Computational Statistics Lab 2

Simge Cinar & Ronald Yamashita

2023-11-12

QUESTION 1:

Part a)

Question: *Derive the gradient and the Hessian matrix in dependence of x,y. Produce a contour plot of the function g.*
The function g(x,y) can be seen below. The aim is to determine the point $(x, y) \in [-3, 3]$ which maximizes the function g.

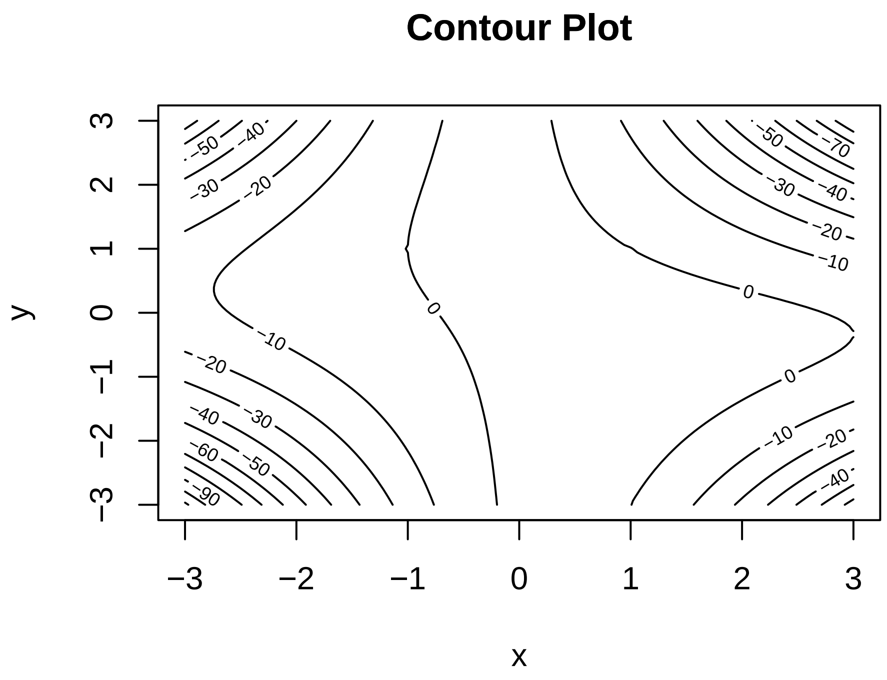
$$g(x,y) = -x^2 - x^2y^2 - 2xy + 2x + 2$$

Its gradient matrix and hessian matrix are as follows:

$$g'(x,y) = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} = \begin{bmatrix} -2x - 2xy^2 - 2y + 2 \\ -2x^2y - 2x \end{bmatrix}$$

$$g''(x,y) = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix} = \begin{bmatrix} -2 - 2y^2 & -4xy - 2 \\ -4xy - 2 & -2x^2 \end{bmatrix}$$

Contour plot the function g can be seen below.



Part b)

Question: *Write an own algorithm based on the Newton method in order to find a local maximum of g.* Our algorithm can be seen below. Convergence criteria is chosen as $\|\mathbf{x}^{(\mathbf{t})} - \mathbf{x}^{(\mathbf{t+1})}\| \leq \epsilon$ and maximum number of iterations is defined to prevent infinite loops. In all of the experiments, we select $\epsilon = 10^{-10}$ and maximum iteration as 10,000

```r
newton_optimization <- function(x, y, eps, max_iter){
  xt <- matrix(c(x,y), ncol= 1)
  iter <- 0
  while (iter < max_iter){
    hessian_mat <- hessian_g(xt[1,1],xt[2,1])
    gradient_mat <- gradient_g(xt[1,1],xt[2,1])
    xt1 <- xt - solve(hessian_mat) %*% gradient_mat
    # stopping condition
    if (dist(xt - xt1) <= eps){
      break
    }
    xt <- xt1
    iter <- iter+1
  }
  newton_list = list(value = g(xt1[1], xt1[2]),
                     param = c(x = xt1[1], y = xt1[2]),
                     iteration = iter)
  return(newton_list)
}
```

Part c)

Question: *Use different starting values: use the three points (x, y) = (2, 0), (−1, −2), (0, 1) and a fourth point of your choice. Describe what happens when you run your algorithm for each of those starting values. If your algorithm converges to points (x, y), compute the gradient and the Hessian matrix at these points and decide about local maximum, minimum, saddle point, or neither of it. Did you find a global maximum for $(x, y) \in [-3, 3]$?*

Newton's optimization method with four different starting values is applied to find the optimal points. The table with the corresponding values are as follows:

Initial Point	Optimal Value	Optimal Point	Iteration	Eigenvalues	Min/Max/Saddle
(2,0)	4	(1,-1)	7	(-0.763932, -5.236068)	Max
(-1,-2)	2	(0,1)	16	(0.8284271, -4.828427)	Saddle
(0,1)	2	(0,1)	0	(0.8284271, -4.828427)	Saddle
(0,0)	2	(0,1)	1	(0.8284271, -4.828427)	Saddle

Table 1: Newton's Optimization Table

$$g'(1,-1) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, g''(1,-1) = \begin{bmatrix} -4 & 2 \\ 2 & -2 \end{bmatrix}$$

$$g'(0,1) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, g''(0,1) = \begin{bmatrix} -4 & -2 \\ -2 & 0 \end{bmatrix}$$

The corresponding gradient and hessian matrix with different converged points can be seen above. Point (1,-1) has a value of 4 and the eigenvalues of its hessian matrix is both negative. This shows that the point (1,-1) is local maximum. Point (0,1) gives the value 2. It is a saddle point since the eigenvalues of the hessian matrix has different signs. It should be noted that Newton's optimization method is dependent on the initial point and does not guarantee optimality. Only initial point (2,0) converged to a local maximum point. Other initial points which are $(-1, -2)$, (0, 1), (0,0) converged to a saddle point.

To find the global maximum a sequence is created in the interval [-3,3] for both x and y by increasing the value 0.01. The function g is calculated for each combination and the maximum value is found as 4 and at the point (1,-1). Moreover optim() function is used and it gives the same result. The corresponding code can be seen in the Appendix.

Part d)

Question: *What would be the advantages and disadvantages when you would run a steepest ascent algorithm instead of the Newton algorithm?*
Newton's method has quadratic convergence while gradient ascent has linear convergence therefore Newton's method reaches the optimal solution in fewer iterations compared to gradient-based methods. Also gradient descent doesn't give information about curvature of the surface.

On the other hand, cost of computing Newton's method is higher because we need to calculate the Hessian matrix. Consider the case we know both hessian matrix and gradient matrix of the function. We still need to calculate the inverse of the Hessian matrix for Newton's method which has computational complexity $O(n^3)$ whereas gradient ascent has $O(n)$ computational complexity. Lastly, Newton does not guarantee that g increases in each step.

QUESTION 2:

part a)

Question: *Write a function for an ML-estimator for (β_0, β_1) using the steepest ascent method with a step-size reducing line search (back-tracking). For this, you can use and modify the code for the steepest ascent example from the lecture. The function should count the number of function and gradient evaluations.*

The aim of this question is find MLE for β_0 and β_1 using steepest ascent. The formula for steepest ascent is as follows:

$$x^{(t+1)} = x^{(t)} + \alpha^{(t)} I g'(x^{(t)})$$

The ML-estimator function is as follows:

```
# The gradient ascent function
gradient_ascent <- function(alpha0, beta_init, eps, max_iter){
  iter <- 0
  alpha <- alpha0
  beta_t <- beta_init
  beta_t1 <- beta_t + alpha * gradient(X, y, beta_t)
  g_t <- log_logistic_reg(X, y, beta_t)
  gradient_t <- gradient(X, y, beta_t)

  loglikelihoods <- c(g_t)
  eval_count <- 1  # Counter for function evaluations
  grad_count <- 1  # Counter for gradient evaluations

  while (iter < max_iter) {
    g_t1 <- log_logistic_reg(X, y, beta_t1)
    eval_count <- eval_count + 1  # Increment function evaluation counter

    # find the stepsize that increases the loglikelihood
    while (g_t1 < g_t){
      alpha <- alpha/2
      beta_t1 <- beta_t + alpha * gradient_t
      g_t1 <- log_logistic_reg(X, y, beta_t1)
      eval_count <- eval_count + 1  # Increment function evaluation counter
      cat("Learning rate changed!\n")
    }

    # assign beta_t1 and g_t1 as previous values
    beta_t <- beta_t1
    g_t <- g_t1

    # calculate new beta
    gradient_t <- gradient(X, y, beta_t)
    beta_t1 <- beta_t + alpha * gradient_t
    grad_count <- grad_count + 1  # Increment gradient evaluation counter

    # save values
    loglikelihoods <- c(loglikelihoods, g_t1)
    iter <- iter + 1

    # stopping condition
    if (abs(beta_t[1] - beta_t1[1]) <= eps && abs(beta_t[2] - beta_t1[2]) <= eps){
      break
    }
  } #end the loop

  result_list <- list(par = t(beta_t1),
                      value = loglikelihoods[length(loglikelihoods)],
                      counts = c('function' = eval_count, 'gradient' = grad_count),
                      iteration = iter)
  return(result_list)
}
```

part b)

Question: *Compute the ML-estimator with the function from a for the data (x_i, y_i) above. Use a stopping criterion such that you can trust five digits of both parameter estimates for β_0 and β_1. Use the starting value $(\beta_0, \beta_1) = (-0.2, 1)$. The exact way to use backtracking can be varied. Try two variants and compare number of function and gradient evaluation done until convergence.*

Initial value of the $\alpha_0 = 1$ is selected and backtracking is used. α value changed 1 time. The value found is -6.484279 and parameters are $(\beta_0, \beta_1) = (-0.009339359, 1.262764)$. Function count is 58 and gradient count is 57.

```
## Learning rate changed!
```

```
## Alpha: 1
```

```
## $par
##              [,1]     [,2]
## [1,] -0.009339359 1.262764
##
## $value
## [1] -6.484279
```

```
##
## $counts
## function gradient
##       58       57
##
## $iteration
## [1] 56
```

Other initial α_0 and beta values used as well. Let us start with $\alpha_0 = 10$. Stepsize (α) changed 4 times. The value found is -6.484279 and parameters are $(\beta_0, \beta_1) = (-0.00934389, 1.262777)$. Function count is 50 and gradient count is 46.

```
## Learning rate changed!
## Learning rate changed!
## Learning rate changed!
## Learning rate changed!
```

```
## Alpha: 10
```

```
## $par
##             [,1]     [,2]
## [1,] -0.00934389 1.262777
##
## $value
## [1] -6.484279
##
## $counts
## function gradient
##       50       46
##
## $iteration
## [1] 45
```

Lastly set $\alpha_0 = 3$ and also change the initial beta values to $(\beta_0, \beta_1) = (5, -5)$. The value found is -6.484279 and parameters are $(\beta_0, \beta_1) = (-0.009355319, 1.262825)$. Function count is 100 and gradient count is 98. Alpha value changed 2 times.

```
## Learning rate changed!
## Learning rate changed!
```

```
## Alpha: 3
```

```
## $par
##              [,1]     [,2]
## [1,] -0.009355319 1.262825
##
## $value
## [1] -6.484279
##
## $counts
## function gradient
##      100       98
##
## $iteration
## [1] 97
```

alpha	initial (β_0, β_1)	optimal (β_0, β_1)	value	function count	gradient count
$\alpha_0 = 1$	(-0.2, 1)	(-0.009339359, 1.262764)	-6.484279	58	57
$\alpha_0 = 10$	(-0.2, 1)	(-0.00934389, 1.262777)	-6.484279	50	46
$\alpha_0 = 3$	(5, -5)	(-0.009355319, 1.262825)	-6.484279	100	98

Table 2: Steepest Ascent Comparison

part c)

Question: *Use now the function optim with both the BFGS and the Nelder-Mead algorithm. Do you obtain the same results compared with b.? Is there any difference in the precision of the result? Compare the number of function and gradient evaluations which are given in the standard output of optim.*

optim() function is used for minimization hence we defined negative logarithm of the function and negative derivate. BFGS and the Nelder-Mead algorithms were used.

```
## Optimal beta with BFGS: -0.009356126 1.262813

## Likelihood: -6.484279

## $par
## [1] -0.009356126  1.262812832
##
## $value
## [1] 6.484279
##
## $counts
## function gradient
##       12        8
##
## $convergence
## [1] 0
##
## $message
## NULL

## Optimal beta with Nelder-Mead: -0.009423433 1.262738

## Likelihood: -6.484279

## $par
## [1] -0.009423433  1.262738266
##
## $value
## [1] 6.484279
##
## $counts
## function gradient
##       47       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Both of the method give the same value and almost same parameters with part b. Nelder-Mead doesn't have the gradient count. Since it doesn't use gradient information, the Nelder-Mead method may require more

Method	function count	gradient count	parameters	value
BFGS	12	8	(-0.009356126, 1.262812832)	-6.484279
Nelder-Mead	47	NA	(-0.009423433, 1.262738)	-6.484279

Table 3: Comparison of BFGS and Nelder-Mead

function evaluations to converge, especially in high-dimensional spaces. Function count of the BFGS method is less than Nelder-Mead because it incorporates information about the local curvature of the objective function. Both method has less function & gradient (BFGS) count compared to part b.

part d)

Question: *Use the function glm in R to obtain an ML-solution and compare it with your results before.* Summary of the logistic regression with glm() function is as follows:

```
data <- data.frame(x = x, y = y)
model <- glm(y ~ x, data = data, family = "binomial")
cat("Optimal beta with glm():", model$coefficients, "\n")
```

```
## Optimal beta with glm(): -0.009359853 1.262823
```

```
cat("Likelihood:", log_logistic_reg(X, y, model$coefficients), "\n")
```

```
## Likelihood: -6.484279
```

glm() fucntion gives the same loglikelihood value with Nelder-Mead, BFGS and part b.

Method	Parameters	Value	Accuracy
Steepest Ascent $(\alpha_0 = 1)$, $(\beta_0, \beta_1) = (-0.2, 1)$	(-0.009339359, 1.262764)	-6.484279	0.7
Steepest Ascent $(\alpha_0 = 10)$, $(\beta_0, \beta_1) = (-0.2, 1)$	(-0.00934389, 1.262777)	-6.484279	0.7
Steepest Ascent $(\alpha_0 = 3)$, $(\beta_0, \beta_1) = (5, -5)$	(-0.009355319, 1.262825)	-6.484279	0.7
BFGS	(-0.009356126, 1.262812832)	-6.484279	0.7
Nelder-Mead	(-0.009423433, 1.262738266)	-6.484279	0.7
glm()	(-0.009359853, 1.262823430)	-6.484279	0.7

Table 4: Comparison of the Methods

The parameters of all the methods is almost same and they all yield to same accuracy in the model. Accuracy is calculated putting the obtained beta values to p(x). The code for the calculation can be seen in the Appendix.

Appendix

QUESTION 1:

part a)

```r
# Define the function, gradient and hessian matrix
g <- function(x,y){
  result_g <- -x^2 - x^2*y^2 - 2*x*y + 2*x + 2
  return(result_g)
}

gradient_g <- function(x, y){
  dx <- -2*x - 2*x*y^2 - 2*y + 2
  dy <- -2*y*x^2 - 2*x
  grad_mat <- matrix(c(dx, dy), ncol = 1)
  return(grad_mat)
}

hessian_g <- function(x,y){
  dxx <- -2 - 2*y^2
  dyx <- -4*x*y -2
  dxy <- -4*x*y -2
  dyy <- -2*x^2
  hessian_mat <- matrix(c(dxx,dyx,dxy,dyy), ncol = 2)
  return(hessian_mat)
}
```

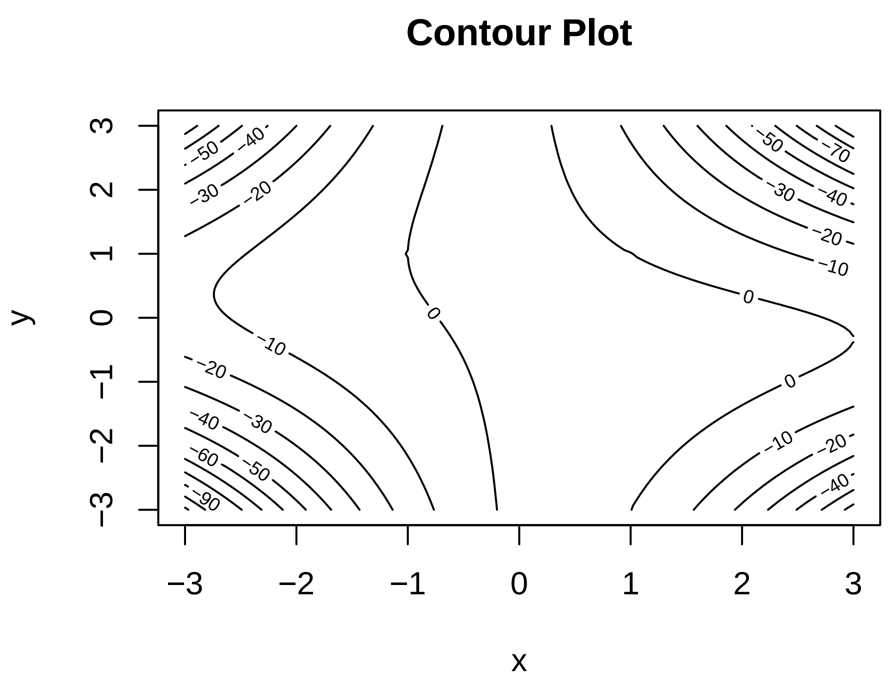
```r
# Generate x and y values for the contour plot
x_vals <- seq(-3, 3, length.out = 100)
y_vals <- seq(-3, 3, length.out = 100)

# Create a grid of x and y values
grid <- expand.grid(x = x_vals, y = y_vals)

# Evaluate the function g at each point in the grid
grid$z <- apply(grid, 1, function(row) g(row[1], row[2]))

# Create the contour plot
contour(x = x_vals, y = y_vals, z = matrix(grid$z, nrow = length(x_vals), ncol = length(y_vals)), main =

# Add labels and a color scale
xlabel <- expression(x)
ylabel <- expression(y)
title(main = "Contour Plot", xlab = xlabel, ylab = ylabel)
```



part b)

```r
newton_optimization <- function(x, y, eps, max_iter){
  xt <- matrix(c(x,y), ncol= 1)
  iter <- 0
  while (iter < max_iter){
    hessian_mat <- hessian_g(xt[1,1],xt[2,1])
    gradient_mat <- gradient_g(xt[1,1],xt[2,1])
    xt1 <- xt - solve(hessian_mat) %*% gradient_mat
    # stopping condition
    if (dist(xt- xt1) <= eps ){
      break
    }
    xt <- xt1
    iter <- iter+1
  }
  newton_list = list(value = g(xt1[1], xt1[2]),
                     param = c(x = xt1[1], y = xt1[2]),
                     iteration = iter)
  return(newton_list)
}
```

part c)

```r
cat("Starting value is (x,y) = (2,0):\n")
```

```
## Starting value is (x,y) = (2,0):
```

```r
x <- 2
y <- 0
fnc <- newton_optimization(x, y, eps = 1e-10, max_iter = 10000)
x_new <- fnc$param[1]
```

```r
y_new <- fnc$param[2]
cat("Newton Optimization Result:\n")
```

```
## Newton Optimization Result:
```

```r
fnc
```

```
## $value
## [1] 4
##
## $param
##  x  y
##  1 -1
##
## $iteration
## [1] 7
```

```r
cat("Gradient matrix:\n")
```

```
## Gradient matrix:
```

```r
gradient_g(x_new,y_new)
```

```
##      [,1]
## [1,]    0
## [2,]    0
```

```r
cat("Hessian matrix:\n")
```

```
## Hessian matrix:
```

```r
hessian_g(x_new,y_new)
```

```
##      [,1] [,2]
## [1,]   -4    2
## [2,]    2   -2
```

```r
cat("Eigenvalues:\n", eigen(hessian_g(x_new,y_new))$values, "\n")
```

```
## Eigenvalues:
##  -0.763932 -5.236068
```

```r
cat("Starting value is (x,y) = (-1,-2):\n")
```

```
## Starting value is (x,y) = (-1,-2):
```

```r
x <- -1
y <- -2
fnc <- newton_optimization(x, y, eps = 1e-10, max_iter = 10000)
x_new <- fnc$param[1]
y_new <- fnc$param[2]
cat("Newton Optimization Result:\n")
```

```
## Newton Optimization Result:
```

```r
fnc
```

```
## $value
## [1] 2
##
## $param
## x y
## 0 1
##
## $iteration
## [1] 16
```

```
cat("Gradient matrix:\n")
```

```
## Gradient matrix:
```

```
gradient_g(x_new,y_new)
```

```
##      [,1]
## [1,]    0
## [2,]    0
```

```
cat("Hessian matrix:\n")
```

```
## Hessian matrix:
```

```
hessian_g(x_new,y_new)
```

```
##      [,1] [,2]
## [1,]   -4   -2
## [2,]   -2    0
```

```
cat("Eigenvalues:\n", eigen(hessian_g(x_new,y_new))$values, "\n")
```

```
## Eigenvalues:
##  0.8284271 -4.828427
```

```
cat("Starting value is (x,y) = (0,1):\n")
```

```
## Starting value is (x,y) = (0,1):
```

```
x <- 0
y <- 1
fnc <- newton_optimization(x, y, eps = 1e-10, max_iter = 10000)
x_new <- fnc$param[1]
y_new <- fnc$param[2]
cat("Newton Optimization Result:\n")
```

```
## Newton Optimization Result:
```

```
fnc
```

```
## $value
## [1] 2
##
## $param
## x y
## 0 1
##
## $iteration
## [1] 0
```

```
cat("Gradient matrix:\n")
```

```
## Gradient matrix:
```

```
gradient_g(x_new,y_new)
```

```
##      [,1]
## [1,]    0
## [2,]    0
```

```
cat("Hessian matrix:\n")
```

```
## Hessian matrix:
```

```
hessian_g(x_new,y_new)
```

```
##      [,1] [,2]
## [1,]   -4   -2
## [2,]   -2    0
```

```
cat("Eigenvalues:\n", eigen(hessian_g(x_new,y_new))$values, "\n")
```

```
## Eigenvalues:
##  0.8284271 -4.828427
```

```
cat("Starting value is (x,y) = (0,0):\n")
```

```
## Starting value is (x,y) = (0,0):
```

```
x <- 0
y <- 0
fnc <- newton_optimization(x, y, eps = 1e-10, max_iter = 10000)
x_new <- fnc$param[1]
y_new <- fnc$param[2]
cat("Newton Optimization Result:\n")
```

```
## Newton Optimization Result:
```

```
fnc
```

```
## $value
## [1] 2
##
## $param
## x y
## 0 1
##
## $iteration
## [1] 1
```

```
cat("Gradient matrix:\n")
```

```
## Gradient matrix:
```

```
gradient_g(x_new,y_new)
```

```
##      [,1]
## [1,]    0
## [2,]    0
```

```
cat("Hessian matrix:\n")
```

```
## Hessian matrix:
```

```
hessian_g(x_new,y_new)
```

```
##      [,1] [,2]
## [1,]   -4   -2
## [2,]   -2    0
```

```
cat("Eigenvalues:\n", eigen(hessian_g(x_new,y_new))$values, "\n")
```

```
## Eigenvalues:
##  0.8284271 -4.828427
```

```
x_values <- seq(-3, 3, by = 0.01)
y_values <- seq(-3, 3, by = 0.01)

# Create a data frame with all combinations of x and y
combinations <- expand.grid(x = x_values, y = y_values)

# Calculate the corresponding g values for each combination
combinations$g_values <- mapply(g, combinations$x, combinations$y)

max_row <- combinations[which.max(combinations$g_values), ]
cat("Maximum point (x, y):", max_row$x, max_row$y, "\n")
```

```
## Maximum point (x, y): 1 -1
```

```
cat("Maximum value:", max_row$g_values, "\n")
```

```
## Maximum value: 4
```

```
g_optimize <- function(init){
  x <- init[1]
  y <- init[2]
  result_g <- -x^2 - x^2*y^2 - 2*x*y + 2*x + 2
  return(-result_g)
}
initial_values <- c(2,0)
max_val <- optim(par = initial_values, fn = g_optimize, method = "L-BFGS-B", lower = c(-3, -3), upper =
cat("Maximum point (x,y):", max_val$par, "\n")
```

```
## Maximum point (x,y): 1 -1
```

```
cat("Maximum value:", -max_val$value, "\n")
```

```
## Maximum value: 4
```

QUESTION 2:

```
# DATA
x <- c(0, 0, 0, 0.1, 0.1, 0.3, 0.3, 0.9, 0.9, 0.9)
X <- cbind(1, as.matrix(x))
y <- c(0, 0, 1, 0, 1, 1, 1, 0, 1, 1)
beta <- matrix(c(-0.2, 1), ncol = 1)
```

```
#calculate prob for each x
logistic_reg <- function(X, beta){
  z <- X %*% beta
  p <- 1/(1+exp(-z))
  return(p)
}

#calculate loglikelihood with given data and parameter
log_logistic_reg <- function(X, y, beta){
```

```r
  p <- logistic_reg(X, beta)
  result <- sum(y*log(p) + (1-y)*log(1-p))
  return(result)
}

# calculate gradient for both beta
gradient <- function(X,y, beta){
  diff <- y-logistic_reg(X, beta)
  grad <- t(X) %*% as.matrix(diff)
  return(grad)
}
```

part a,b)

```r
# The gradient ascent function
gradient_ascent <- function(alpha0, beta_init, eps, max_iter){
  iter <- 0
  alpha <- alpha0
  beta_t <- beta_init
  beta_t1 <- beta_t + alpha * gradient(X, y, beta_t)
  g_t <- log_logistic_reg(X, y, beta_t)
  gradient_t <- gradient(X, y, beta_t)

  loglikelihoods <- c(g_t)
  eval_count <- 1  # Counter for function evaluations
  grad_count <- 1  # Counter for gradient evaluations

  while (iter < max_iter) {
    g_t1 <- log_logistic_reg(X, y, beta_t1)
    eval_count <- eval_count + 1  # Increment function evaluation counter

    # find the stepsize that increases the loglikelihood
    while (g_t1 < g_t){
      alpha <- alpha/2
      beta_t1 <- beta_t + alpha * gradient_t
      g_t1 <- log_logistic_reg(X, y, beta_t1)
      eval_count <- eval_count + 1  # Increment function evaluation counter
      cat("Learning rate changed!\n")
    }

    # assign beta_t1 and g_t1 as previous values
    beta_t <- beta_t1
    g_t <- g_t1

    # calculate new beta
    gradient_t <- gradient(X, y, beta_t)
    beta_t1 <- beta_t + alpha * gradient_t
    grad_count <- grad_count + 1  # Increment gradient evaluation counter

    # save values
    loglikelihoods <- c(loglikelihoods, g_t1)
    iter <- iter + 1
    # stopping condition
    if (abs(beta_t[1] - beta_t1[1]) <= eps && abs(beta_t[2] - beta_t1[2]) <= eps){
      break
    }
  } #end the loop

  result_list <- list(par = t(beta_t1),
                      value = loglikelihoods[length(loglikelihoods)],
                      counts = c('function' = eval_count, 'gradient' = grad_count),
                      iteration = iter)
  return(result_list)
}
```

```
alpha0 <- 1
beta_init <- matrix(c(-0.2, 1), ncol= 1)
eps <- 1e-5
max_iter <- 10000
result1 <- gradient_ascent(alpha0, beta_init, eps, max_iter)
```

```
## Learning rate changed!
```

```
cat("Alpha:", alpha0, "\n")
```

```
## Alpha: 1
```

```
result1
```

```
## $par
##              [,1]     [,2]
## [1,] -0.009339359 1.262764
##
## $value
## [1] -6.484279
##
## $counts
## function gradient
##       58       57
##
## $iteration
## [1] 56
```

```
alpha0 <- 10
beta_init <- matrix(c(-0.2, 1), ncol= 1)
eps <- 1e-5
max_iter <- 10000
result2 <- gradient_ascent(alpha0, beta_init, eps, max_iter)
```

```
## Learning rate changed!
## Learning rate changed!
## Learning rate changed!
## Learning rate changed!
```

```
cat("Alpha:", alpha0, "\n")
```

```
## Alpha: 10
```

```
result2
```

```
## $par
##             [,1]     [,2]
## [1,] -0.00934389 1.262777
##
## $value
## [1] -6.484279
##
## $counts
## function gradient
##       50       46
##
## $iteration
## [1] 45
```

```
alpha0 <- 3
beta_init <- matrix(c(5, -5), ncol= 1)
eps <- 1e-5
max_iter <- 10000
result3 <- gradient_ascent(alpha0, beta_init, eps, max_iter)
```

```
## Learning rate changed!
## Learning rate changed!
```

```
cat("Alpha:", alpha0, "\n")
```

```
## Alpha: 3
```

```
result3
```

```
## $par
##              [,1]     [,2]
## [1,] -0.009355319 1.262825
##
## $value
## [1] -6.484279
##
## $counts
## function gradient
##      100       98
##
## $iteration
## [1] 97
```

part c)

```
# optim() function minimizes so I need to define new functions
neg_log_likelihood <- function(X, y, beta) {
  return(-log_logistic_reg(X, y, beta))
}

neg_gradient <- function(X, y, beta) {
  return(-gradient(X, y, beta))  # We minimize the negative log-likelihood
}
```

```
result_BFGS <- optim(par = as.vector(beta), fn = neg_log_likelihood, gr = neg_gradient, X = X, y = y, m
optimal_beta_BFGS <- matrix(result_BFGS$par, ncol = 1)
```

```r
cat("Optimal beta with BFGS:", optimal_beta_BFGS, "\n")
```

```
## Optimal beta with BFGS: -0.009356126 1.262813
```

```r
cat("Likelihood:", log_logistic_reg(X, y, optimal_beta_BFGS), "\n") # The same result
```

```
## Likelihood: -6.484279
```

```r
result_BFGS
```

```
## $par
## [1] -0.009356126  1.262812832
##
## $value
## [1] 6.484279
##
## $counts
## function gradient
##       12        8
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```r
result_NM <- optim(par = as.vector(beta), fn = neg_log_likelihood, gr = neg_gradient, X = X, y = y, met
optimal_beta_NM <- matrix(result_NM$par, ncol = 1)

cat("Optimal beta with Nelder-Mead:", optimal_beta_NM, "\n")
```

```
## Optimal beta with Nelder-Mead: -0.009423433 1.262738
```

```r
cat("Likelihood:", log_logistic_reg(X, y, optimal_beta_NM), "\n") # The same result
```

```
## Likelihood: -6.484279
```

```r
result_NM
```

```
## $par
## [1] -0.009423433  1.262738266
##
## $value
## [1] 6.484279
##
## $counts
## function gradient
##       47       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

part d)

```
data <- data.frame(x = x, y = y)
model <- glm(y ~ x, data = data, family = "binomial")
summary(model)
```

```
##
## Call:
## glm(formula = y ~ x, family = "binomial", data = data)
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -0.00936    0.87086  -0.011    0.991
## x            1.26282    1.86663   0.677    0.499
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 13.460  on 9  degrees of freedom
## Residual deviance: 12.969  on 8  degrees of freedom
## AIC: 16.969
##
## Number of Fisher Scoring iterations: 4
```

```
cat("Coefficients:", model$coefficients, "\n")
```

```
## Coefficients: -0.009359853 1.262823
```

```
cat("Value:", log_logistic_reg(X, y, model$coefficients), "\n")
```

```
## Value: -6.484279
```

```
accuracy <- function(x,y, beta_MLE){
  X <- cbind(1, as.matrix(x))
  beta_new <- matrix(c(beta_MLE[1], beta_MLE[2]), ncol = 1)
  probs <- logistic_reg(X, beta_new)
  pred <- c()
  for (i in 1:10){
    if (probs[i] > 0.5){
      pred <- c(pred, 1)
    } else{
      pred <- c(pred, 0)
    }
  }
  acc <- mean(pred == y)
  result_accuracy <- list(pred = pred, accuracy = acc)
  return(result_accuracy)
}
```

```
accuracy(x,y, result1$par)
```

```
## $pred
##  [1] 0 0 0 1 1 1 1 1 1 1
##
## $accuracy
## [1] 0.7
```

```r
accuracy(x,y, result2$par)
```

```
## $pred
##  [1] 0 0 0 1 1 1 1 1 1 1
##
## $accuracy
## [1] 0.7
```

```r
accuracy(x,y, result3$par)
```

```
## $pred
##  [1] 0 0 0 1 1 1 1 1 1 1
##
## $accuracy
## [1] 0.7
```

```r
accuracy(x,y, result_BFGS$par)
```

```
## $pred
##  [1] 0 0 0 1 1 1 1 1 1 1
##
## $accuracy
## [1] 0.7
```

```r
accuracy(x,y, result_NM$par)
```

```
## $pred
##  [1] 0 0 0 1 1 1 1 1 1 1
##
## $accuracy
## [1] 0.7
```

```r
accuracy(x,y, model$coefficients)
```

```
## $pred
##  [1] 0 0 0 1 1 1 1 1 1 1
##
## $accuracy
## [1] 0.7
```