Assignment 5

In line with Lecture 7, Assignment 5 is somewhat more technical. This means there are no basic warm-up tasks since even the use of a canned routine for optimization requires us to write a function for our cost function.

Still, this assignment covers frequently occuring coding tasks in the context of function optimization. That is, you will write many functions, transfer a lot of mathematical expressions from the lecture slides into this script, create some simple plots and, most importantly, spend far too much time on trying to figure out what is wrong with your code. On the upside, this assignment gives you a little more freedom to write code in your own way. Enjoy!

Preamble: Preparing the data

We are going to recycle a dataset on purchases in online shops that we already used in Assignment 2. Load the data into R and save it in an object called `shoppers`. The dataset is contained in a comma-separated spreadsheet. Accordingly, you will need to use the ´ read.csv()´ command in R.

```
setwd("C:/Users/claes/OneDrive/Universitet/DataScience Magisterprogram/STAN51 - Maskininlärning ur ett
shoppers <- read.csv("online_shoppers_intention.csv")
```

In the steps below, we will fit a quite small logistic regression model including the following variables in addition to the intercept:

1. `ExitRate` without further transformation
2. The (natural) logarithm of `ProductRelated_Duration + 1`

The code below does all data transformations for you. However, please note the shortened variable names that we are going to use from now on.

```
X <- model.matrix(Revenue ~ ExitRates + log(1 + shoppers$ProductRelated_Duration), data=shoppers)
X <- as.matrix(X)
colnames(X) <- c("intercept", "ER", "lPR_Dur")
y           <- rep(1, times=dim(X)[1])
y[shoppers$Revenue==FALSE] <- -1
```

Part 1: Learning logistic regression without `glm()`

Task 1a)

If we want to learn a logistic regression manually, we need to specify the cost function that we are going to minimize So as a first step, create a function `cost_logistic_1a()` whose inputs are

1. coefficients `theta`,
2. an n vector of outputs `y`,

3. an $n \times p$ matrix of inputs `X`.

The function is then supposed to return the cost on the data consisting of `y` and `X` with logistic loss and coefficient vector values `theta`.

```
cost_logistic_1a <- function(theta, y, X){
  n <- length(y)
  h <- (1+exp(-(y*X%*%theta)))
  cost <- log(h)
  c <- mean(cost)
  return(c)
}

cost_logistic_1a(c(0,0,0),y,X)
```

```
## [1] 0.6931472
```

Task 1b)

As a next step, we need a function that returns the gradient of the cost function. Write such a function `grad_logistic_1b()` that takes the same arguments as `cost_logistic()` and which returns a $p \times 1$ vector containing the gradient of our cost function. Use the `matrix()` command to ensure that the dimensions of the returned vector are correct.

```
grad_logistic_1b <- function(theta, y, X) {
  n <- length(y)
  h <- (1 + exp(y*X %*% theta))
  gradient <- (1 / n) * -t(X*y) %*% (h)^(-1)
  return(matrix(gradient))
}

grad_logistic_1b(c(0,0,0),y,X)
```

```
##             [,1]
## [1,] 0.34525547
## [2,] 0.01851034
## [3,] 1.91077480
```

Task 1c)

R provides you with a canned routine for optimization: The `optim()` command. `optim()` minimizes a function that you feed into it. It also allows you to provide a gradient function to speed up optimization and to achieve better numerical stability in complicated cases.

`optim()` allows you to choose from several algorithms from the very broad literature on optimization methods. We are going to use the `BFGS` method which is a very popular modification of Newton's method.

Now use `optim()` to minimize `cost_logistic_1a` on your training data. Follow the instructions below:

1. Let all initial coefficient values be zero.
2. Choose the BFGS minimization method.

3. Supply your gradient function `grad_logistic_1b` as well. If you could not solve Task 1b, leave out the gradient.
4. Specify input `y` and outputs `X` as additional arguments of `optim()`. You *must* assign them the name that they have inside your functions `cost_logistic_1a()` and `grad_logistic_1b`.
5. Feed the optimization controls `optim_ctrl_1c` that I already specified in the code chunk below to `optim()`

Save the resulting list object as `optim_result_1c`.

```
optim_ctrl_1c     <- list(maxit=10000, reltol=1e-16)
initial_theta <- c(0,0,0)

optim_result_1c <- optim(par = initial_theta,
                         fn = cost_logistic_1a,
                         gr = grad_logistic_1b,
                         y = y,
                         X = X,
                         method = "BFGS",
                         control = optim_ctrl_1c)


optim_result_1c
```

```
## $par
## [1]  -2.4893245 -30.2814317   0.2474496
##
## $value
## [1] 0.3868898
##
## $counts
## function gradient
##     5408     4729
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Task 1d)

Have a closer look at the list object `optim_result_1c`. What do its different components tell us about our solved minimization problem? Explain this in your *own* words by writing explanations into the four string variables below.

```
whatis_par_1d <- "The best set of parameters found"
whatis_value_1d <- "The value of fn corresponding"
whatis_counts_1d <- "   A two-element integer vector giving the number of calls to fn and gr respectivel
whatis_convergence_1d <- "An indicator of convergence: 0 indicates successful convergence."
```

Part 2: Gradient descent manually

In this part, you are going to write your own routine for gradient descent.

Task 2a)

Below, I have prepared a fragment of a function that is supposed to conduct gradient descent. This function, `grad_desc_2a`, takes the following inputs:

- `par`: The vector of initial coefficient values
- `fn`: The objective function
- `gr`: The gradient of the objective function
- `stepsize`: The step size
- `maxitr`: The maximum number of parameter updates. Default is set to 5000
- `tol`: The tolerance for coefficient updates to be considered "effectively 0". Default is set to 0.000001
- `y`: an n vector of outputs
- `X`: an $n \times p$ matrix of inputs.

Additionally, the function already contains the following objects:

- `coef_path`: A matrix whose columns eventually contain the entire sequence of coefficient vectors,
- `coef_upd`: A matrix whose columns eventually contain the entire sequence of coefficient updates,
- `fn_path`: A vector that eventually contains all values of the objective function.

Your task is to write code that conducts the coefficient updates of gradient descent until the Euclidean norm of a coefficient update is below the tolerance level `tol`.

The full sequence of coefficient vectors, coefficient updates and function values should be saved in the objects `coef_path`, `coef_upd` and `fn_path`, respectively. Maximally `maxitr` coefficient updates should be made (fewer if coefficient updates are below tolerance).

```
grad_desc_2a <- function(par, fn, gr, stepsize, maxitr=5000, tol=1e-6, y, X) {

  # Don't change anything here
  coef_path <- matrix(par, nrow=length(par), ncol=1)
  coef_upd  <- matrix(NA, nrow=length(par), ncol=0)
  fn_path   <- fn(theta=par, X=X, y=y)

  itr <- 0

  # Start writing stuff here
  while (itr < maxitr) {
    gradient <- gr(theta=par, X=X, y=y)
    update <- stepsize * gradient
    par <- par - update

    # Save the current values
    coef_path <- cbind(coef_path, par)
    coef_upd <- cbind(coef_upd, update)
    fn_val <- fn(theta=par, X=X, y=y)
    fn_path <- c(fn_path, fn_val)

    # Check for convergence
    if (sqrt(sum(update^2)) < tol) {
      break
    }
```

```
    itr <- itr + 1
  }

  # Don't change anything here
  returnobj <- list(coef_final = coef_path[,ncol(coef_path)],
                    itr        = itr,
                    coef_path  = coef_path,
                    updates    = coef_upd,
                    fun_path   = fn_path)
  return(returnobj)
}

# grad_desc_2a(initial_theta, cost_logistic_1a, grad_logistic_1b, 0.001, maxitr=5000, tol=1e-6, y=y, X=
```

Task 2b)

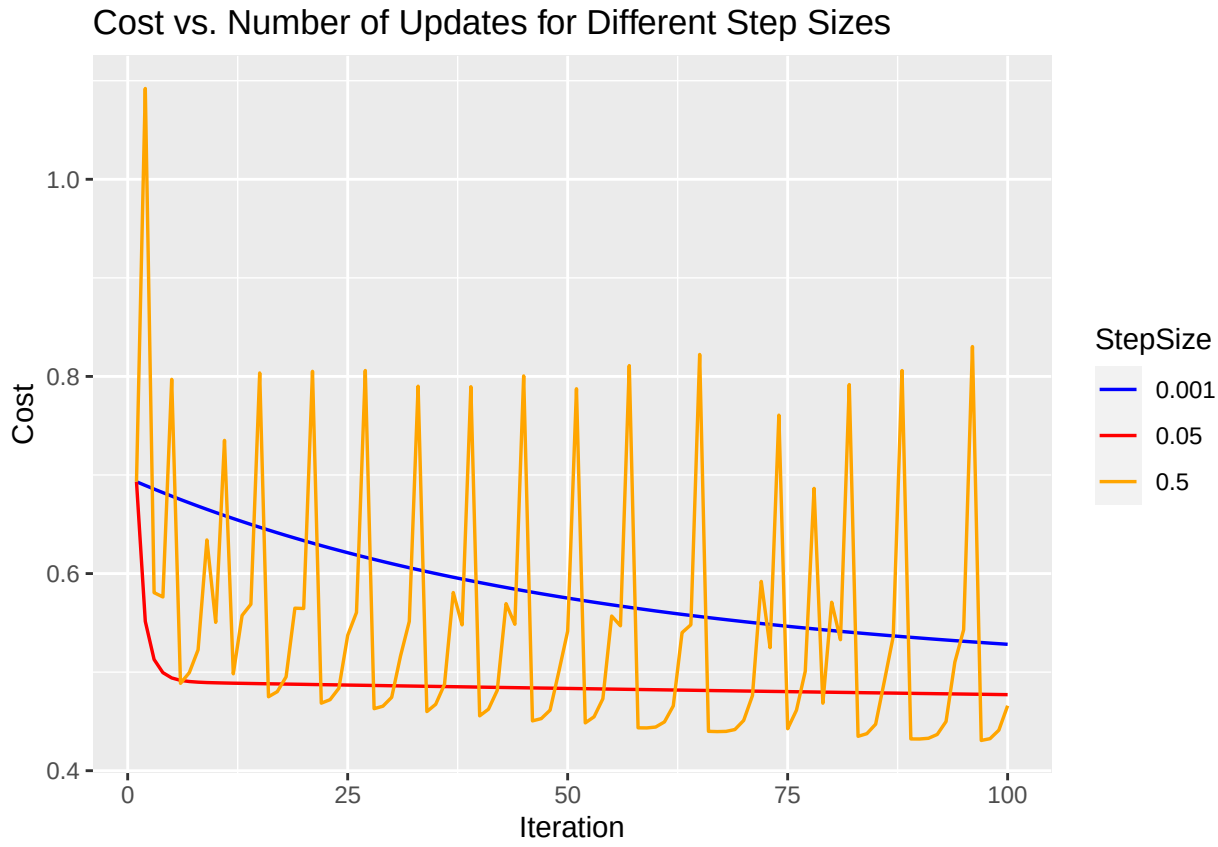
Now we tune the step size of gradient descent. In order to do this, write a function `plot_cost_2b` which runs gradient descent with a chosen step size and only 100 coefficient updates. The function should then create an object containing a line plot of the number of coefficient updates against the cost at the specific update. This line plot object is to be returned by the function.

I have already prepared a list object `gd_ctrl_2b` for you that contains the gradient descent function as well as most inputs to this function. Use `gd_ctrl_2b` as an input and define further arguments to your step size as well as the data.

Once you have created `plot_cost_2b`, play around with different step sizes. Save a plot with a step size that you deem as too low (high) as object `plot_tooLO_2b` (`plot_tooHI_2b`). Save another plot `plot_decent_2b` which contains the development of cost with a good step size.

```
library(ggplot2)

gd_ctrl_2b <- list(par = c(0, 0, 0),
                   fn = cost_logistic_1a,
                   gr = grad_logistic_1b,
                   gd_fun = grad_desc_2a,
                   maxitr = 100)

stepsize_low_decent_high <- c(0.001, 0.05, 0.5)

plot_cost_2b <- function(gd_ctrl, stepsize_low_decent_high, X, y) {
  combined_df <- data.frame()  # Create an empty data frame to store cost values

  for (stepsize in stepsize_low_decent_high) {
    gd_ctrl$par <- c(0, 0, 0)  # Initial coefficients
    gd_ctrl$stepsize <- stepsize  # Set the step size
    gd_result <- gd_ctrl$gd_fun(par = gd_ctrl$par,
                                fn = gd_ctrl$fn,
                                gr = gd_ctrl$gr,
                                stepsize = gd_ctrl$stepsize,
                                maxitr = 100,
                                y = y,
                                X = X)

    # Create a data frame for the current step size
```

```
    df_stepsize <- data.frame(itr = 1:gd_result$itr, cost = gd_result$fun_path[1:gd_result$itr], StepSi

    combined_df <- rbind(combined_df, df_stepsize)
  }

  # Convert StepSize to a factor
  combined_df$StepSize <- factor(combined_df$StepSize)

  # Create the plot using ggplot2 with conditional lines
  ggplot(combined_df, aes(x = itr, y = cost, color = StepSize)) +
    geom_line(lwd = 0.6) +
    labs(title = "Cost vs. Number of Updates for Different Step Sizes", x = "Iteration", y = "Cost") +
    scale_color_manual(values = c("0.001" = "blue", "0.05" = "red", "0.5" = "orange"))
}

stepsize_low<-0.001
stepsize_decent<-0.05
stepsize_high <- 0.5



# Call the function with the step sizes
plot_cost_2b(gd_ctrl_2b, stepsize_low_decent_high, X, y)
```



Task 2c)

Use the string variable `stepsize_motivate_2c` to motivate why you chose the three step size examples in Task 2b. More specifically, explain what lead you to the conclusion that a step size is too low or too high.

```
stepsize_motivate_2c <- "If we look at the plot we can see it will take vary many iterations for the sma
```

Task 2d)

Use `grad_desc_2a` with your chosen step size to learn the coefficients of the logistic regression model that we have been working with so far. Keep the maximum number of updates at its default value. Save the resulting obect as `gd_result_2d`.

Does gradient descent stop before the maximum number of updates? If not, how far does gradient descent get the cost function towards the value achieved by `optim()` in Task 1c? Write your answer into the string variable `gd_conclusion_2d`

```
gd_result_2d <- grad_desc_2a(par=gd_ctrl_2b$par,
                             fn=gd_ctrl_2b$fn,
                             gr=gd_ctrl_2b$gr,
                             stepsize=stepsize_decent,
                             y=y,
                             X=X)

stopped_early <- gd_result_2d$itr < 5000
stopped_early # FALSE
```

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

```
cost_difference <- abs(gd_result_2d$fun_path[gd_result_2d$itr] - optim_result_1c$value)

gd_conclusion_2d <- "Gradient descent with a learning rate/stepsize of 0.05 reached the maxitr limit. 0

The difference is instead ~0.018293. More iterations are needed in order to find the optimum with the sp
```

Part 3: Newton's method manually

Given that you have implemented gradient descent in Part 2, you have already done most of the hard work that is required for Newton's method. After all, the only thing that differs is the expression for your coefficient update. For this reason, we will work on our own routine for Newton's method.

Task 3a)

Newton's method is more demanding than gradient descent in that it requires us to obtain the Hessian of our objective function for any desired vector of coefficients. Write such as function `hessian_logistic_3a` which has the same arguments as `grad_logistic_1b` and which returns the hessian of your cost function with logistic loss.

```r
hessian_logistic_3a <- function(theta, X, y) {
  expo <- exp(y*X%*%theta)
  ratio <- (expo/(1+expo)^2)
  XD <- X*y^2*as.vector(ratio)
  L_sum <- t(X)%*%XD
  babl_c <- L_sum/length(y)
  return(babl_c)
}


hessian_logistic_3a(initial_theta, X, y)
```

```
##           intercept          ER    lPR_Dur
## intercept 0.2500000 0.010768199 1.49063228
## ER        0.0107682 0.001054175 0.04491585
## lPR_Dur   1.4906323 0.044915854 9.93140902
```

Task 3b)

Modify `grad_desc_2a` to turn it into a function for Newton's method. This only requires you to remove step size from the function arguments, to add a new argument for the Hessian and to change the line for your coefficient update. Save your function as object `newton_3b`.

```r
newton_3b <- function(par, fn, gr, hessian, maxitr=5000, tol=1e-6, y, X) {

  # Don't change anything here
  coef_path <- matrix(par, nrow=length(par), ncol=1)
  coef_upd  <- matrix(NA, nrow=length(par), ncol=0)
  fn_path   <- fn(theta=par, X=X, y=y)

  itr <- 0

  # Start writing stuff here
  while (itr < maxitr) {
    gradient <- gr(theta=par, X=X, y=y)
    hes <- hessian(theta=par, X=X, y=y)
    update <- hes %*% gradient
    par <- par - update

    # Save the current values
    coef_path <- cbind(coef_path, par)
    coef_upd <- cbind(coef_upd, update)
    fn_val <- fn(theta=par, X=X, y=y)
    fn_path <- c(fn_path, fn_val)

    # Check for convergence
    if (sqrt(sum(update^2)) < tol) {
      break
    }

    itr <- itr + 1
  }
```

```
  # Don't change anything here
  returnobj <- list(coef_final = coef_path[,ncol(coef_path)],
                    itr        = itr,
                    coef_path  = coef_path,
                    updates    = coef_upd,
                    fun_path   = fn_path)
  return(returnobj)
}

#newton_3b(initial_theta, cost_logistic_1a, grad_logistic_1b, hessian_logistic_3a, maxitr=5000, tol = 1
```

Task 3c)

Use your function `newton_3b` to learn the logistic regression model that you already learned in Tasks 1c and 2d. Keep the maximum number of updates at its default value. Save the resulting obect as `nm_result_3c`.

Does Newton's method stop before the maximum number of updates? If not, how far does gradient descent get the cost function towards the value achieved by `optim()` in Task 1c? What is your verdict about the performance of Newton's method relative to gradient descent and the `optim()` command in the case of logistic regression? Write your answer into the string variable `nm_conclusion_3c`.

```
nm_result_3c <- newton_3b(par = initial_theta, fn = cost_logistic_1a,
                          gr = grad_logistic_1b, hessian = hessian_logistic_3a,
                          maxitr = 5000, tol = 1e-6, y = y, X = X)

# Check if Newton's method stopped before maximum updates
stopped_early_newton <- nm_result_3c$itr < 5000

# Compare the cost achieved by Newton's method with the one from optim()
cost_from_newton <- nm_result_3c$fun_path[nm_result_3c$itr]

# Verdict on performance
nm_conclusion_3c <- ifelse(stopped_early_newton,
                          "Newton's method stopped before max updates. Cost from Newton is in the same
                          "Newton's method reached max updates. Cost from Newton is in the same range a

# Print the results
# print(nm_result_3c)
print(stopped_early_newton)
```

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

```
print(cost_from_newton)
```

```
## [1] 20.67505
```

```
print(optim_result_1c)  # Cost from optim() (previously obtained in Task 1c)
```

```
## $par
## [1]  -2.4893245 -30.2814317   0.2474496
```

```
##
## $value
## [1] 0.3868898
##
## $counts
## function gradient
##     5408     4729
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```
print(nm_conclusion_3c)
```

```
## [1] "Newton's method reached max updates. Cost from Newton is in the same range as Optim."
```