HW8_cq2203

Chang Qu (cq2203)

12/1/2018

1. Run the following code block to create synthetic regression data, with 100 observations and 10 predictor variables:

```
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)

data <- data.frame(cbind(x,y))

# obtain correlation between x and y
corrs <- apply(x,2,cor,y)
order(abs(corrs), decreasing = TRUE)
```

```
##  [1]  1  4  3  6  2  7  8 10  5  9
```

We want to pick the variable X that is the most correlated to the response Y. From the correlations above, we can see that they are X_1, X_3, X_4.

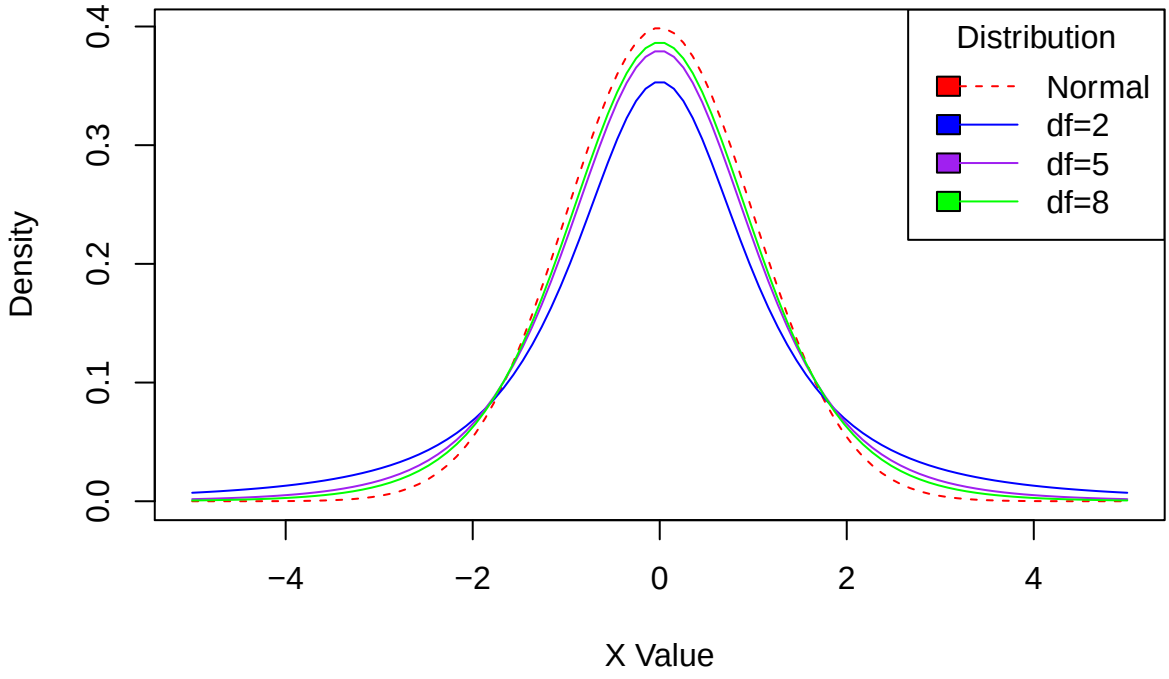
2. Plotting the normal density and the t-density on the same plot, with the latter having 3 degrees of freedom.

```
# df=3 or 3 different dfs?
df <- c(2,5,8)
color <- c("blue","purple","green")
label <- c("Normal","df=2","df=5","df=8")

x1 <- seq(-5,5,length.out = 100)
plot(x1, dnorm(x1), col="red",type = "l",lty=2, xlab = "X Value", ylab = "Density",
     main = "Comparison between Normal and t-distribution")

for(i in 1:3){
  lines(x1, dt(x1,df[i]), col=color[i])
}


legend("topright", title = "Distribution",label, col = c("red", color),
       fill = c("red", color), lty = c(2,1,1,1))
```



From the plot above, we can see that the t-distribution definitely have a thicker tail than the normal distribution.

3. Write a function called huber.loss() that takes in as an argument a coefficient vector beta, and returns the sum of psi() applied to the residuals (from regressing y on x).

```
huber.loss <- function(beta){
  yhat <- x%*%beta
  resi <- y-yhat
  psi <- function(r, c = 1) {
    return(ifelse(r^2 > c^2, 2*c*abs(r) - c^2, r^2))
    }
  return(sum(psi(resi)))
}
```

4. Store the output of grad.descent() in gd. How many iterations did it take to converge, and what are the final coefficient estimates?

```
library(numDeriv)

grad.descent <- function(f,x0, max.iter = 200,
                                stopping.deriv = 0.1, step.size = 0.001){
  iter <- 0
  grad.cur <- Inf
  xmat <- matrix(0, nrow = length(x0), ncol = max.iter+1)
  xmat[,1] <- x0
  for(i in 2:(max.iter+1)){
    iter <- iter + 1
    grad.cur <- grad(f, xmat[,i-1])
    xmat[,i] <- xmat[,i-1] - grad.cur*step.size
    if  (all(abs(grad.cur) < stopping.deriv)){break()}
```

```
  }
  f <- f(xmat[,i])
  result <- list(x = xmat[,i-1], global.min = f, x.step = xmat,
                 iteration = i-1, converged = (iter < max.iter) )
  return(result)
}

gd <- grad.descent(huber.loss, x0=rep(0,p))

# number of iterations to converge
gd$iteration
```

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

```
# final coefficient estimates
gd$x
```

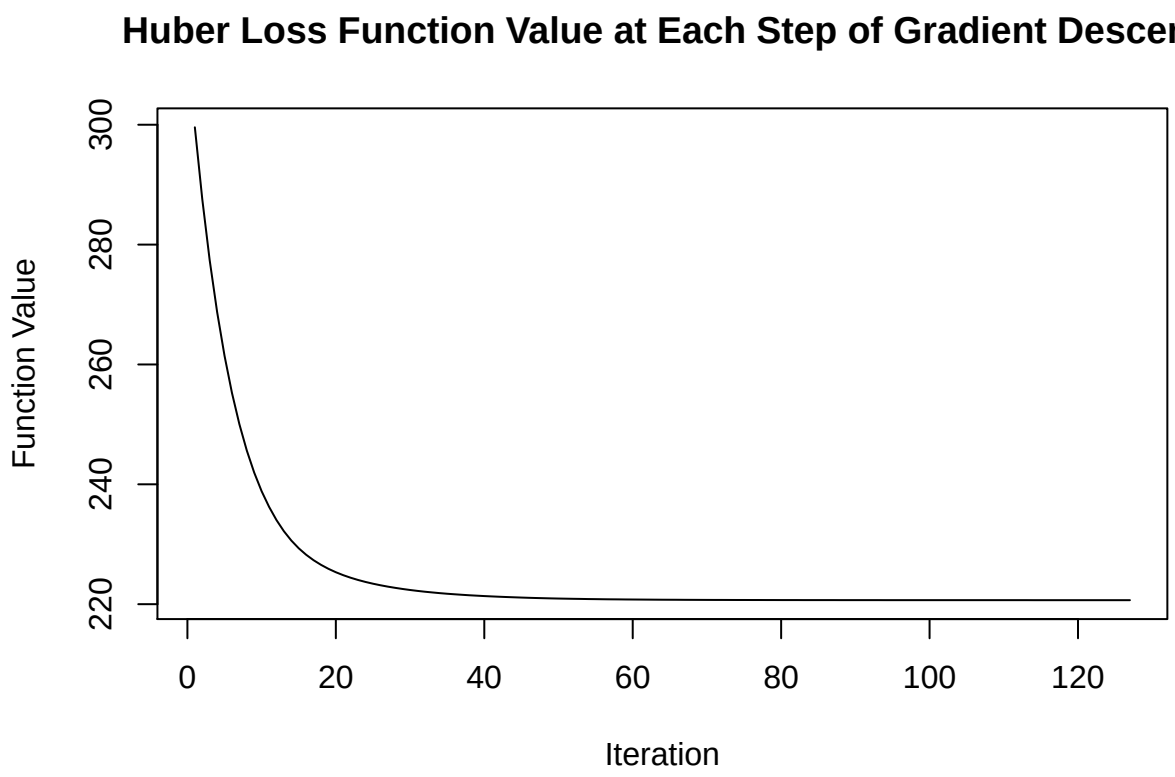
```
##  [1] -0.87346579  0.61828938  0.87989797 -0.04910821  0.07277491
##  [6]  0.10229815 -0.12513246 -0.14559243 -0.11903666 -0.02250130
```

It took 127 iterations to converge. And above output gave the final coefficient estimates.

5. Using gd, construct a vector obj of the values objective function encountered at each step of gradient descent. Plot these values against the iteration number, to confirm that gradient descent is indeed making the objective function at each iteration.

```
# get the value of hubert loss function at each iteration
obj <- apply(gd$x.step[,1:gd$iteration],2, huber.loss)
#obj

# plot these value against iteration
plot(1:gd$iteration, obj, xlab = "Iteration", ylab = "Function Value", main = "Huber Loss Function Valu
```



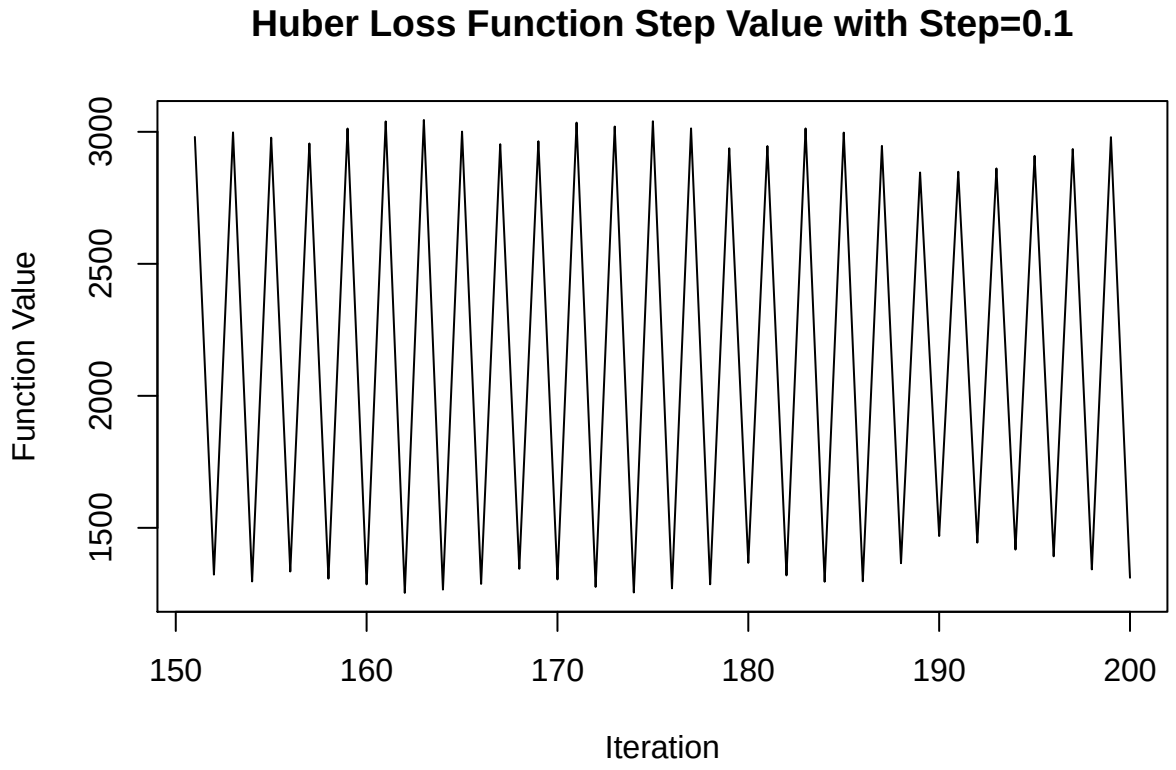
In the early iterations, the algorithm progresses significantly, with each step make a big decrease in the function value. As moving to later iterations, the progress start to become not so obvious and insignificant.

6. Rerun gradient descent as in question 4, but with step.size = 0.1.

```
# apply again using step size=0.1
gd2 <- grad.descent(huber.loss, rep(0,p),max.iter = 200,stopping.deriv = 0.1, step.size = 0.1)

# evaluate object function value at each step
obj2 <- apply(gd2$x.step[,1:gd2$iteration],2, huber.loss)

# plot the last fifty
index <- (gd2$iteration-49):gd2$iteration
plot(index, obj2[index], xlab = "Iteration", ylab = "Function Value", main = "Huber Loss Function Step V
```



Under this situation, the plot flucuates through iterations. The objective function value is not decreasing at each step. Instead, it increases at some steps then decreases, and repeat.

```
# to see of converged
gd2$converged
```

```
## [1] FALSE
```

```
gd2$x
```

```
##  [1]  1.0740298 -0.7971898  2.8860325 -1.8822687  2.1897562  0.8721260
##  [7] -1.0055026 -1.5049278  0.9241456  4.7508245
```

```
gd2$iteration
```

```
## [1] 200
```

And we can see that the gradient descent did not converge in the end.

```
# check the x.step in gd and gd2
gd$x.step[,1:5]
```

```
##       [,1]         [,2]         [,3]         [,4]         [,5]
##  [1,]    0 -0.057333921 -0.110579829 -0.159594790 -0.204266836
##  [2,]    0  0.041772705  0.081319260  0.118460144  0.153250180
##  [3,]    0  0.069072391  0.134130588  0.195155692  0.252503191
##  [4,]    0 -0.027494865 -0.051034153 -0.070570818 -0.086468331
##  [5,]    0 -0.017602248 -0.029911400 -0.037434068 -0.041221526
##  [6,]    0  0.034044352  0.064721825  0.092226665  0.116679834
##  [7,]    0 -0.012249313 -0.024290334 -0.036154803 -0.047245096
##  [8,]    0 -0.010539868 -0.019859263 -0.028033369 -0.035210482
##  [9,]    0  0.000947101  0.001949435  0.002587417  0.002600211
## [10,]    0  0.011872905  0.022540225  0.032307821  0.041373715
```

```
gd2$x.step[,1:5]
```

```
##       [,1]       [,2]       [,3]       [,4]        [,5]
##  [1,]    0 -5.7333921  0.2385443 -3.4761031 -2.06901588
##  [2,]    0  4.1772705  0.3337634 -0.0438562  1.22771917
##  [3,]    0  6.9072391 -3.2113741  5.6546730 -4.25390855
##  [4,]    0 -2.7494865  1.4065462 -3.1140734  0.41091614
##  [5,]    0 -1.7602248  0.2066779  0.9124582 -4.13723932
##  [6,]    0  3.4044352 -3.4886602  2.4086685  0.02019723
##  [7,]    0 -1.2249313 -1.1592270 -1.5794785  0.38995079
##  [8,]    0 -1.0539868  2.6858237 -4.6366824  2.39510015
##  [9,]    0  0.0947101 -4.6206763  3.6801790 -3.16069955
## [10,]    0  1.1872905 -2.1245342  4.9868710 -7.00246120
```

By comparing the x.step matrix in the previous gradient descent, we can see that the estimated coefficients are not reaching to the point where they minimize the huber loss function.

7. Modify the function grad.descent() so that at every iteration k, after taking a gradient step but before saving the new estimated coefficients, we threshold small values in these coefficients to zero. Here small means less than or equal to 0.05, in absolute value.

```
sparse.grad.descent <- function(f,x0, max.iter = 200,
                                stop.deriv = 0.1, step.size = 0.001){
  iter <- 0
  grad.cur <- Inf
  xmat <- matrix(0, nrow = length(x0), ncol = max.iter)
  xmat[,1] <- x0
  for(i in 2:max.iter){
    iter <- iter + 1
    grad.cur <- grad(f, xmat[,i-1])
    update <- xmat[,i-1] - grad.cur*step.size
    update[abs(update)<=0.05]<-0
    xmat[,i] <- update
    if  (all(abs(grad.cur) < stop.deriv)){break()}
  }
  f <- f(xmat[,i])
  result <- list(x = xmat[,i-1], global.min = f, x.step = xmat,
                 iteration = iter, converged = (iter < max.iter) )
  return(result)
}
```

```
gd.sparse <- sparse.grad.descent(huber.loss, x0=rep(0,p))
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
```

Output above gave the estimated coefficient, which has the last 7 all equal to zero. It is much more accurate than the basic gradient descent.

8. Now compute estimates of the regression coefficients in the usual manner, using lm(). How do these compare to those from question 4, from question 7? Compute the mean squared error between each of these three estimates of the coefficients and the true coefficients b. Which is best?

```
mse.coef <- function(beta){
  mean((b-beta)^2)
}


est_lm <- coef(lm(y~x-1))
est_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
```

```
mse.coef(est_lm)
```

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

```
mse.coef(gd$x)
```

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

```
mse.coef(gd.sparse$x)
```

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

By computing the MSE, we can see that the estimated coefficient we obtain using sparse gradient descent has the smallest MSE from the true coefficients b. Therefore, the sparse gradient descent has the best estimates.

9. Rerun your Huber loss minimization in questions 4 and 7, but on different data.

```
set.seed(10)
y = x%*% b + rt(n, df=2)

# gradient descent
gd_y2 <- grad.descent(huber.loss, rep(0,p))
gd_y2$x
```

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

```
# sparse gradient descent
gd.sparse_y2 <- sparse.grad.descent(huber.loss, rep(0,p))
gd.sparse_y2$x
```

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

```r
# mean squared error
mse.coef(gd_y2$x)
```

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

```r
mse.coef(gd.sparse_y2$x)
```

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

By looking from the output above, we can see that by using another set of data, sparse gradient descent has the estimate for the first coefficient also as zero, which should originally close to -0.7. It result in a bigger MSE.

This induce that the variability of sparse gradient descent is bigger, because sparse gradient descent will assign the estimates that is smaller than 0.05 to 0, which sometimes will give the estimation of a non-zero coefficient to 0. Therefore, we will have a bigger residual under this situation, and will result in a higher variance for the estimates.

10. Repeat the experiment from question 9, generating 10 new copies of y, running gradient descent and sparse gradient descent, and recording each time the mean squared errors of each of their coefficient estimates to b.

```r
set.seed(10)
ymat <- matrix(0, nrow = n, ncol = 10)
mse.mat <- matrix(0, nrow = 2, ncol = 10)
for(i in 1:10){
  y <- x%*% b + rt(n, df=2)
  gd <- grad.descent(huber.loss, rep(0,p))
  gd.sparse <- sparse.grad.descent(huber.loss, rep(0,p))
  mse.mat[1,i] <- mse.coef(gd$x)
  mse.mat[2,i] <- mse.coef(gd.sparse$x)
}

apply(mse.mat,1,mean)
```

```
## [1] 0.02639296 0.03108199
```

```r
min.index <- apply(mse.mat,1,which.min)
mse.mat[,min.index]
```

```
##              [,1]         [,2]
## [1,] 0.015224697 0.0221191368
## [2,] 0.006021432 0.0006265158
```

From the output above, we see that the mean MSE for gradient descent is the smaller than that of the sparse gradient descent. However, the minimum MSE of sparse gradient descent is smaller than that of the regular gradient descent. This in lines with the iterpretation that sparse gradient descent has greater variability.