Hwk8_bs2996

Bangda Sun

November 12, 2016

1. Examine the correltion coefficients

Run the following code first

```
n <- 100
p <- 10
s <- 3
set.seed(0)
x <- matrix(rnorm(n*p), n, p)
b <- c(-0.7, 0.7, 1, rep(0, p - s))
y <- x %*% b + rt(n, df = 2)
```

Then calculate the correlation coefficient between the column of x and y

```
cor(x, y)
```

```
##                [,1]
##  [1,] -0.2526434175
##  [2,]  0.1239284685
##  [3,]  0.1673840288
##  [4,] -0.2522804417
##  [5,] -0.0371161818
##  [6,]  0.1561141420
##  [7,] -0.1175268150
##  [8,] -0.0899681839
##  [9,] -0.0002104895
## [10,]  0.0506851086
```

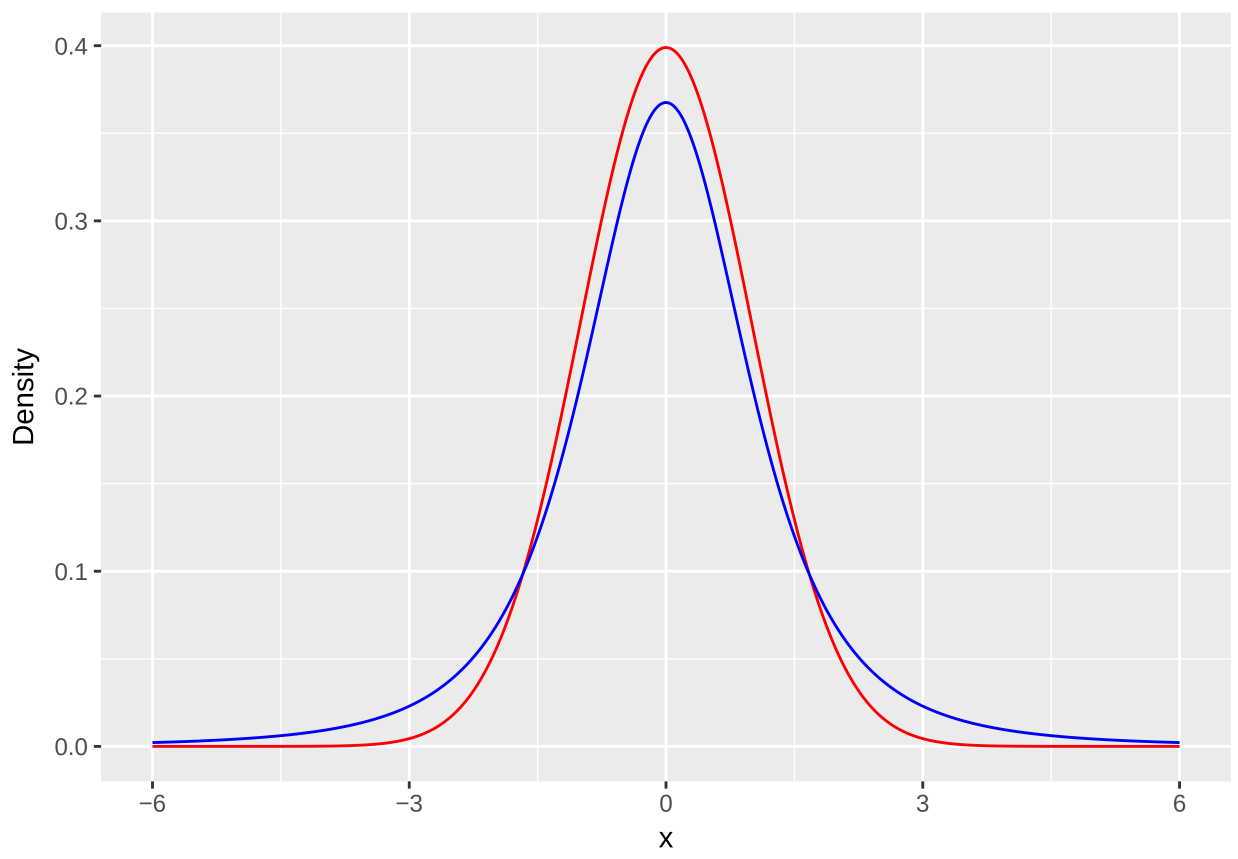
We can see that the first three coefficients are -0.252, 0.124, 0.167, but we cannot clearly figure out them just based on the coefficients (if we just know there are three relevent variables but don't know their position).

2. Plotting normal distribution (standard) and t distribution

```
library(ggplot2)
```

```
## Warning: package 'ggplot2' was built under R version 3.3.2
```

```
xt <- seq(-6, 6, by = .001)
pdfdata <- data.frame(xt = xt, nd = dnorm(xt), td = dt(xt, df = 3))
ggplot(data = pdfdata) +
  geom_line(mapping = aes(x = xt, y = nd), col = "red") +
  geom_line(mapping = aes(x = xt, y = td), col = "blue") +
  labs(x = "x", y = "Density")
```



We can see that t-distribution has a thicker tail than normal distribution.

3. Write function huber.loss()

```
huber.loss <- function(beta){
  # input vector of coefficient
  # return the sum of psi applied to residuals
  r <- y - x %*% beta
  c <- 1
  return(sum(ifelse(r^2 > c^2, 2*c*abs(r) - c^2, r^2)))
}
```

4. Use gradient descent to minimize huber.loss()

```
# first let's introduce function which applies gradient descent
library(numDeriv)
```

```
## Warning: package 'numDeriv' was built under R version 3.3.2
```

```
grad.descent <- function(f, x0, max.iter = 200, step.size = 0.05, stopping.deriv = 0.01, ...) {
  # f: function need to be minimize
  # x0: initial point
  # max.iter
  # setp.size: eta
  # stopping.derive
  n    <- length(x0)
  # store every parameters in iteration
  xmat <- matrix(0, nrow = n, ncol = max.iter)
  xmat[,1] <- x0

  for (k in 2:max.iter) {
    # Calculate the gradient
    grad.cur <- grad(f, xmat[ ,k-1], ...)

    # Should we stop?
    if (all(abs(grad.cur) < stopping.deriv)) {
      k <- k-1; break
    }

    # Move in the opposite direction of the grad
    xmat[ ,k] <- xmat[ ,k-1] - step.size * grad.cur
  }

  xmat <- xmat[ ,1:k] # Trim
  return(list(x = xmat[,k], xmat = xmat, k = k))
}
gd <- grad.descent(huber.loss, x0 = rep(0, p), step.size = 0.001, stopping.deriv = 0.1)
# the number of iteration
num_iter <- gd$k; num_iter
```

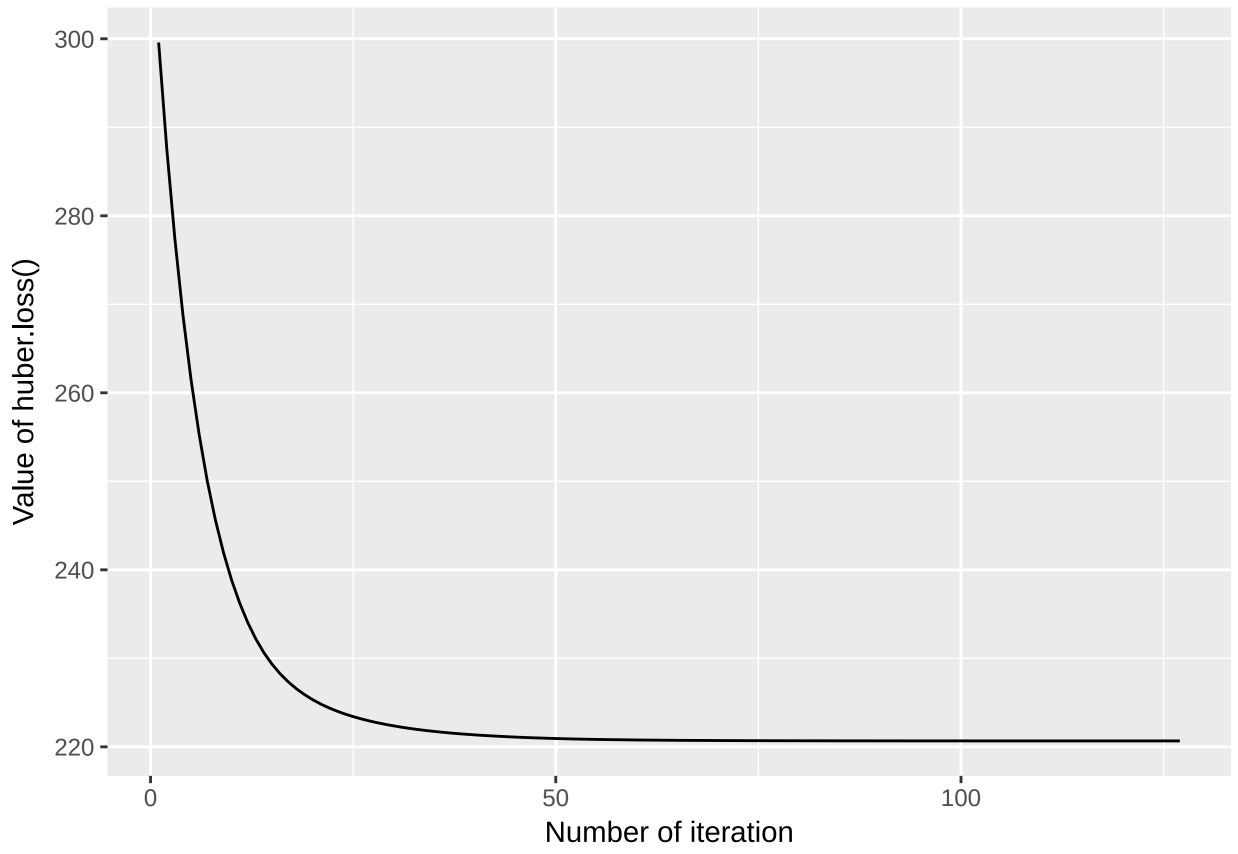
```
## [1] 127
```

```
# final coefficient
final_coef <- gd$x; final_coef
```

```
##  [1] -0.87346579  0.61828938  0.87989797 -0.04910821  0.07277491
##  [6]  0.10229815 -0.12513246 -0.14559243 -0.11903666 -0.02250130
```

5. Plotting the change of values in iteration

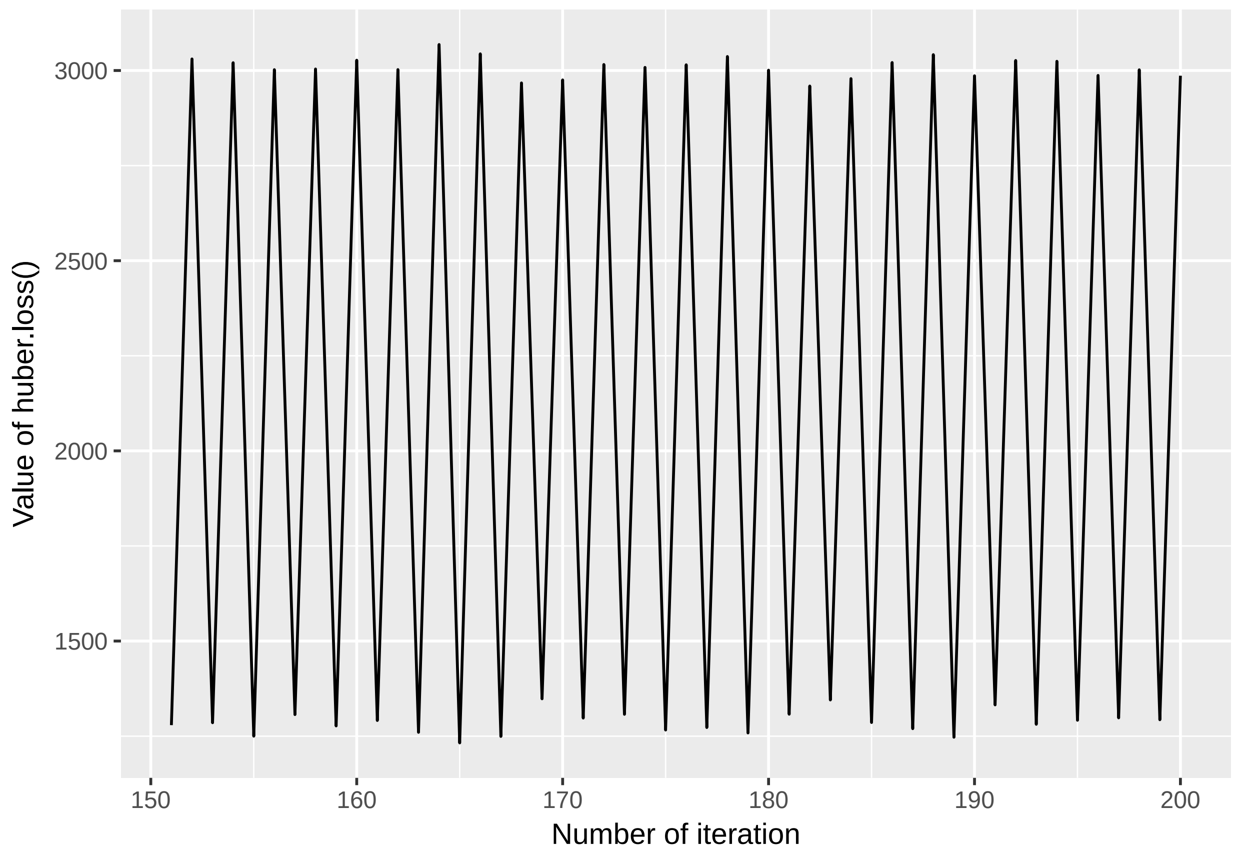
```
obj <- rep(NA, num_iter)
for (i in 1:num_iter){
  obj[i] <- huber.loss(gd$xmat[,i])
}
ggplot() +
  geom_line(mapping = aes(x = 1:num_iter, y = obj)) +
  labs(x = "Number of iteration", y = "Value of huber.loss()")
```



We can see at the early iteration, the value of huber.loss() is around 280 to 300, then it goes down exponentially and converges to around 220.

6. Rerun the gradient descent

```
gd2 <- grad.descent(huber.loss, x0 = rep(0, p), step.size = 0.1, stopping.deriv = 0.1)
obj2 <- rep(NA, 50)
for (i in 1:50){
  obj2[i] <- huber.loss(gd2$xmat[,i+49])
}
ggplot() +
  geom_line(mapping = aes(x = (gd2$k-49):gd2$k, y = obj2)) +
  labs(x = "Number of iteration", y = "Value of huber.loss()")
```



We can notice that there is a zig-zagging. The criterion isn't decreasing at each step and the gradient descent doesn't converge at the end. We can deduce that the coefficent estimates will also not converge.

7. Modify function grad.descent()

```r
sparse.grad.descent <- function(f, x0, max.iter = 200, step.size = 0.05, stopping.deriv = 0.01, ...) {
  # f: function need to be minimize
  # x0: initial point
  # max.iter
  # setp.size: eta
  # stopping.derive
  n    <- length(x0)
  # store every parameters in iteration
  xmat <- matrix(0, nrow = n, ncol = max.iter)
  xmat[,1] <- x0

  for (k in 2:max.iter) {
    # Calculate the gradient
    grad.cur <- grad(f, xmat[ ,k-1], ...)

    # Should we stop?
    if (all(abs(grad.cur) < stopping.deriv)) {
      k <- k-1; break
    }

    # Move in the opposite direction of the grad
    xmat[ ,k] <- xmat[ ,k-1] - step.size * grad.cur
    xmat[, k][abs(xmat[, k]) < 0.05] <- 0
  }

  xmat <- xmat[ ,1:k] # Trim
  return(list(x = xmat[,k], xmat = xmat, k = k))
}
gd.sparse <- sparse.grad.descent(huber.loss, x0 = rep(0, p), step.size = 0.001, stopping.deriv = 0.1)
# final coefficient estimates
gd.sparse$x
```

```
##  [1] -0.8944804  0.6332991  0.8823860  0.0000000  0.0000000  0.0000000
##  [7]  0.0000000  0.0000000  0.0000000  0.0000000
```

8. Compare three methods

```
regdata <- as.data.frame(cbind(y, x))
colnames(regdata) <- c("y", paste("x", "1":"10", sep = ""))
# estimates in the usual manner using lm()
b_lm <- lm(y ~.-1, data = regdata)$coefficient; b_lm
```

```
##            x1            x2            x3            x4            x5 
## -0.9477210986  0.4864220270  0.5875664655 -0.7416200316  0.0008874065 
##            x6            x7            x8            x9           x10 
##  0.3149846567 -0.3994729398 -0.2712937636 -0.1445449407  0.0788007924
```

```
# compute the mse of using lm()
mse_lm <- mean((b_lm - b)^2); mse_lm
```

```
## [1] 0.1186581
```

```
# compute the mse of question 4
mse_q4 <- mean((gd$x - b)^2); mse_q4
```

```
## [1] 0.01208955
```

```
# compute the mse of question 7
mse_q7 <- mean((gd.sparse$x - b)^2); mse_q7
```

```
## [1] 0.005610471
```

We can see that the estimates in question 7 is the best.

9. Run on different data

```
set.seed(10)
y <- x %*% b + rt(n, df = 2)
# in question 4
gd3 <- grad.descent(huber.loss, x0 = rep(0, p),
                    step.size = 0.001, stopping.deriv = 0.1)
gd3$x
```

```
##  [1] -0.46329748  0.92390614  0.92287242 -0.06526259  0.24633002
##  [6] -0.04406371  0.01858892 -0.18921630  0.19479185 -0.18395820
```

```
mse_gd <- mean((gd3$x - b)^2); mse_gd
```

```
## [1] 0.02869228
```

```
# in question 7
gd.sparse2 <- sparse.grad.descent(huber.loss, x0 = rep(0, p),
                                  step.size = 0.001, stopping.deriv = 0.1)
gd.sparse2$x
```

```
##  [1] 0.0000000 0.7850744 0.9398727 0.0000000 0.0000000 0.0000000 0.0000000
##  [8] 0.0000000 0.0000000 0.0000000
```

```
mse_sparsegd <- mean((gd.sparse2$x - b)^2); mse_sparsegd
```

```
## [1] 0.0500853
```

As we can see, gradient descent estimates has a lower MSE, therefore it gives a better estimates. Compared with previous estimates, sparse gradient descent does not perform well, it suggest that the variability of its estimates is high, which means the reliability of sparse gradient descent is not very good.

10. Repeat the experiment

```
mse_gd2 <- rep(NA, 10)
mse_sparsegd2 <- rep(NA, 10)
for (i in 1:10){
  y <- x %*% b + rt(n, df = 2)
  gd4 <- grad.descent(huber.loss, x0 = rep(0, p),
                      step.size = 0.001, stopping.deriv = 0.1)
  mse_gd2[i] <- mean((gd4$x - b)^2)
  gd.sparse3 <- sparse.grad.descent(huber.loss, x0 = rep(0, p),
                                    step.size = 0.001, stopping.deriv = 0.1)
  mse_sparsegd2[i] <- mean((gd.sparse3$x - b)^2)
}
# average
mean(mse_gd2)
```

```
## [1] 0.02495459
```

```
mean(mse_sparsegd2)
```

```
## [1] 0.02650818
```

```
# standard deviation
sd(mse_gd2)
```

```
## [1] 0.01006557
```

```
sd(mse_sparsegd2)
```

```
## [1] 0.02961213
```

```
# minimum
min(mse_gd2)
```

```
## [1] 0.01430856
```

```
min(mse_sparsegd2)
```

```
## [1] 0.0006265157
```

We can see the average MSE of gradient descent is lower, and the minimum of sparse gradient descent is lower. We can also report the standard deviation of the MSE and find sparse gradient descent has a high variability.