STAT GU4206/GR5206 Homework 6
Practice

In this homework you'll explore various optimization algorithms for forming statistical estimates in linear regression.

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)
```

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?

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 t-distribution 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.

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.

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.

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)?

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`)?

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 `gd.sparse`. What are the final coefficient estimates?

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?

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)
```

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?