Stochastic gradient descent Exercise

Eliza Chai

04/28/2022

Stochastic gradient descent

- 1.1

Implement and test a function `lm_sgd` that performs stochastic gradient descent as described above. Describe (in words and small code sections) the main changes you have made to the `lm_gd` code, introduced in class, to implement `lm_sgd`. Put the entire code in an appendix.

I made the following changes to the `lm_gd` code in class:

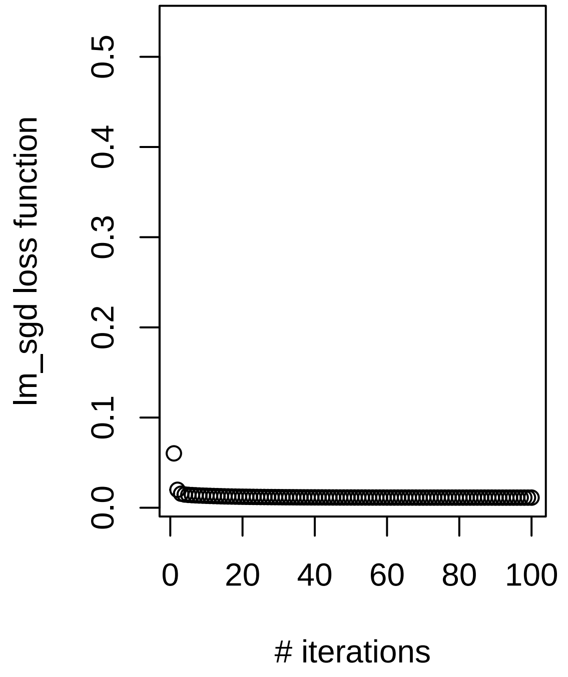
- Create a new variable `folds` which determine the number of folds/ batch numbers
- Check if numbers of rows/ observations exceed the numbers of batches `if(nrow(x) < batch) stop("Mini-batch exceed number of observations")`
- Write a for loop to compute the loss function (using the entire training set) and the gradient of the loss function w.r.t. beta (using **only** the observations in the batches)

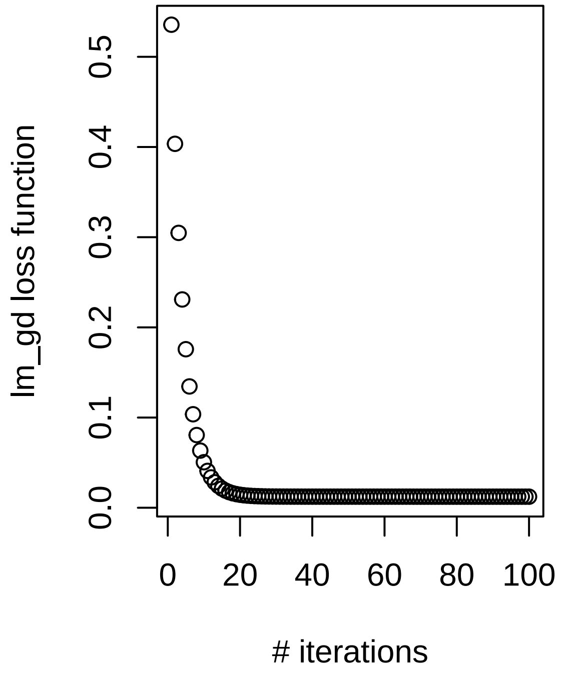
```r
## Main changes made to lm_gd code in class
for(it in 1:niter)
        {
                beta_gd = beta_gd - learn_rate*(-2/n)*(t(x)%*%(y-x%*%beta_gd))

                MSE_new = mean((y - x%*%beta_gd)^2)

                if(verbose)
                {
                        print(MSE_new)

                        x1_grid = seq(0,1,length.out = 100)
                        y_grid_hat =  cbind(rep(1,100),x1_grid)%*%beta_gd

                        plot(x[,2],y)
                        lines(x1_grid,y_grid_hat)
                        Sys.sleep(0.1)
                }
                loss_gd[it] <- MSE_new
        }
## ----------------------------------------------------------------------------
```

- 1.2

Display the values of the loss function at every iteration in a scatter plot `# iteration` vs `loss function` for both `lm_gd` and `lm_sgd`. In light of this plot, why do you think the technique is called ***stochastic*** gradient descent?

From the plots, we can see that stochastic gradient descent converge much faster than gradient descent as stochastic gradient descent reach the optimal values faster and keep oscillating there. Using gradient descent takes longer time because it run through the complete training set in every iteration to update the values of the parameters.





- 1.3

Generate a list of 20 random vectors `beta_init`. For every element in the list run stochastic gradient descent with that initialization value. Use `purrr:map` for both the generation of the random vectors and the application of `lm_sgd` (see Lecture 4). Display the 20 estimation errors $\|\beta - \beta_0\|^2$, where β_0 is the true beta used to generate the data and β is the estimated one from `lm_sgd`.

```
## [[1]]
## [1] 1.003323
##
## [[2]]
## [1] 0.9768513
##
## [[3]]
## [1] 1.016839
##
## [[4]]
## [1] 0.9728111
##
## [[5]]
## [1] 1.003116
```

```
##
## [[6]]
## [1] 1.01679
##
## [[7]]
## [1] 1.017097
##
## [[8]]
## [1] 0.9968685
##
## [[9]]
## [1] 1.020245
##
## [[10]]
## [1] 0.9863764
##
## [[11]]
## [1] 0.953922
##
## [[12]]
## [1] 0.9903366
##
## [[13]]
## [1] 0.9915558
##
## [[14]]
## [1] 0.9814975
##
## [[15]]
## [1] 0.9586105
##
## [[16]]
## [1] 0.9794109
##
## [[17]]
## [1] 1.022046
##
## [[18]]
## [1] 1.009331
##
## [[19]]
## [1] 0.9866287
##
## [[20]]
## [1] 1.019564
```

Code Appendix

```
## Setting up the packages, options we'll need:
library(knitr)
knitr::opts_chunk$set(echo = TRUE)
## -----------------------------------------------------------------------------
## Implementation of stochastic gradient descent

## install.packages("caret")
library(caret)
rm(list=ls())
lm_sgd = function(x, y, beta_init = NULL, learn_rate = 0.05,
                  niter = 100, folds=10, verbose = F)
{

        n = nrow(x)
        p = ncol(x)-1

        if(nrow(x) != length(y)) stop("Check the dimensions of x and y")
        if(nrow(x) < folds) stop("folds exceed number of observations")
        if(verbose && p>2) stop("p > 2, -- Plotting not implemented")

        if(is.null(beta_init)) beta_init = runif(p+1)

        beta_sgd = beta_init

        f <- createFolds(y, k = folds, list = T, returnTrain = F)
        loss_sgd <- rep(0, niter)

        for(it in 1:niter)
        {
                for(batch in 1:folds){
                        x_b <- x[f[[batch]], ]
                        y_b <- y[f[[batch]]]

                        beta_sgd = beta_sgd - learn_rate*
                          (-2/length(y_b))*(t(x_b)%*%(y_b - x_b%*%beta_sgd))

                        MSE_new = mean((y_b - x_b%*%beta_sgd)^2)

                        if(verbose)
                        {
                                print(MSE_new)

                                x1_grid = seq(0,1,length.out = 100)
                                y_grid_hat = cbind(rep(1,100),x1_grid)%*%beta_gd

                                plot(x[,2],y)
                                lines(x1_grid,y_grid_hat)
                                Sys.sleep(0.1)
                        }
                }
                loss_sgd[it] <- MSE_new
```

```r
        }
        return(list(beta_sgd, loss_sgd))
}

## Test
n = 30
p = 1

beta = rep(1,p+1)

x = cbind(rep(1,n),matrix(runif(n*p,0,1),n,p))
epsilon = rnorm(n,0,.1)

y = x%*%beta + epsilon

lm_sgd(x, y)

## ----------------------------------------------------------------------------

## Main changes made to lm_gd code in class
for(it in 1:niter)
        {
                beta_gd = beta_gd - learn_rate*(-2/n)*(t(x)%*%(y-x%*%beta_gd))

                MSE_new = mean((y - x%*%beta_gd)^2)

                if(verbose)
                {
                        print(MSE_new)

                        x1_grid = seq(0,1,length.out = 100)
                        y_grid_hat =  cbind(rep(1,100),x1_grid)%*%beta_gd

                        plot(x[,2],y)
                        lines(x1_grid,y_grid_hat)
                        Sys.sleep(0.1)
                }
                loss_gd[it] <- MSE_new
        }
## ----------------------------------------------------------------------------

## Implementation: linear regression
lm_gd = function(x, y, beta_init = NULL, learn_rate = 0.05,
                 niter = 100, verbose = F)
{

        n = nrow(x)
        p = ncol(x)-1

        if(nrow(x) != length(y)) stop("Check the dimensions of x and y")
        if(verbose && p>2) stop("p > 2, -- Plotting not implemented")

        if(is.null(beta_init)) beta_init = runif(p+1)
        beta_gd = beta_init

        loss_gd <- rep(0, niter)

        for(it in 1:niter)
        {
                beta_gd = beta_gd - learn_rate*(-2/n)*(t(x)%*%(y-x%*%beta_gd))

                MSE_new = mean((y - x%*%beta_gd)^2)

                if(verbose)
                {
                        print(MSE_new)

                        x1_grid = seq(0,1,length.out = 100)
                        y_grid_hat =  cbind(rep(1,100),x1_grid)%*%beta_gd

                        plot(x[,2],y)
                        lines(x1_grid,y_grid_hat)
                        Sys.sleep(0.1)
                }
                loss_gd[it] <- MSE_new
        }
        return(list(beta_gd, loss_gd))
}


## Test
n = 30
p = 1

beta = rep(1,p+1)

x = cbind(rep(1,n),matrix(runif(n*p,0,1),n,p))
epsilon = rnorm(n,0,.1)

y = x%*%beta + epsilon
## ---------------------------------------------------------------------------

## Store value of loss function
gd_loss <- lm_gd(x, y)[[2]]
sgd_loss <- lm_sgd(x, y)[[2]]

## Plot scatterplots of # iteration vs loss function for both gradient descent and stochastic gradient
par(mfrow=c(1,2))
yrange <- range(gd_loss, sgd_loss)
plot(x=1:100, y=sgd_loss, ylim = yrange,
     xlab = "# iterations", ylab = "lm_sgd loss function")
plot(x=1:100, y=gd_loss, ylim = yrange,
     xlab = "# iterations", ylab = "lm_gd loss function")

library(purrr)
beta_init <- map(1:20, ~ sample(1:100, size=2))
```

```
map(beta_init, ~ lm_sgd(x,y)[[1]][[2]])
```