Lab 2 - Computational Statistics (732A89)

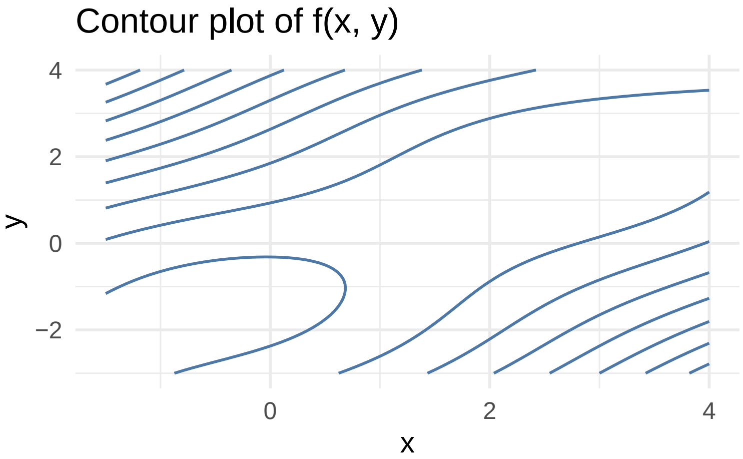
Helena Llorens Lluís (hllor282), Yi Yang (yiyan338)

QUESTION 1

First, we plot the function

$$f(x, y) = sin(x + y) + (x - y)^2 - 1.5x + 2.5y + 1$$

within the intervals $x \in [-1.5, 4]$ and $y \in [-3, 4]$.



With both the gradient and the Hessian matrix calculated, we have been able to compute the Newton algorithm to find local minimums of this function. This function takes as input both the gradient and the Hessian matrix, a pair of starting values x_0 and y_0, and a tolerance (set as default at 0.0001) and a maximum of iterations (set at 100 as default). We tried the function for 3 different starting points and the results are presented in the following table.

Table 1: Newton algorithm results

initial_x	initial_y	final_x	final_y	function_value	eigenvalues1	eigenvalues2
1	3	2.5943951	1.5943951	1.228370	4	1.732051
2	4	1.5471976	0.5471976	1.913223	4	-1.732051
0	-1	-0.5471976	-1.5471976	-1.913223	4	1.732051

A result is considered a local minimum if all its eigenvalues are positive. As shown in the table, two of the three solutions satisfy this condition: (2.5943951, 1.5943951) and $(-0.5471976, -1.5471976)$. Among them, we identify the latter as the global minimum, as it yields the lowest function value, -1.913223.

On the other hand, (1.5471976, 0.5471976) is classified as a saddle point since one of its eigenvalues is negative.

QUESTION 2

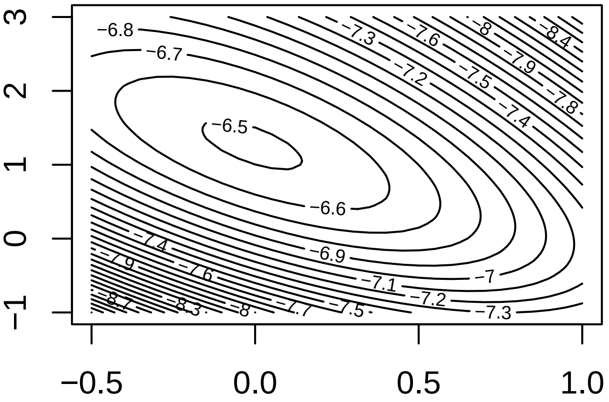
First,we implemented a maximum likelihood estimator using the steepest ascent method with a step-size-reducing line search.The function also counts the number of evaluations of the log-likelihood function and its gradient to monitor computational efficiency.The log likelihod is given by

$$g(\mathbf{b}) = \sum_{i=1}^{n} \left[y_i \log \left(\frac{1}{1+\exp(-\beta_0 - \beta_1 x_i)} \right) + (1-y_i) \log \left(1 - \frac{1}{1+\exp(-\beta_0 - \beta_1 x_i)} \right) \right]$$

where $\mathbf{b} = (\beta_0, \beta_1)^T$.The gradient of the log likelihood function is:

$$g'(\mathbf{b}) = \sum_{i=1}^{n} \left[y_i - \frac{1}{1+\exp(-\beta_0 - \beta_1 x_i)} \right] \binom{1}{x_i}$$

Then we generated a contour plot, which illustrates the shape of the log-likelihood surface.From the plot we could roughly identify the location of global /local maximum.



Then we implemented the algorithm with two variants of the backtracking line search method:

The first variant keeps α as its initial value $\alpha_0 = 1$, while the second variant uses a reduced α.

Both strategies were initialized with $\beta_0 = -0.2$ and $\beta_1 = 1$, and the algorithm was stopped when the norm of the gradient was smaller than 10^{-5} to ensure convergence.

The number of function and gradient evaluations performed to convergence are shown as follows:

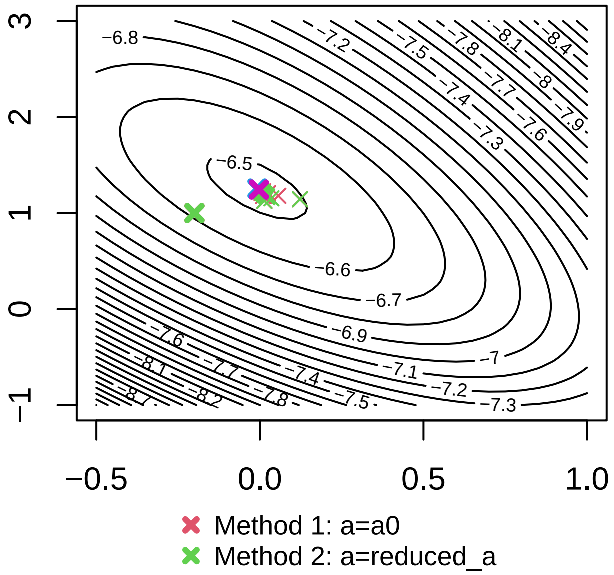


Table 2: Steepest Ascent Method Results

Method	B0	B1	Counts_function	Counts_gradiet
a=ao	-0.0072995	1.254972	35	14
a=reduced_a	-0.0031586	1.244742	33	16

Comparison

Due to different choices of step sizes, the paths of parameter updates vary slightly, leading to some differences in the final parameter estimates.
The first method(fixed step strategy) has slightly more function evaluations but fewer gradient evaluations.

The second method(reduced step size strategy) has slightly fewer function evaluations but more gradient evaluations.

Reason for Differences

The first method: Since the step size is fixed, fewer function evaluations are required per iteration. However, because the step size is relatively large, the algorithm may "miss" the optimal value during updates, potentially requiring more iterations to converge.

The second method: More gradient evaluations are needed because the gradient must be recalculated after each step size adjustment.

We used function `optim` with both the `BFGS` and the `Nelder-Mead` algorithm, the number of function and gradient evaluations and the coefficients show as follows:

Table 3: BFGS and Nelder-Mead Results

Method	Value	B0	B1	Counts_function	Counts_gradiet
BFGS	-6.484279	-0.0093561	1.262813	12	8
Nelder-Mead	-6.484279	-0.0094234	1.262738	47	NA

Comparison

Results:
The results from BFGS and Nelder-Mead are similar but not identical from part b. However, the difference in β_0 and β_1 are relatively small due to differences in the optimization algorithms and their convergence criteria.

The precision of the result:
The precision of the results is high across all method. The small differences indicate that all method converge to a similar solution.

The number of function and gradient evaluations:
BFGS is the most efficient method, requiring only 12 function evaluations and 8 gradient evaluations.This efficiency is due to its use of gradient information and its ability to approximate the Hessian matrix, which accelerates convergence. Nelder-Mead doesn't require the computation of derivatives and relies sole on function evaluations.It requires 47 function evaluation.

We used function glm and the results are as follows:

```
## B0: -0.009359853
```

```
## B1: 1.262823
```

Comparison

1.The results from BFGS and Nelder-Mead are almost identical to those from glm, indicating that these methods achieve high precision.

2.The results from the steepest ascent method show some deviation from glm , suggesting slightly lower precision. This may be due to its slower convergence rate or step size selection. However, the results are still within a reasonable range.

Appendix

Question 1

```
# define the function f(x, y) that returns f(x, y) if both x and y are in the interval
f <- function(x, y){
  ifelse(x >= -1.5 & x <= 4 & y >= -3 & y <= 4,
         sin(x + y) + (x - y)^2 - 1.5*x + 2.5*y + 1,
         NA)
}

# data frame with x, y and f(x, y)
x <- seq(-1.5, 4, length.out = 100)
y <- seq(-3, 4, length.out = 100)
```

```
grid <- expand.grid(x = x, y = y)
grid$z <- with(grid, f(x, y))

# plot f(x, y)
ggplot(grid, aes(x = x, y = y, z = z)) +
  geom_contour(col = "#4E79A7") +
  labs(x = "x", y = "y", title = "Contour plot of f(x, y)") +
  theme_minimal()

# gradient for f(x, y)
gradient <- function(x, y){
  grad_x <- cos(x + y) + 2*(x - y) - 1.5
  grad_y <- cos(x + y) - 2*(x - y) + 2.5
  return(c(grad_x, grad_y))
}

# hessian matrix for f(x, y)
hessian <- function(x, y){

  m <- matrix(c(-sin(x + y) + 2, -sin(x + y) - 2,
              -sin(x + y) - 2, -sin(x + y) + 2),
              byrow = T, nrow = 2, ncol = 2)

  return(m)
}

# newton algorithm
newton <- function(gradient, hessian, x_init, y_init, tol = 0.0001, maxit = 100){

  # check that the initial values are within the intervals
  if(x_init < -1.5 || x_init > 4 || y_init < -3 || y_init > 4){
    stop("The x and y values have to be within the intervals")
  }

  # set initial values
  x0 <- c(x_init, y_init)

  # while the stopping criteria is not met
  for(it in 1:maxit){

    H <- hessian(x0[1], x0[2])

    # use MASS library in case of a singular Hessian matrix
    if(abs(det(H)) < 1e-6){
      x1 <- x0 - as.vector(MASS::ginv(H) %*% gradient(x0[1], x0[2]))
    } else{
      x1 <- x0 - as.vector(solve(H) %*% gradient(x0[1], x0[2]))
    }

    # set the values within the intervals
    x1[1] <- max(min(x1[1], 4), -1.5)
    x1[2] <- max(min(x1[2], 4), -3)
    # if the stopping criteria is met
    if(norm(x1 - x0, type = "2") < tol){  # t(x1 - x0) %*% (x1 - x0)
      return(list(x = x1[1], y = x1[2], iterations = it,
                  gradient = gradient(x1[1], x1[2]), eigenvalues = eigen(hessian(x1[1], x1[2]))$values))
    }

    x0 <- x1
  }

  warning("Maximum number of iterations reached without convergence.")
  return(list(x = x1[1], y = x1[2], iterations = maxit,
              gradient = gradient(x1[1], x1[2]), eigenvalues = eigen(hessian(x1[1], x1[2]))$values))
}

# trying algorithm with different starting points
x_init1 <- 1
y_init1 <- 3
res1 <- newton(gradient, hessian, x_init1, y_init1)

x_init2 <- 2
y_init2 <- 4
res2 <- newton(gradient, hessian, x_init2, y_init2)

x_init3 <- 0
y_init3 <- -1
res3 <- newton(gradient, hessian, x_init3, y_init3)

# table of results
df <- data.frame(initial_x = c(x_init1, x_init2, x_init3),
                 initial_y = c(y_init1, y_init2, y_init3),
                 final_x = c(res1$x, res2$x, res3$x),
                 final_y = c(res1$y, res2$y, res3$y),
                 function_value= c(f(res1$x, res1$y), f(res2$x, res2$y), f(res3$x, res3$y)),
                 eigenvalues1 = c(res1$eigenvalues[1], res2$eigenvalues[1], res3$eigenvalues[1]),
                 eigenvalues2 = c(res1$eigenvalues[2], res2$eigenvalues[2], res3$eigenvalues[2]))
kable(df, caption = "Newton algorithm results")
```

Question 2

```
#a

#log likelihood
g <- function(b){
  xi <- c(0,0,0,0.1,0.1,0.3,0.3,0.9,0.9,0.9)
  yi <- c(0,0,1,0,1,1,1,0,1,1)
  p <- 1/(1+exp(-b[1]-b[2]*xi))    #b[1] represents b0,b2 represents b1
  loglik <- sum(yi*log(p)+(1-yi)*log(1-p))
  return(loglik)
}

#gradient
```

```
dg <- function(b){
  xi <- c(0,0,0,0.1,0.1,0.3,0.3,0.9,0.9,0.9)
  yi <- c(0,0,1,0,1,1,1,0,1,1)
  p <- 1/(1+exp(-b[1]-b[2]*xi))

  dg_b0 <- sum(yi-p) #partial derivative with respect to b0
  dg_b1 <- sum(xi*(yi-p)) #partial derivative with respect to b1
  return(c(dg_b0,dg_b1))
}

#produce the contours plot
x1grid <- seq(-0.5, 1, by=0.05)
x2grid <- seq(-1, 3, by=0.05)
dx1 <- length(x1grid)
dx2 <- length(x2grid)
dx  <- dx1*dx2
gx  <- matrix(rep(NA, dx), nrow=dx1)
for (i in 1:dx1)
  for (j in 1:dx2)
  {
    gx[i,j] <- g(c(x1grid[i], x2grid[j]))
  }
mgx <- matrix(gx, nrow=dx1, ncol=dx2)
contour(x1grid, x2grid, mgx, nlevels=30)  # Note: For other functions g, you might need to choose anoth

#Steepest ascent function:
steepestasc <- function(x0, eps=1e-8, alpha0=1,path_col=2, final_col=4)
{
  xt   <- x0
  conv <- 999

  #function and gradient evaluations
  funEvaluation <- 0
  gradEvaluation <- 0

  points(xt[1], xt[2], col=path_col, pch=4, lwd=3)


  while(conv>eps)
  {
    alpha <- alpha0
    xt1   <- xt
    grad_xt1 <- dg(xt1)
    xt    <- xt1 + alpha*grad_xt1

    #update gradient evaluation
    gradEvaluation <-gradEvaluation+1

    #update function evaluation
    g_x <- g(xt)
    funEvaluation <-funEvaluation+1
    g_x1 <- g(xt1)
    funEvaluation <-funEvaluation+1
    while (g_x<g_x1 )
    {
      alpha <- alpha/2
      xt    <- xt1 + alpha*grad_xt1

      g_x <- g(xt)
      funEvaluation <-funEvaluation+1
    }
    points(xt[1], xt[2],col=path_col,pch=4, lwd=1)
    conv <- sum((xt-xt1)*(xt-xt1))
  }
  points(xt[1], xt[2], col=final_col ,pch=4, lwd=3)
  return(list(coefficients=xt,

              funEvaluation=funEvaluation,
              gradientEvaluation=gradEvaluation))
}

#b

inital_b <- c(-0.2,1)

#the second solution: a=a

steepestasc_keep <- function(x0, eps=1e-5, alpha0=1, path_col=3, final_col=6){

  xt   <- x0
  conv <- 999

  #function and gradient evaluations
  funEvaluation <- 0
  gradEvaluation <- 0

  points(xt[1], xt[2], col=path_col, pch=4, lwd=3)


  alpha <- alpha0

  while(conv>eps)
  {

    xt1   <- xt
    grad_xt1 <- dg(xt1)
    xt    <- xt1 + alpha*grad_xt1

    #update gradient evaluation
    gradEvaluation <-gradEvaluation+1

    #update function evaluation
    g_x <- g(xt)
    funEvaluation <-funEvaluation+1
    g_x1 <- g(xt1)
    funEvaluation <-funEvaluation+1
    while (g(xt)<g(xt1))
    {
      alpha <- alpha/2
      xt    <- xt1 + alpha*grad_xt1

      g_x <- g(xt)
      funEvaluation <- funEvaluation + 1
    }
    points(xt[1], xt[2],col=path_col,  pch=4, lwd=1)
    conv <- sum((xt-xt1)*(xt-xt1))
  }
  points(xt[1], xt[2], col=final_col, pch=4, lwd=3)
  return(list(coefficients=xt,
              funEvaluation=funEvaluation,
              gradientEvaluation=gradEvaluation))


}

#first solution: a=a0

print(steepestasc(inital_b,eps=1e-5, alpha0=1))
#second solution:

print(steepestasc_keep(inital_b,eps=1e-5, alpha0=1))



#c

# BFGS
optim_BFGS <- optim(par = inital_b,fn=g,gr=dg,method = "BFGS",control = list(fnscale = -1))
print(optim_BFGS)

#Nelder-Mead


optim_NM <- optim(par = inital_b,fn=g,gr=dg,method ="Nelder-Mead",control = list(fnscale = -1))
print(optim_NM)



#d.glm
xi <- c(0,0,0,0.1,0.1,0.3,0.3,0.9,0.9,0.9)
yi <- c(0,0,1,0,1,1,1,0,1,1)
fit <- glm(yi~xi,family = binomial)
cat("The coefficients are:","\n",coef(fit))
```