STAT 542: Homework 4

Spring 2022, by Kamila Makhambetova (kamilam3)

Due: Thur, Feb 17, 11:59 PM CT

Instruction

Students are encouraged to work together on homework. However, sharing, copying, or providing any part of a homework solution or code is an infraction of the University's rules on Academic Integrity. Any violation will be punished as severely as possible. Final submissions must be uploaded to compass2g. No email or hardcopy will be accepted. For **late submission policy and grading rubrics** (https://teazrq.github.io/stat542/homework.html), please refer to the course website.

- What is expected for the submission to **Gradescope**
    - You are required to submit one rendered **PDF** file `HWx_yourNetID.pdf`. For example, `HW01_rqzhu.pdf`. Please note that this must be a `.pdf` file generated by a `.Rmd` file. `.html` format cannot be accepted.
    - Please follow the instructions on Gradescope to select corresponding PDF pages for each question.
- Please note that your homework file is a **PDF** report instead of a messy collection of R codes. This report should **include**:
    - Your Name and NetID. (Replace `Ruoqing Zhu (rqzhu)` by your name and NetID if you are using this template).
    - Make all of your `R` code chunks visible for grading.
    - Relevant outputs from your `R` code chunks that support your answers.
    - Provide clear answers or conclusions for each question. For example, you could start with `Answer: I fit SVM with the following choice of tuning parameters ...`
    - Many assignments require your own implementation of algorithms. **Basic comments are strongly encouraged** to explain the logic to our graders. However, line-by-line code comments are unnecessary.
- Requirements regarding the `.Rmd` file.
    - You do **NOT** need to submit `Rmd` files. However, your PDF file should be rendered directly from it.
    - Make sure that you **set random seeds** for simulation or randomized algorithms so that the results are reproducible. If a specific seed number is not provided in the homework, you can consider using your NetID.
    - For some questions, there will be restrictions on what packages/functions you can use. Please read the requirements carefully. As long as the question does not specify such restrictions, you can use anything.

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

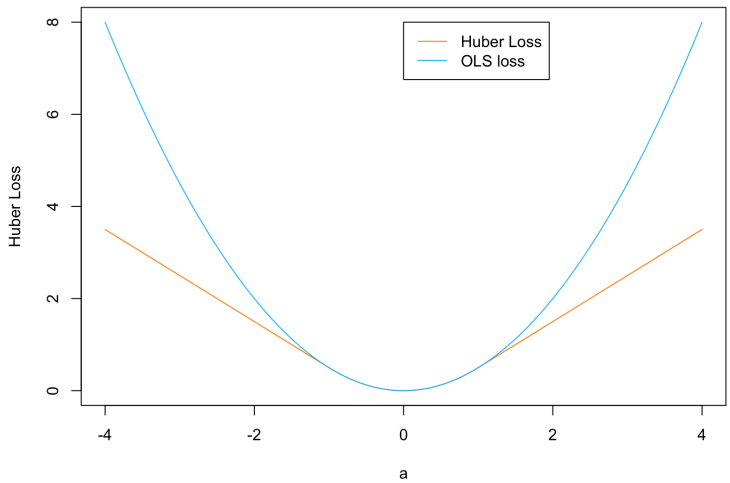
$$\frac{1}{n}\sum_{i=1}^{n} \ell_\delta(y_i - x_i^{\mathrm{T}}\boldsymbol{\beta}),$$

where the loss function ℓ_δ 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 ℓ_2 loss. We can see that the Huber loss assigns much less value when $y_i - x_i^{\mathrm{T}}\boldsymbol{\beta}$ 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

```
# 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 ℓ_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 β_1 to bias upwards or downwards? Explain your reason. Hint: is the outlier pulling the regression line slope up or down?

```
library(matlib)

#OLS model with l2 loss

ols2<-lm(y~x)

print(paste0("The B_0 = ",ols2$coefficients[1]))
```

```
## [1] "The B_0 = -0.125372514246031"
```

```
print(paste0("The B_1 = ",ols2$coefficients[2]))
```

```
## [1] "The B_1 = 1.90365455007681"
```

Comment

The coefficients are $B_0 = -0.1254$ and $B_1 = 1.9037$. I think this estimator is biased based on how we set up the observed data. β_1 is biased downwards as the outlier is negative $Y_{outlier} = -30$ and it pulls regression line slope down.

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.

```
Huber <- function(a, delta ) ifelse(abs(a) <= delta, 0.5*a^2, delta*( abs(a) - 0.5*delta))

#Huber Loss function
huberLoss<-function(b, trainX, trainY){
  n=nrow(trainY)
  sigma=1
  error= rep(0,n)

  for(i in c(1:n)){
    a=trainY[i]-trainX[i, ]%*%b
    error[i]=Huber(a, sigma)
  }
  sum(error)/n
}

#optimizing Huber loss
result<-optim(par=c(0,0), fn=huberLoss, method="BFGS", trainX=X, trainY=y)

print(paste0("The B_0 = ",result$par[1]))
```

```
## [1] "The B_0 = 0.754594039903637"
```

```
print(paste0("The B_1 = ",result$par[2]))
```

```
## [1] "The B_1 = 0.622355061979864"
```

```
print(paste0("Optimal HuberLoss(B_0, B_1) = ",huberLoss(result$par, X, y)))
```

```
## [1] "Optimal HuberLoss(B_0, B_1) = 0.628448500753617"
```

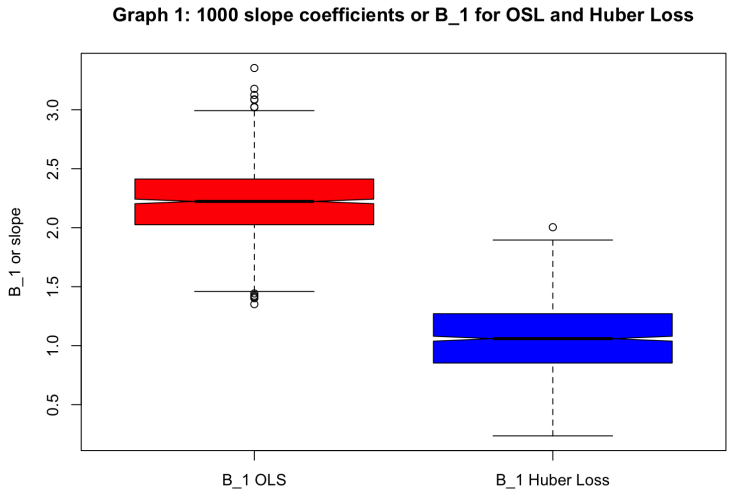
Comment

The coefficients are $B_0 = 0.7545940$ and $B_1 = 0.6223551$. The corresponding optimal Huber Loss = 0.6284485.

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 ℓ_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?

```
set.seed(542)

beta_ols_0=rep(0,1000)
beta_ols_1=rep(0,1000)

beta_Huber_0=rep(0,1000)
beta_Huber_1=rep(0,1000)

for(i in (1:1000)){

  # generate data from a simple linear model
  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

   #find osl beta
   ols3<-lm(y~x)
   beta_ols_0[i]=ols3$coefficients[1]
   beta_ols_1[i]=ols3$coefficients[2]

   #find Huber loss beta
   result<-optim(par=c(0,0), fn=huberLoss, method="BFGS", trainX=X, trainY=y)
   beta_Huber_0[i]=result$par[1]
   beta_Huber_1[i]=result$par[2]
}

#Draw graphs
data_slope<-data.frame(beta_ols_1,beta_Huber_1)
boxplot(data_slope, main = 'Graph 1: 1000 slope coefficients or B_1 for OSL and Huber Loss', yl
ab = "B_1 or slope", names=c("B_1 OLS","B_1 Huber Loss"), notch=TRUE, col=(c("red", "blue")))
```




```
#Bias= E(pred Beta) - true_Beta
Bias_ols=mean(beta_ols_1)-1
Bias_Huber=mean(beta_Huber_1)-1

print(paste0("For OSL bias of B_1 = ", Bias_ols))
```

```
## [1] "For OSL bias of B_1 = 1.22147105511238"
```

```
print(paste0("For Huber Loss bias of B_1 = ",Bias_Huber))
```

```
## [1] "For Huber Loss bias of B_1 = 0.0592877704629744"
```

Comment

According to boxplots graph OLS method has more bias, as it is Q2= 2.25 and it is more far away from true $\beta_1 = 1$ than Q2 of Huber Loss, Q2=1.1. So Q2 of Huber Loss is closer to true value of $\beta_1 = 1$. Also the range for β_1 of Huber Loss is approximately [0.2, 2], while range for β_1 of OLS is approximately [1.4, 3.2]. So we can see from boxplots range of OSL boxplot doesn't even contain true value of $\beta_1 = 1$. After calculating bias for β_1 2 methods, For OSL bias of $\beta_1 = 1.22147105511238$, which is larger than for Huber Loss bias of $\beta_1 = 0.0592877704629744$. I expected that bias of OSL will be greater than bias of Huber Loss, because if we look at Huber loss function we can see that it assigns much less value when $y_i - X_i^T\beta$ is more extreme. As we have outliers $Y_{min} = -30$ the loss of corresponding $y_i - X_i^T\beta$ is $\ell_{\delta=1}(y_{min} - X_i^T\beta) = |y_{min} - X_i^T\beta| - \frac{1}{2}$, while simple regression will assign more value of β_1, when y is outlier. Because $\beta_1 = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2}$. So if there y is outlier $(y_i - \bar{y})$ will become larger and β_1 is estimated father from β_{true}.

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 ℓ_2 and ℓ_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:

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

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

- In this equation, the first line is the model fitted with scaled and centered data. And we obtain the fitted parameters as γ_j's
- In the second line, the coefficients β_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 γ_j's.
- Recover the original parameters β_0 and β_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.

```r
#osl betas
osl2_orig<-lm(y_org~X_org)
B_ols=osl2_orig$coefficients

#function for srandardization
center_scale<-function(X){
 (1/sd(X))*(X-mean(X))
}

#X_scaled and Y_scaled
X_scaled = matrix(0, nrow(X_org), ncol(X_org))

for(i in (1:ncol(X_org))){
  X_scaled[ ,i]=center_scale(X_org[,i])
}

Y_scaled=center_scale(y_org)

#get Betas for scaled X and Y
osl2_scaled<-lm(Y_scaled~X_scaled)
B_ols_scaled=osl2_scaled$coefficients

#calculate initial B0 using scaled Beta parameters
B_recovery=rep(0,4)
B_recovery[1]=mean(y_org)-(mean(X_org[ ,1])*sd(y_org)*B_ols_scaled[2])/sd(X_org[ ,1])-
  (mean(X_org[ ,2])*sd(y_org)*B_ols_scaled[3])/sd(X_org[ ,2])-
  (mean(X_org[ ,3])*sd(y_org)*B_ols_scaled[4])/sd(X_org[ ,3])

#calculate B1, B2, B3 using scaled Beta parameters
for(i in (2:4)){
  B_recovery[i]=(sd(y_org)*B_ols_scaled[i])/sd(X_org[ ,i-1])
}

#create table of Betas from different methods
B<-matrix(c(B_ols, B_ols_scaled, B_recovery), nrow = 3, byrow = TRUE)
Beta<-as.data.frame(B)
colnames(Beta)<-c("B0", "B1", "B2", "B3")
rownames(Beta)<-c("OLS", "OLS Scaled", "OLS Recovered")

Beta
```

	B0 <dbl>	B1 <dbl>	B2 <dbl>	B3 <dbl>
OLS	-5.165936e-01	0.6592889	2.1828432	0.24983631
OLS Scaled	5.237722e-17	0.2313313	0.8114034	0.06753726
OLS Recovered	-5.165936e-01	0.6592889	2.1828432	0.24983631

3 rows

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 β_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

β's) of residual r before and after this update. Based on our lecture, how to make the update of r computationally efficient?

Answer

$L(\beta) = \|y - X\beta\|^2$

We can rewrite $X\beta$ as $X\beta = X_j\beta_j + X_{-j}\beta_{-j}^{(k)}$, where X_j is j^{th} column of X, β_j is j^{th} entry of β vector, X_{-j} is design matrix after removing j^{th} column from X, and $\beta_{-j}^{(k)}$ is k^{th} iteration β vector after removing j^{th} entry.

$$L(\beta) = \|y - X_j\beta_j - X_{-j}\beta_{-j}^{(k)}\|^2$$

$$r = y - X_{-j}\beta_{-j}^{(k)}$$

$$L(\beta) = \|r - X_j\beta_j\|^2 = (r - X_j\beta_j)^T(r - X_j\beta_j)$$

Take derivative of $L(\beta)$ with respect to β_j.

$$\frac{\partial L(\beta)}{\partial \beta_j} = 2X_j^T(r - X_j\beta_j)$$

Set $\frac{\partial L(\beta)}{\partial \beta_j} = 0$

$$2X_j^T(r - X_j\beta_j) = 0$$

$$2X_j^T r = 2X_j^T X_j\beta_j$$

$$\beta_j = \frac{X_j^T r}{X_j^T X_j}$$

Before the update:

$$r = y - X_{-j}\beta_{-j}^{(k)}$$

After the update:

$$r^{new} = r - X_j\beta_j^{(k+1)} + X_{j+1}\beta_{j+1}^{(k)}$$

Based on our lecture, it is better to use above formula to compute r computationally efficient.

```
- [30 pts] ii) Implement this coordinate descent method with your own code to solve OLS with th
e scaled data. Print and report your **scaled coefficients** (no need to recover the original v
ersion) and compare with the result from the previous question.
  - 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
iteration will loop through all variables). However, stop your algorithm if the $\beta$ value o
f the current iteration is sufficiently similar to the previous one, i.e., $\|\beta^{(k)}- \bet
a ^{(k-1)}\|_1 \leq$ tol. Set `tol = 1e-7` where $\| \cdot \|_1$ is the L1 distance.
```

```
#coordinate descent function
coordinate_descent<-function(X,y, tol, max_iter){

  beta_matrix=matrix(0,ncol(X), max_iter)
  i=1
  beta=rep(0, ncol(X))
  loss=rep(0, max_iter)

  # calculate k-th beta vectors
  while(i<max_iter){
    for(j in (1:ncol(X))){
      X_j=X[ ,j]
      X_w_j= X[, -j]
      b_w_j=beta[-j]

      beta[j]=(t(X_j)%*%(y- X_w_j%*%b_w_j))/(t(X_j)%*%X_j)
      beta_matrix[j,i]=beta[j]
    }

    #calculate loss of i-th Beta vector
    loss[i]=(t(y-X%*%beta)%*%(y-X%*%beta))

    if(i>1){
      #stop loop when L1 of the difference  of beta previous and
      #current beta vectors < tolerance

      if(sum(abs(beta_matrix[, i]-beta_matrix[, i-1]))<tol){
        break
      }
    }
    i=i+1
  }

  #return last iteration, vector of loss and last beta vector
  results <- list(beta=beta_matrix[, i], loss_fn=loss[1:i], iter= i)
  return( results)
}

results=coordinate_descent(X_scaled,Y_scaled, 0.0000001, 100)

#print results
print(paste0("From coordinate descent method B_1 = ", results$beta[1]))
```

```
## [1] "From coordinate descent method B_1 = 0.231331300124625"
```

```
print(paste0("From coordinate descent method B_2 = ",results$beta[2]))
```

```
## [1] "From coordinate descent method B_2 = 0.811403365458755"
```

```
print(paste0("From coordinate descent method B_3 = ",results$beta[3]))
```

```
## [1] "From coordinate descent method B_3 = 0.0675372513325592"
```

```
print(paste0("From fitting lm() with scaled X and y B_1 = ",B_ols_scaled[2]))
```

```
## [1] "From fitting lm() with scaled X and y B_1 = 0.231331283807343"
```

```
print(paste0("From fitting lm() with scaled X and y B_2 = ",B_ols_scaled[3]))
```

```
## [1] "From fitting lm() with scaled X and y B_2 = 0.811403369317911"
```

```
print(paste0("From fitting lm() with scaled X and y B_3 = ",B_ols_scaled[4]))
```

```
## [1] "From fitting lm() with scaled X and y B_3 = 0.0675372551749632"
```

```
results$loss_fn[results$iter]
```

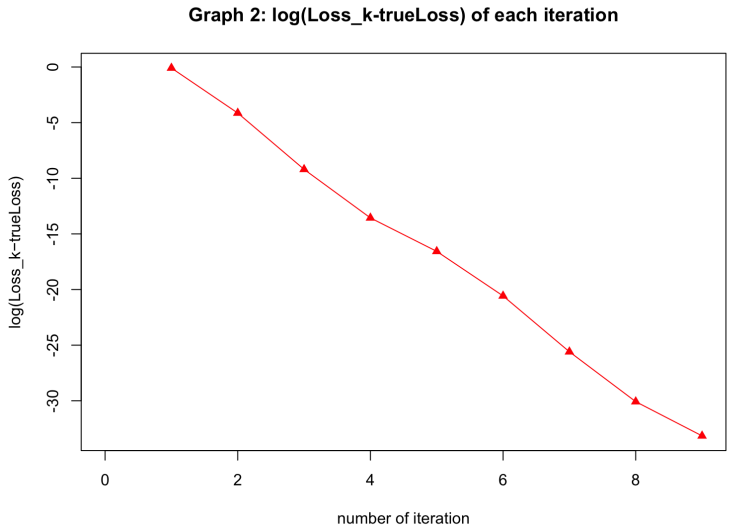
```
## [1] 2.878723
```

Comment

Using coordinate descent method I got that $B_1 = 0.231331300124625$, $B_2 = 0.811403365458755$, $B_3 = 0.0675372513325592$. Fitting lm() with normalized X and y I got $B_1 = 0.231331283807343$, $B_2 = 0.811403369317911$, $B_3 = 0.0675372551749632$. So Betas from 2 different methods are very close to each other.

```
-  [5 pts] Make a plot to analyze the convergence of the coordinate descent. On the x-axis, we use the number of iteration. On the y-axis, use $\log(\text{Loss}_k - \text{trueLoss})$. Here, \text{trueLoss} is the emperical loss based on the true optimizor, which we can simply use the solution from the `lm()` function (the scaled version). The $\text{loss}_k$ is the loss function at the begining of the $k$-th iteration (Keep in mind that within each iteration, we will loop over all $\beta_j$). If this plot displays a stragiht line, then we say that this algorithm has a linear convergence rate. Of course, this is at the log scale.
```

```
#trueLoss=mean(osl2_scaled$residuals^2)
trueLoss=t((Y_scaled-X_scaled%*%B_ols_scaled[2:4]))%*%(Y_scaled-X_scaled%*%B_ols_scaled[2:4])

#plot graph of log(Loss_k-trueLoss)
x=1:results$iter
y=log(results$loss_fn-trueLoss)
plot(x,log(results$loss_fn-trueLoss), type = "l",xlim=range(c(0,results$iter)), col="red", main ="Graph 2: log(Loss_k-trueLoss) of each iteration", xlab = "number of iteration", ylab ="log(Loss_k-trueLoss)")
points(x,log(results$loss_fn-trueLoss), pch=17, col="red")
```



Comment

The coordinate descent method has a linear convergence rate, as there is straight line when we draw graph $log(Loss_k - trueLoss)$ against number of iteration (Graph 2).