huber estimates",
        ylab = "beta1 estimates", names = c("OLS", "Huber"))
```



```r
# bias calculation
sprintf("Bias for OLS: %f", mean(ols_slopes) - 1)
```

```
## [1] "Bias for OLS: 1.209231"
```

```r
sprintf("Bias for Huber: %f", mean(huber_slopes) - 1)
```

```
## [1] "Bias for Huber: 0.055600"
```

The bias is shown as above. Linear model with square error is biased upwards. This is the same as our previous expectation. However, the model with Huber's loss has little bias. Huber's loss becomes linear instead of quadratic for large deviated samples. Thus, the contributed from this one outlier does not have significant impact on the estimation.

## Question 2 [65 Points] Scaling and Coordinate Descent for Linear Regression

**Scaling issue** In the practice, we usually standardize each covariate/feature to mean 0 and standard deviation 1. Standardization is essential when we apply $\ell_2$ and $\ell_1$ penalty on the loss function, because if the covariates are with different scales, then they are penalized differently. Without prior information, we should prevent that from happening. Besides, scaling the data also help to make the optimization more stable, since the step size in many descent algorithms could be affected by the scale.

In practice, after obtaining the coefficients fitted with scaled data, we want to recover the original coefficients of the unscaled data. For this question, we use the following intuition:

$$\frac{Y - \bar{Y}}{\text{sd}_y} = \sum_{j=1}^{p} \frac{X_j - \bar{X}_j}{\text{sd}_j} \gamma_j \tag{1}$$

$$Y = \underbrace{\bar{Y} - \sum_{j=1}^{p} \bar{X}_j \frac{\text{sd}_y \cdot \gamma_j}{\text{sd}_j}}_{\beta_0} + \sum_{j=1}^{p} X_j \underbrace{\frac{\text{sd}_y \cdot \gamma_j}{\text{sd}_j}}_{\beta_j}, \tag{2}$$

- In this equation, the first line is the model fitted with scaled and centered data. And we obtain the fitted parameters as $\gamma_j$'s
- In the second line, the coefficients $\beta_j$'s for the original data is recovered.
- When fitting the scaled and centered data, no intercept term is needed.

Based on this relationship, we perform the following when fitting a linear regression:

- Center and scale both $\mathbf{X}$ (column-wise) and $\mathbf{y}$ and denote the processed data as $\frac{Y-\bar{Y}}{\text{sd}_y}$ and $\frac{X_j-\bar{X}_j}{\text{sd}_j}$ in the above formula. Make sure that the standard error of each variable is 1 after scaling. This means that you should use $N$, not $N-1$ when calculating the estimation of variance.
- Fit a linear regression using the processed data based on the no-intercept model, and obtain the parameter estimates $\gamma_j$'s.
- Recover the original parameters $\beta_0$ and $\beta_j$'s.

Use the following code to generate your data:

```
library(MASS)
set.seed(10)
n = 20
p = 3

# covariance matrix
V = matrix(0.3, p, p)
diag(V) = 1

# generate data
X_org = as.matrix(mvrnorm(n, mu = rep(0, p), Sigma = V))
true_b = c(1, 2, 0)
y_org = X_org %*% true_b + rnorm(n)
```

a) [10 pts] Fit an OLS estimator with the original data `Y_org` and `X_org` by `lm()`. Also, fit another OLS with scaled data by `lm()`. Report the coefficients/parameters. Then, transform coefficients from the second approach back to its original scale, and match with the first approach. Summarize your results in a single table: The rows should contain three methods: OLS, OLS Scaled, and OLS Recovered, and there should be four columns that represents the coefficients for each method. You can consider using the `kable` function, but it is not required.

```
set.seed(24)
# fitting the original model
org.fit = lm(y_org ~ X_org, data = data.frame(X_org))
ols_coeff = org.fit$coefficients

normalizing_factor = sqrt(n/(n - 1))
# Scaling and centering the data
X_scaled = scale(X_org)*normalizing_factor
y_scaled = scale(y_org)*normalizing_factor

# fitting the model with scaled and centered data
scale.fit = lm(y_scaled ~ X_scaled - 1, data = data.frame(X_scaled))
scale_coeff = scale.fit$coefficients
```

Now I will recover the coefficients using the transformed coefficients

```
# standard deviation for scaling and mean for centering
# transformation of X_org
sd_X = apply(X_org, 2, sd)
mean_X = apply(X_org, 2, mean)
```

```
  # transformation of y_org
  sd_y = sd(y_org)
  mean_y = mean(y_org)

  # recovering the beta coefficients
  b1_recover = scale_coeff*(sd_y/sd_X)
  b0_recover = mean_y - sum(mean_X *scale_coeff*(sd_y/sd_X))
```

Summarizing the results in a table below-

```
data = rbind(ols_coeff, c(0, scale_coeff), c(b0_recover,b1_recover))
rownames(data) = c("OLS", "OLS Scaled",  "OLS Recovered")
knitr::kable(data,  caption = "Coefficients", digits = 3, escape = FALSE)
```

Table 1: Coefficients

| | (Intercept) | X_org1 | X_org2 | X_org3 |
|---|---|---|---|---|
| OLS | -0.517 | 0.659 | 2.183 | 0.250 |
| OLS Scaled | 0.000 | 0.231 | 0.811 | 0.068 |
| OLS Recovered | -0.517 | 0.659 | 2.183 | 0.250 |

b) Instead of using the `lm()` function, write your own coordinate descent code to solve the scaled problem. This function will be modified and used next week when we code the Lasso method. Complete the following steps:

- [10 pts] i) Given the loss function $L(\beta) = \|y - X\beta\|^2$ or $\sum_{i=1}^n (y_i - \sum_{j=0}^p x_{ij}\beta_j)^2$, derive the updating/calculation formula of coefficient $\beta_j$, when holding all other covariates fixed. You must use LaTex to typeset your derivation with proper explaination of notations. Write down the formula (in terms of $y$, $x$ and $\beta$'s) of residual $r$ before and after this update. Based on our lecture, how to make the update of $r$ computationally efficient?

Answer- To minimize the $\beta_j$ holding all the other covariates fixed, we can take the following equation -

$$L(\beta) = \|y - X\beta\|^2 = \sum_{i=1}^n \|y_i - x_i\beta\|^2$$

To start with taking partial derivative of $L(\beta)$ w.r.t. $j^{th}$ entry of $\beta$.

Taking the partial derivative,

$$\frac{\partial L}{\partial \beta_j} = \sum_{i=1}^n x_j(y_i - x_{i(-j)}\beta_{(-j)} - x_j\beta_j) = 0$$

We know residuals is given as,

$$r = y - x_{(-j)}\beta_{(-j)}$$

Plugging $r$ into the equation $\frac{\partial L}{\partial \beta_j}$ we get,

$$\frac{\partial L}{\partial \beta_j} = \sum_{i=1}^n x_j^T(y_i - x_{i(-j)}^T\beta_{(-j)} - x_j\beta_j) = \sum_{i=1}^n x_j^T(r - x_j\beta_j) = 0$$

Solving this we get $\beta_j$,

$$\hat{\beta_j}^{OLS} = \frac{x_j^T r}{x_j^T x_j}$$

**Update rule**

Calculating the residual $r = y - X_{(-j)}\beta_{(-j)}$ can be very costly since it involves multiplying using a n $\times$ (p − 1) matrix. Instead, since we only update one $\beta_j$ at a time, the residual r at the next iteration can be obtained with -

$$r = r - x_j^T \beta_j^{(k+1)} + x_{(j+1)}^T \beta_{(j+1)}^{(k)}$$

where,the first term $r - x_j^T \beta_j^{(k+1)}$ is residual of current model and the term of $j+1$ added back to residual is $x_{(j+1)}^T \beta_{(j+1)}^{(k)}$.

```
- [30 pts] ii) Implement this coordinate descent method with your own code to solve OLS with the scaled
  - Do not use functions from any additional library.
  - Start with a vector $\boldsymbol \beta = 0$.
  - Run your coordinate descent algorithm for a maximum of maxitr = 100 iterations (while each iteratio
```

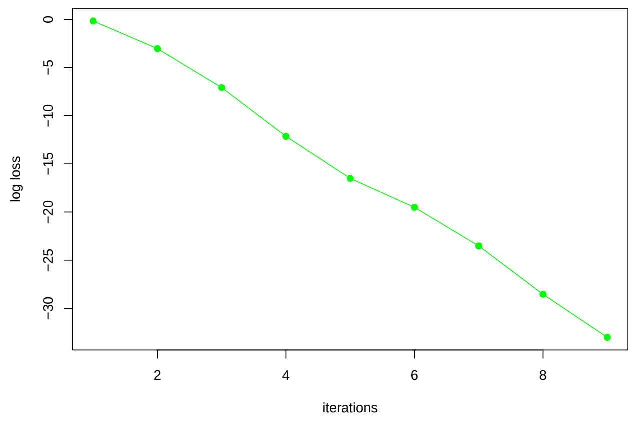
```
# coordinate descent algorithm
set.seed(24)

tol = 1e-7
maxitr = 100
loss_vec =rep(0, ncol(X_scaled))
oldB = rep(0, ncol(X_scaled))
tempB = oldB

# max iteration loop
for (k in 1:maxitr){
  oldB = tempB
  res = y_scaled - X_scaled %*% oldB
  loss_vec[k] = mean(res^2)

  # updating each beta_j value per k then update residual and beta
  for (j in 1:ncol(X_scaled)){
    res = res + X_scaled[, j] * tempB[j]
    tempB[j] = (t(X_scaled[,j]) %*% res)/(t(X_scaled[,j]) %*% X_scaled[,j])
    res = res - X_scaled[, j] * tempB[j]
  }
  # stopping criteria
  if (sum(abs(tempB - oldB)) < tol) break;
}

rbind("Coefficients using coordinate descent function: "= round(tempB, 3))
```

```
##                                                    [,1]  [,2]  [,3]
## Coefficients using coordinate descent function:  0.231 0.811 0.068
```

**Comments** The coefficients match exactly with the results for scaled OLS in part (a).

```
-  [5 pts] Make a plot to analyze the convergence of the coordinate descent. On the x-axis, we use the n
```

```
trueLoss = mean(scale.fit$residuals^2)
plot(1:length(loss_vec),  log(loss_vec - trueLoss), type ="o", pch=19,
     col = "green", xlab = "iterations", ylab = "log loss")
```


log loss
iterations