Solution 1:

(a) Write a function in R implementing a gradient descent routine for the optimization of the linear model defined in the previous exercise sheet. Start with:

```
#' @param step_size the step_size in each iteration
#' @param X the feature input matrix X
#' @param y the outcome vector y
#' @param beta a starting value for the coefficients
#' @param eps a small constant measuring the changes in each update step.
#' Stop the algorithm if the estimated model parameters do not change
#' more than \code{eps}.

#' @return a set of optimal coefficients beta
gradient_descent <- function(step_size, X, y, beta = rep(0,ncol(X)),
                             eps = 1e-8){

  change <- 1 # something larger eps

  XtX <- crossprod(X)
  Xty <- crossprod(X,y)

  while(sum(abs(change)) > eps){

    # Use standard gradient descent:
    change <- + step_size * (Xty - XtX%*%beta)

    # update beta in the end
    beta <- beta + change


  }

  return(beta)

}
```

(b) Run a small simulation study by creating 100 data sets as indicated below and test different step sizes α (fixed across iterations) against each other and against the state-of-the-art routine for linear models in R using the function `lm`.

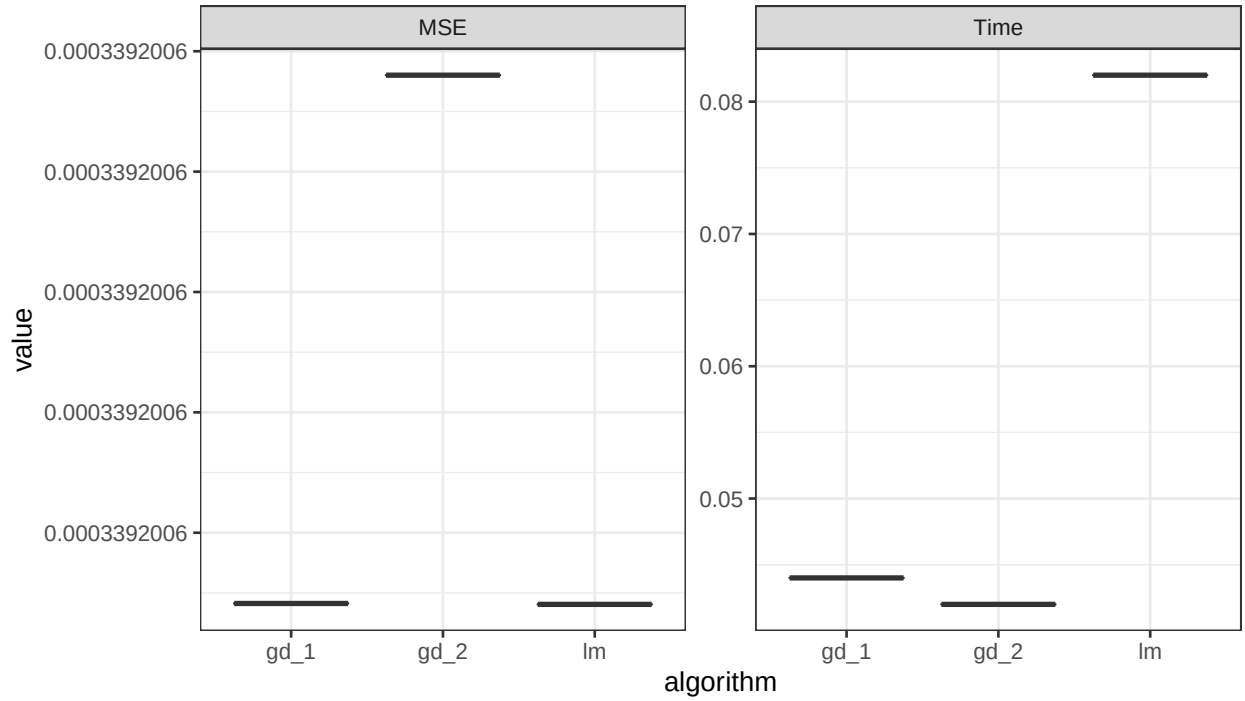
- Compare the difference in estimated coefficients $\beta_j, j = 1, \ldots, p$ using the mean squared error, i.e.

$$p^{-1} \sum_{j=1}^{p} (\beta_j^{truth} - \hat{\beta}_j)^2$$

  and summarize the difference over all 100 simulation repetitions.
- Compare the run times of your implementation and the one given by `lm` by wrapping the function calls into `system.time()`.

```r
n <- 10000
p <- 100
nr_sims <- 20

# define mse
mse <- function(x,y) mean((x-y)^2)

# create data (only once)
X <- matrix(rnorm(n*p), ncol=p)
beta_truth <- runif(p, -2, 2)
f_truth <- X%*%beta_truth

# create result object
result_list <- vector("list", nr_sims)

# make it all reproducible
set.seed(2020-4-6)

for(sim_nr in nr_sims)
{

  # create response
  y <- f_truth + rnorm(n, sd = 2)

  time_lm <- system.time(
    coef_lm <- coef(lm(y~-1+X))
  )["elapsed"]

  time_gd_1 <- system.time(
    coef_gd_1 <- gradient_descent(step_size = 0.0001, X = X, y = y)
  )["elapsed"]

  time_gd_2 <- system.time(
    coef_gd_2 <- gradient_descent(step_size = 0.00001, X = X, y = y)
  )["elapsed"]


  mse_lm <- mse(coef_lm, beta_truth)
  mse_gd_1 <- mse(coef_gd_1, beta_truth)
  mse_gd_2 <- mse(coef_gd_2, beta_truth)

  # save results in list (performance, time)
  result_list[[sim_nr]] <- data.frame(mse_lm = mse_lm,
                                      mse_gd_1 = mse_gd_1,
                                      mse_gd_2 = mse_gd_2,
                                      time_lm = time_lm,
                                      time_gd_1 = time_gd_1,
                                      time_gd_2 = time_gd_2)

}

library(ggplot2)
library(dplyr)
library(tidyr)

do.call("rbind", result_list) %>%
  gather() %>%
  mutate(what = ifelse(grepl("mse", key), "MSE", "Time"),
```

```
        algorithm = gsub("(mse|time)\\_(.*)","\\2", key)) %>%
ggplot(aes(x = algorithm, y = value)) +
geom_boxplot() + theme_bw() +
facet_wrap(~ what, scales = "free")
```



(c) There exists an analytic solution to this problem, namely $\hat{\boldsymbol{\beta}} = (\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{y}$. Gradient descent might sometimes be slower and less exact. However, for very large data sets, a numerical optimization might be the preferred solution (in this case, you would rather apply **stochastic** gradient descent). Analytically solving the problem involves inverting the matrix $\boldsymbol{X}^\top \boldsymbol{X}$, which should never be done explicitly, but rather by solving the linear equation $\boldsymbol{X}^\top \boldsymbol{y} = \boldsymbol{X}^\top \boldsymbol{X} \boldsymbol{\beta}$ or by decomposing $\boldsymbol{X}^\top \boldsymbol{X}$ first, e.g., using Cholesky or QR decomposition.

(d) Our learning algorithm $\mathcal{I}$ will always have an approximation error if $f^* \notin \mathcal{H}$, and is thus not consistent.