Homework Six

Christine Chong cc4190

June 25, 2017

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

```
library(numDeriv)
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)
```

Notice that only 3 of the 10 predictor variables in total are actually relevant in predicting the response. (That is, only the first three coefficients in b are nonzero.) Examine the correlation coefficients between predictor variables x and the response y; would you be able to pick out each of the 3 relevant variables based on correlations alone?

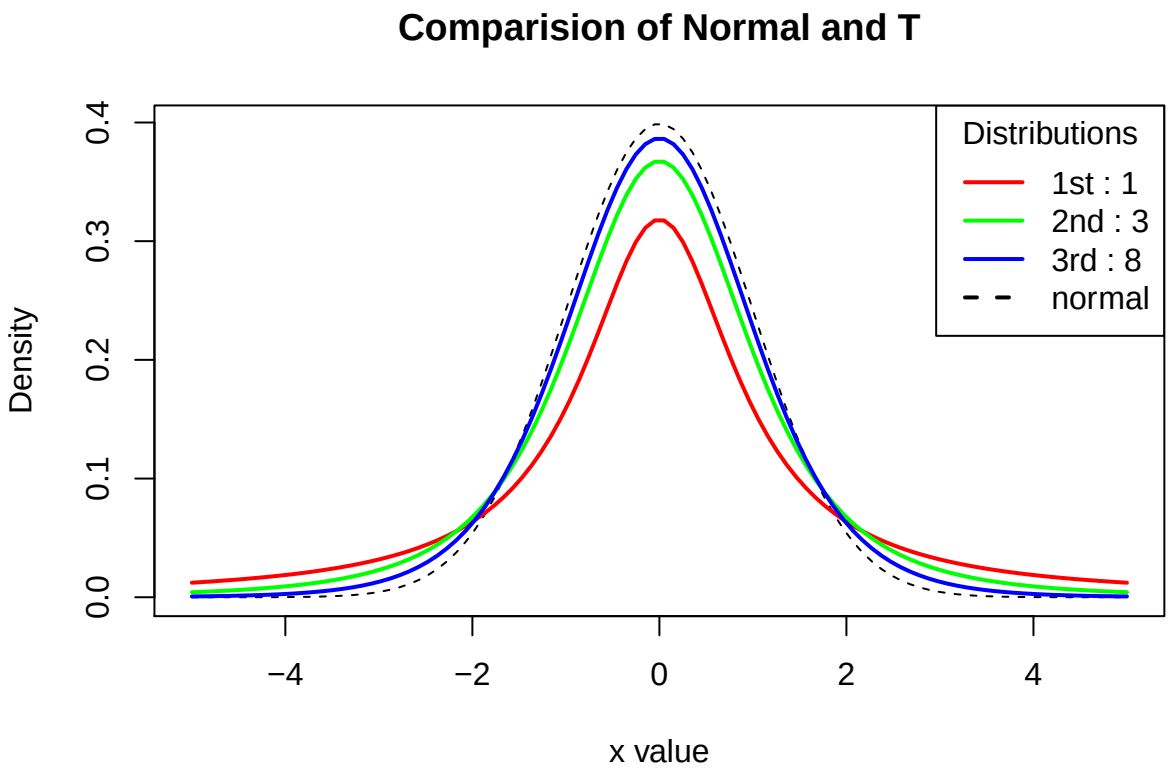
```
cors <- apply (x, 2 , cor, y)
cors
```

```
##  [1] -0.2526434175  0.1239284685  0.1673840288 -0.2522804417 -0.0371161818
##  [6]  0.1561141420 -0.1175268150 -0.0899681839 -0.0002104895  0.0506851086
```

I don't think it is easy to tell what the 3 relevant variables are going to be based on the correlation alone. We don't know if our function is minimized.

2. Note that the noise in the above simulation (the difference between y and x %*% b) was created from the rt() function, which draws t-distributed random variables. The tdistribution has thicker tails than the normal distribution, so we are more likely to see large noise terms than we would if we used a normal distribution. Verify this by plotting the normal density and the t-density on the same plot, with the latter having 3 degrees of freedom. Choose the plot ranges appropriately, and draw the densities in different colors, so that the plot is easy to read.

```
a<- seq(-5,5, length = 100)
ha <-dnorm(a)
degf <- c(1,3,8)
colors <- c("red", "green", "blue", "black")
labels <- c("1st : 1", "2nd : 3", "3rd : 8", "normal")
plot(a,ha, type ="l", lty=2, xlab="x value", ylab="Density", main ="Comparision of Normal and T")
for(i in 1:length(degf)){
  lines(a, dt(a, degf[i]), lwd = 2, col = colors[i])
}
legend("topright", title = "Distributions", labels, lwd=2, lty = c(1,1,1,2), col = colors )
```



3. Because we know that the noise in our regression has thicker tails than the normal distribution, we are more likely to see outliers. Hence we're going to use the Huber loss function, which is more robust to outliers:

```
psi <- function(r, c = 1) {
return(ifelse(r^2 > c^2, 2*c*abs(r) - c^2, r^2))
}
```

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). x and y should not be provided as arguments, but referred to directly in the function. You may stick with the default cutoff of c=1. This Huber loss is going to take the place of the usual (nonrobust) linear regression loss, i.e., the sum of squares of the residuals.

```
huber.loss<- function(beta){
  model <- y - x %*% beta
  return(sum(psi(model)))
}
```

4. Using the grad.descent() function from lecture, run gradient descent starting from beta = rep(0, p), to get an estimate of the coefficients beta that minimize the Huber loss, when regressing y on x. Use the settings max.iter = 200, step.size = 0.001, and stopping.deriv = 0.1. Store the output of grad.descent() in gd. How many iterations did it take to converge, and what are the final coefficient estimates? Note: you may need to run install.packages("numDeriv") in order to load the numDeriv library.

```
grad.descent <- function(f, x0, max.iter = 200, step.size = 0.001, stopping.deriv = 0.1, ...) {


  n    <- length(x0)
```

```
  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,
              minimum=f(xmat[,k],...)
  )
  )
}
beta <- as.vector(rep(0,p))
gd<- grad.descent(huber.loss, beta)
gd$k
```

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

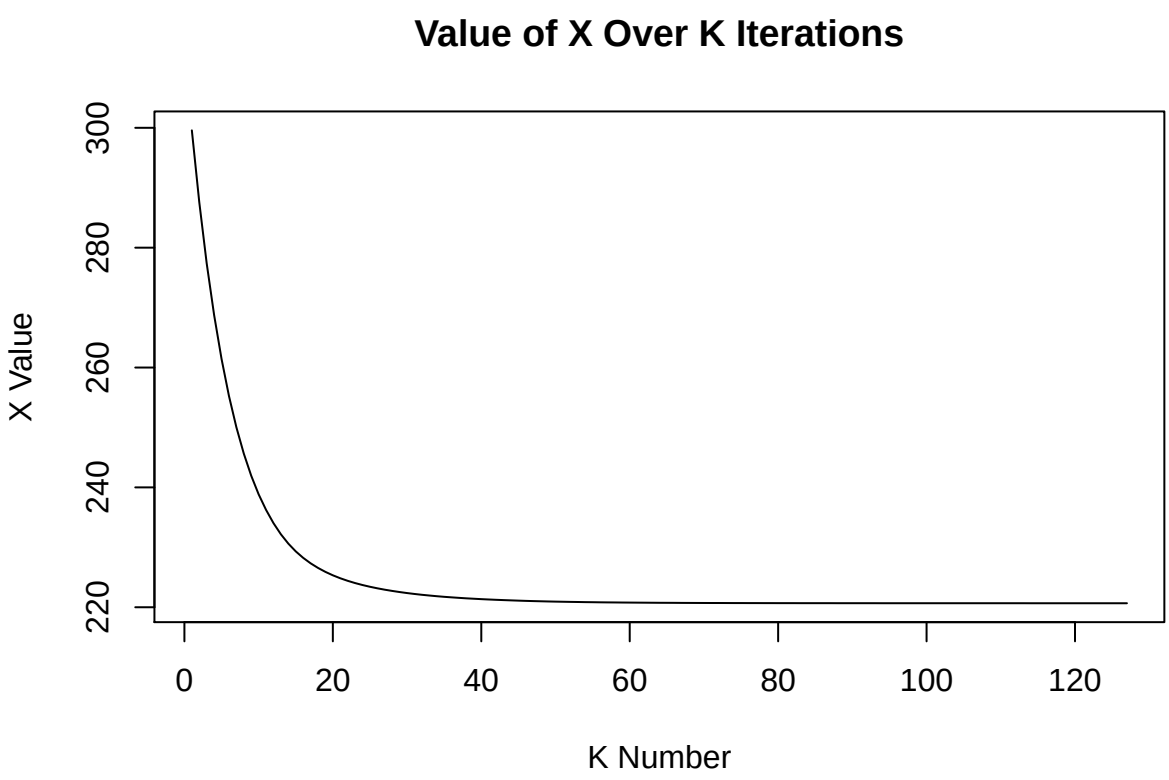
```
gd$x
```

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

It took around 127 iterations to converge and the final coefficient estimates are -0.8734658, 0.6182894, 0.879898, -0.0491082, 0.0727749, 0.1022982, -0.1251325, -0.1455924, -0.1190367, -0.0225013

5. Using gd, construct a vector obj of the values objective function encountered at each step of gradient descent. Note: here the objective function for minimization is the Huber loss. Plot these values against the iteration number, to confirm that gradient descent is indeed making the objective function at each iteration. How does the progress of the algorithm compare at the start (early iterations) versus towards the end (later iterations)?

```
gdx<- gd$x
obj <- apply(gd$xmat[, 1:gd$k], 2, huber.loss)
plot(1:gd$k, obj[1:gd$k], xlab = "K Number", ylab ="X Value", type = "l", main = "Value of X Over K Ite
```



It seems like the more iterations there are, the smaller the x values are. This just shows that gradient descent does work.

6. Rerun gradient descent as in question 4, but with step.size = 0.1. Compute the new criterion values across iterations, and plot the last fifty criterion values. What do you notice now? Is the criterion decreasing at each step, and has gradient descent converged at the end (settled on a single criterion value)? What can you deduce from your plot is happening to the coefficient estimates (confirm this by looking at the xmat values in gd)?

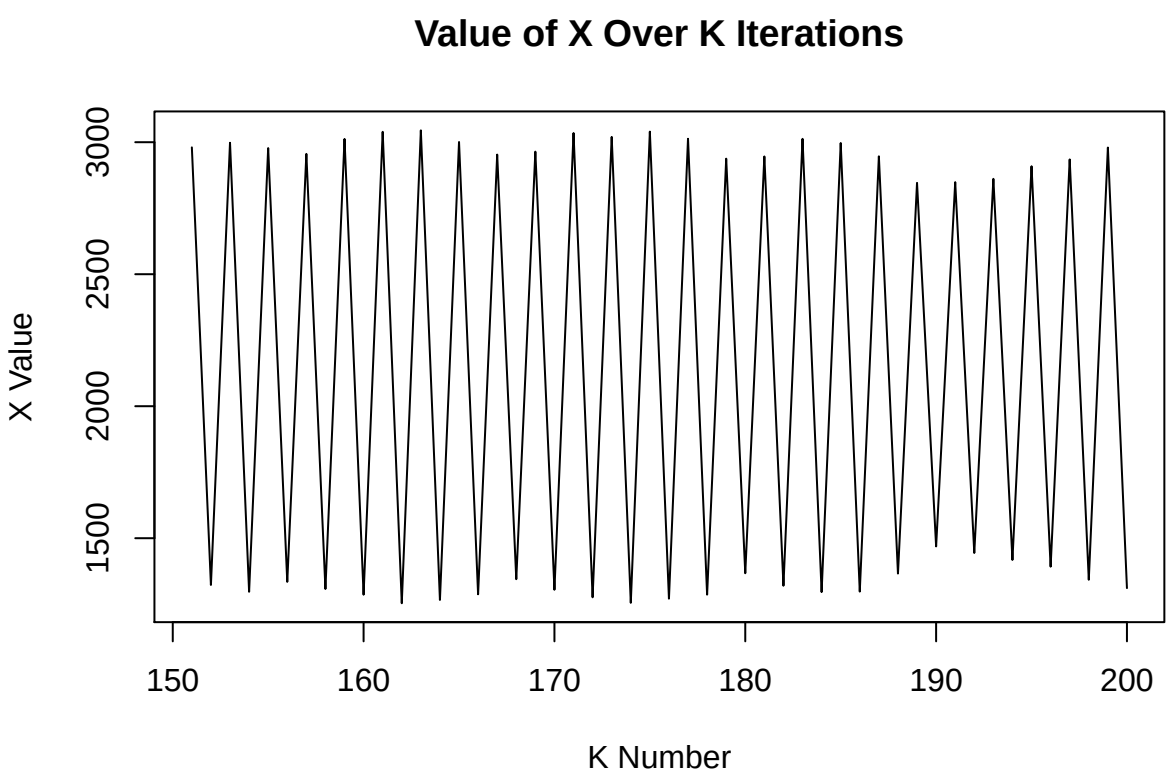
```
gd2<- grad.descent(huber.loss, beta, max.iter = 200, step.size = 0.1)
gd2$k
```

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

```
gd2$x
```

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

```
obj <- apply(gd2$xmat[, 1:gd2$k], 2, huber.loss)
plot((gd2$k-49):gd2$k, obj[(gd2$k- 49):gd2$k], xlab = "K Number", ylab ="X Value", type = "l", main = "
```



Here there is a lot of oscillation, so the gradient descent cannot figure out the smallest value of x. This is probably because it is swithcing back and forth between a high value and a low value. This means that the step size is too big.

7. Inspect the coefficients from the first gradient descent run (stored in gd), and compare them to the true (unknown) underlying coefficients b constructed in question 1. They should be pretty close for the first 3 variables, but the next 7 are not very accurate-that is, they're not all close to 0, as they should be. In order to fix this, we're going to apply a sparsified version of gradient descent (formally known as proximal gradient descent). 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. Call the new function sparse.grad.descent() and rerun with the same settings as in question 4, in order to produce a sparse estimate of the regression coefficients. Stores the results in 2gd.sparse. What are the final coefficient estimates?

```
sparse.grad.descent <- function(f, x0, max.iter = 200, step.size = 0.001, stopping.deriv = 0.01, ...) {

  n    <- length(x0)
  xmat <- matrix(0, nrow = n, ncol = max.iter)
  xmat[,1] <- x0

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

    if (all(abs(grad.cur) < 0.05)) {
      k <- k-1; break
    }
```

```
    sparse <- xmat[ ,k-1] - step.size * grad.cur
    sparse[abs(sparse)< 0.05] <- 0
    xmat[,k] <- sparse

  }

  xmat <- xmat[ ,1:k]
  return(list(x = xmat[,k], xmat = xmat, k = k))
}
gd$x
```

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

```
b
```

```
##  [1] -0.7  0.7  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

```
beta = rep(0,p)
gd2.sparse<- sparse.grad.descent(huber.loss, beta)
gd2.sparse$x
```

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

The final coefficient estimates are -0.8944804, 0.6332991, 0.882386, 0, 0, 0, 0, 0, 0, 0 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 coeffi- cients b. Which is best?

```
coef(lm(y~x))
```

```
## (Intercept)          x1          x2          x3          x4          x5
## -0.33722705 -0.93698042  0.46526790  0.62053171 -0.74473066 -0.01110637
##          x6          x7          x8          x9         x10
##  0.28837271 -0.42313741 -0.24659171 -0.17223446  0.06106860
```

```
gd$x
```

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

```
gd2.sparse$x
```

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

Here the Sparse Gradient Descent has gotten the closest values to the true coefficients of b.

9. Rerun your Huber loss minimization in questions 4 and 7, but on different data. That is, just generate another copy of y, per the same formula as you used in question 1: y = x %*% b + rt(n, df=2). How do the new coefficient estimates look from gradient descent, and sparsified gradient descent? Which has a better mean squared error when measured against the b used to generate data in question 1? What do you deduce about the sparse method (e.g., what does this suggest about the variability of its estimates)? In order to ensure that your results are comparable to other students', please run the following before generating a new y vector:

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

```
beta = rep(0,p)
gd3<-grad.descent(huber.loss, beta)
gd3.sparse<-sparse.grad.descent(huber.loss, beta)
gd3$x
```

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

```
gd3$k
```

```
## [1] 120
```

```
gd3.sparse$x
```

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

```
gd3.sparse$k
```

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

```
mse1 = mean( (gd3$x - b)^2, na.rm = TRUE)
mse2 = mean( (gd3.sparse$x - b)^2, na.rm = TRUE)
mse1
```

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

```
mse2
```

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

The sparsified coefficients seem to be closer to the true value. (except for the first coefficient) I think the regular gradient descent would generate a better MSE because it is not missing a first coefficient. The sparse method just shows that there is not that much variability to the estimates. (except when it goes out of bounds). The sparse method however does take more time.

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. Report the average mean squared error, for gradient descent, and its sparse variant, over the 10 trials. Which average lower? Also report the minimum mean squared error, for the two methods, over the 10 trials. Which is lower? Is this in line with your interpretation of the variability associated with the sparse gradient descent method?

```
for(i in 5:15){
  y <- x%*% b + rt(n, df=2)
  beta = rep(0,p)
  print(grad.descent(huber.loss, beta)$x)
  print(grad.descent(huber.loss, beta)$k)
  print(sparse.grad.descent(huber.loss, beta)$x)
  print(sparse.grad.descent(huber.loss, beta)$k)
  print(mean(grad.descent(huber.loss, beta)$x))
  print(mean(sparse.grad.descent(huber.loss, beta)$x))
  print( mean( (grad.descent(huber.loss, beta)$x - b)^2, na.rm = TRUE))
  print(mean( (sparse.grad.descent(huber.loss, beta)$x - b)^2, na.rm = TRUE))
}
```

```
##  [1] -0.92689040  0.59127732  1.10957984  0.03350239 -0.13954552
##  [6] -0.15040975  0.04657005 -0.01574992 -0.05607405  0.16780873
## [1] 113
```

```
##  [1] -0.8427806  0.5480689  1.1294024  0.0000000  0.0000000  0.0000000
##  [7]  0.0000000  0.0000000  0.0000000  0.0000000
## [1] 200
## [1] 0.06600687
## [1] 0.08346908
## [1] 0.0152247
## [1] 0.006021432
##  [1] -0.77076319  0.97311742  0.78047857  0.28721042  0.04181551
##  [6]  0.08121688 -0.11843135 -0.22684175  0.16198407  0.31710758
## [1] 120
##  [1] 0.0000000 0.7865611 0.7828980 0.0000000 0.0000000 0.0000000 0.0000000
##  [8] 0.0000000 0.0000000 0.0000000
## [1] 200
## [1] 0.1526894
## [1] 0.1569459
## [1] 0.0410904
## [1] 0.05446261
##  [1] -0.941766442  0.654059390  0.844180476  0.047090635  0.063838968
##  [6]  0.132437941  0.215312819 -0.079798776 -0.004476336  0.121920560
## [1] 105
##  [1] -0.9048100  0.6767882  0.9247907  0.0000000  0.0000000  0.0000000
##  [7]  0.0000000  0.0000000  0.0000000  0.0000000
## [1] 200
## [1] 0.1052799
## [1] 0.06967689
## [1] 0.01762861
## [1] 0.004814236
##  [1] -0.543012843  0.659887660  0.785096168  0.002645988 -0.166507295
##  [6]  0.064710144  0.399864369 -0.053070272  0.039003049  0.099711101
## [1] 92
##  [1] 0.000000 0.682856 0.755266 0.000000 0.000000 0.000000 0.000000
##  [8] 0.000000 0.000000 0.000000
## [1] 200
## [1] 0.1288328
## [1] 0.1438122
## [1] 0.02785282
## [1] 0.05501886
##  [1] -0.77551888  0.89119529  1.19070368 -0.10331148  0.02210827
##  [6] -0.24092728  0.13779745  0.16206320 -0.10082434  0.17924398
## [1] 124
##  [1] 0.0000000 0.7299036 1.0700651 0.0000000 0.0000000 0.0000000 0.0000000
##  [8] 0.0000000 0.0000000 0.0000000
## [1] 200
## [1] 0.136253
## [1] 0.1799969
## [1] 0.02353812
## [1] 0.04958033
##  [1] -0.95358542  0.59049713  0.96207366 -0.10907840  0.02333549
##  [6] -0.05269599 -0.31448879 -0.04803350 -0.11920315  0.18086776
## [1] 116
##  [1] -0.8751270  0.6106523  0.9353363  0.0000000  0.0000000  0.0000000
##  [7]  0.0000000  0.0000000  0.0000000  0.0000000
## [1] 200
## [1] 0.01596888
```

```
## [1] 0.06708617
## [1] 0.02410873
## [1] 0.004283386
##  [1] -0.4829585157  0.8331951201  1.0127205081 -0.4672266868 -0.0674999923
##  [6] -0.1929619298  0.2116135354  0.2503142573 -0.0573186217  0.0002401937
## [1] 100
##  [1]  0.0000000  0.8073050  1.0496884 -0.5400071  0.0000000  0.0000000
##  [7]  0.0000000  0.0000000  0.0000000  0.0000000
## [1] 200
## [1] 0.1040118
## [1] 0.1316986
## [1] 0.04358241
## [1] 0.0795591
##  [1] -0.65221114  0.70352349  0.98170614  0.13452635 -0.09939925
##  [6]  0.21844605  0.10931289  0.35375139  0.02128186  0.07295213
## [1] 107
##  [1] -0.6483199  0.7158742  0.9421870  0.0000000  0.0000000  0.0000000
##  [7]  0.0000000  0.0000000  0.0000000  0.0000000
## [1] 200
## [1] 0.184389
## [1] 0.1009741
## [1] 0.02211914
## [1] 0.0006265157
##  [1] -0.74539393  0.57471163  0.93753484 -0.06692913  0.15467616
##  [6] -0.01437636  0.01980830 -0.06725038  0.13899556 -0.35555413
## [1] 97
##  [1] -0.8425395  0.5135557  0.9072479  0.0000000  0.0000000  0.0000000
##  [7]  0.0000000  0.0000000  0.0000000  0.0000000
## [1] 200
## [1] 0.05762226
## [1] 0.05782641
## [1] 0.02009241
## [1] 0.006368194
##  [1] -0.75599364  0.47390681  0.92294649  0.09596641 -0.02196637
##  [6]  0.15862437  0.12304236 -0.04982223  0.09272424  0.14772185
## [1] 93
##  [1] -0.7489795  0.4991611  0.9728740  0.0000000  0.0000000  0.0000000
##  [7]  0.0000000  0.0000000  0.0000000  0.0000000
## [1] 200
## [1] 0.118715
## [1] 0.07230556
## [1] 0.01430856
## [1] 0.004347107
##  [1] -0.94899598  0.70361279  1.35669270 -0.09340251  0.18331280
##  [6] -0.16244956 -0.20950926  0.02692566  0.05574795 -0.09522889
## [1] 111
##  [1] -0.9850948  0.6527446  1.3076437  0.0000000  0.0000000  0.0000000
##  [7]  0.0000000  0.0000000  0.0000000  0.0000000
## [1] 200
## [1] 0.08167057
## [1] 0.09752935
## [1] 0.03147547
## [1] 0.01781568
```

On average the mean of the gradient descent result was lower, though, the Sparse Gradient Descent had a lower MSE. I think this does agree with the variability of the sparse gradient descent. If you intentionally limit which values you accept, you will heavily limit the outputs you can get.