Computational Statistics Computer Lab 2 (Group 7)

Qinyuan Qi(qinqi464) Satya Sai Naga Jaya Koushik Pilla (satpi345)

2023-11-14

Question 1: Optimisation of a two-dimensional function (Solved by Satya Sai Naga Jaya Koushik Pilla)

Answer:

(A)

The gradient and the Hessian matrix is as follows:

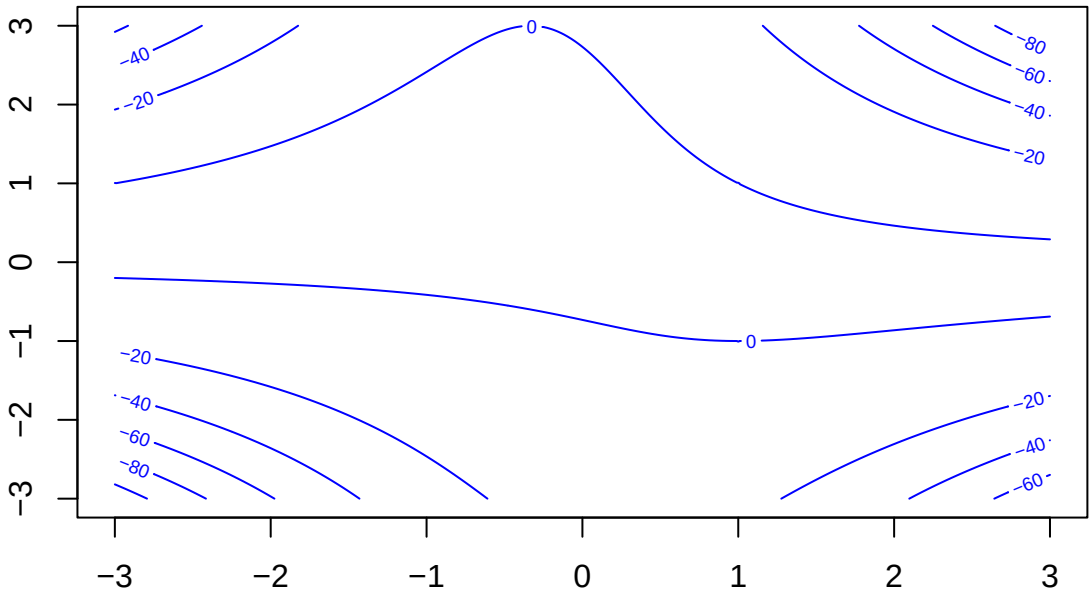
Gradient of g is:

$$\begin{pmatrix} -2x - 2xy^2 - 2y + 2 \\ -2x^2y - 2x \end{pmatrix}$$

Hessian of g:

$$\begin{pmatrix} -2y^2 - 2 & -4xy - 2 \\ -4xy - 2 & -2x^2 \end{pmatrix}$$

The related contour plot as follows:



(B)

The following is the code for the newton method.

```r
########################## [ 1 b ] ##############################################
# define distance of two points
distance <- function(xt1, xt0) {
  ret <- t(xt1 - xt0) %*% (xt1 - xt0)
  return(ret[1][1])
}

# newton function with eps = 1e-8 and max_step = 1000
# x0 is the initial value
newton <- function(x0, eps = 1e-8, max_step = 1000) {
  xt0  <- x0
  for (i in 1:max_step){
    xt1  <- xt0 - solve(d2g(xt0[1], xt0[2])) %*% dg(xt0[1], xt0[2])
    criterion <- distance(xt1, xt0)
    # we get the qualified result
    if (criterion < eps) {
      cat("Converged after ", i, " steps\n")
      return(xt1)
    }
    xt0 <- xt1
  }
  # after max_step steps, we still cannot get the qualified result
  cat("Did not converge after", max_step, " steps\n")
  return(NULL)
}
```

(C)

Let's use (2,0),(-1,2),(0,-1) and (0,2) as initial values.And we get the following results using the newton function above.

We find the following results $(1,-1)$,$(7.022621e-12,1)$,$(0,1)$,$(0,1)$.

Since the second one is almost same as $(0,1)$, so in total we get 2 points,$(1,-1)$ and $(0,1)$.

We can calculate the Gradient vector and Hessian matrix of g at these points using dg and d2g function defined. The result is as follows.All the calculation are based on the code provided in the appendix.

Point	Gradient	Hessian	Type
$(1,-1)$	$\begin{pmatrix}0\\0\end{pmatrix}$	$\begin{pmatrix}-4 & 2\\2 & -2\end{pmatrix}$	Local maximum
$(0,1)$	$\begin{pmatrix}0\\0\end{pmatrix}$	$\begin{pmatrix}-4 & -2\\-2 & 0\end{pmatrix}$	Saddle point

According to the table above and the contour plot, we can find that point $(0,1)$ are the global maximum if $x, y \in [-3,3]$ range.

(D)

The advantages of the steepest ascent algorithm is that it is easy to calculate without need to consider about the second derivative of the function. But the steepest ascent algorithm is not as efficient as the Newton method.

Newton method, can get the answer much faster than other methods,however, it cannot guaranteed that g(x) will increase every single step. And we have to calculate the second derivative of the function. Another thing

we need to mention is that we have to choose a proper initial value to get a convergent result. If we choose a bad initial value, we may get a diverge result since the Newton method is sensitive to the initial value.

Question 2 (Solved by Qinyuan Qi)

(A)

We implemented 2 version of algorithms using steepest ascent method.

First method not convergence, we reset the alpha to 1 every time when we loop. Second method convergence, we don't reset the alpha every time when we loop.

```
########################## [ 2 a ] ############################################
p <- function(beta, x_i) {
  b0 <- beta[1]
  b1 <- beta[2]
  return(1 / (1 + exp(-b0 - b1 * x_i)))
}

# Function g
g <- function(beta) {
  n <- length(x)
  ret <- 0
  for (i in 1:n){
    p_val <- p(beta, x[i])
    ret <- ret + y[i] * log(p_val) + (1 - y[i]) * log(1 - p_val)
  }
  return(ret)
}

# First derivative
dg <- function(beta) {
  n <- length(x)

  # init ret
  ret <- matrix(data = c(0, 0), ncol = 1)
  tmp <- 0
  for (i in 1:n){
    p_val <- p(beta, x[i])
    tmp <- tmp + y[i] - p_val
  }
  ret[1, 1] <- tmp
  ret[2, 1] <- tmp * x[i]
  return(ret)
}

distance <- function(beta0, beta1) {
  ret <- t(beta1 - beta0) %*% (beta1 - beta0)
  #ret <- g(beta1) - g(beta0)
  return(ret[1][1])
}

# steepest ascent method 1
# beta_init: the initial value
# alpha_init: the learning rate
```

```r
# eps: the threshold of the gradient
# max_step: the maximum number of steps
# rate: the rate of change of alpha when g not increase

steepest_ascent_1 <- function(beta_init, alpha_init = 1, eps = 1e-10,
                              max_step = 1000, rate = 0.5) {
  beta0 <- beta_init

  for (i in 1:max_step){
    alpha <- alpha_init

    beta1 <- beta0 + alpha * diag(2) %*% dg(beta0)
    criterion <- distance(beta1, beta0)
    dg0 <- dg(beta0)
    while (g(beta1) < g(beta0)) {
      alpha <-  alpha * rate
      beta1 <- beta0 + alpha * diag(2) %*% dg0
      print(alpha)
    }
    if (criterion < eps) {
      cat("Converged after ", i, " steps\n")
      return(beta1)
    }
    beta0 <- beta1
  }

  # after max_step steps, we still cannot get the qualified result
  cat("Did not converge after", max_step, " steps\n")
  return(NULL)
}


# steepest ascent method 2
# beta_init: the initial value
# alpha_init: the learning rate
# eps: the threshold of the gradient
# max_step: the maximum number of steps
# rate: the rate of change of alpha when g not increase

steepest_ascent_2 <- function(beta_init, alpha_init = 1, eps = 1e-10,
                              max_step = 1000, rate = 0.5) {
  beta0 <- beta_init
  alpha <- alpha_init
  for (i in 1:max_step){
    beta1 <- beta0 + alpha * diag(2) %*% dg(beta0)
    criterion <- distance(beta1, beta0)
    dg0 <- dg(beta0)
    while (g(beta1) < g(beta0)) {
      alpha <-  alpha * rate
      beta1 <- beta0 + alpha * diag(2) %*% dg0
    }
    if (criterion < eps) {
      cat("Converged after ", i, " steps\n")
```

```
      return(beta1)
    }
    beta0 <- beta1
  }

  # after max_step steps, we still cannot get the qualified result
  cat("Did not converge after", max_step, " steps\n")
  return(NULL)
}
```

(B)

First method not convergence, every time we reset the alpha to init value.

Second method convergence, we don't reset the alpha.

The result is: $\beta = (0.0169513, 1.1952562)$

(C)

According to the result, the two methods give almost the same result. compare to the result from 2b, we can say that there is no big difference.

The result of the function and the gradient of the function using the parameters calculated by above methods are as follows:

	b0	b1	g	g'
Own method	0.0169513	1.1952562	-6.484974	$\begin{pmatrix} -0.01309331 \\ -0.01178398 \end{pmatrix}$
BFGS	0.01248138	1.19123325	-6.485018	$\begin{pmatrix} -1.610715e-07 \\ -1.449644e-07 \end{pmatrix}$
Nelder-Mead	-0.009423433	1.262738266	-6.484279	$\begin{pmatrix} 0.0002044366 \\ 0.0001839929 \end{pmatrix}$
glm	-0.00936	1.26282	-6.484279	$\begin{pmatrix} 2.714971e-06 \\ 2.443474e-06 \end{pmatrix}$

(D)

We use glm to obtain the optimal parameters. code in appendix.

Compare to the result from the output of optim function, all the results are near $(0, 1.2)$

	β
Own method	(0.0169513,1.1952562)
BFGS	(0.01248138,1.19123325)
Nelder-Mead	(-0.009423433,1.262738266)
glm	(-0.00936,1.26282)

Appendix: All code for this report

```
################### Init code for Question 1 ###################################
knitr::opts_chunk$set(echo = TRUE)
rm(list = ls())
########################## [ 1 a ] #############################################
# Function g
g <- function(x, y) {
  ret <- -x^2 - x^2 * y^2 - 2 * x * y + 2 * x + 2
  return(ret)
}

# First derivative
dg <- function(x, y) {
  ret <- matrix(data = c(-2 * x - 2 * x * y^2 - 2 * y  + 2,
                         -2 * x^2 * y - 2 * x),
                ncol = 1, nrow = 2)
  return(ret)
}

# Second derivative
d2g <- function(x, y) {
  ret <- matrix(data = c(-2 * y^2 - 2, -4 * x * y - 2,
                         -4 * x * y - 2, -2 * x^2),
                ncol = 2, nrow = 2)
  return(ret)
}

# Draw contour plot
n <- 3
x <- seq(-n, n, by = 0.01)
y <- seq(-n, n, by = 0.01)
dx <- length(x)
dy <- length(y)
gx <- matrix(rep(NA, dx * dy), nrow = dx, ncol = dy)
gx <- sapply(y, function(i) sapply(x, function(j) g(i, j)))

contour(x, y, gx, nlevels = 7, col = "blue")

########################## [ 1 b ] #############################################
# define distance of two points
distance <- function(xt1, xt0) {
  ret <- t(xt1 - xt0) %*% (xt1 - xt0)
  return(ret[1][1])
}

# newton function with eps = 1e-8 and max_step = 1000
# x0 is the initial value
newton <- function(x0, eps = 1e-8, max_step = 1000) {
  xt0  <- x0
  for (i in 1:max_step){
    xt1  <- xt0 - solve(d2g(xt0[1], xt0[2])) %*% dg(xt0[1], xt0[2])
    criterion <- distance(xt1, xt0)
    # we get the qualified result
```

```
    if (criterion < eps) {
      cat("Converged after ", i, " steps\n")
      return(xt1)
    }
    xt0 <- xt1
  }
  # after max_step steps, we still cannot get the qualified result
  cat("Did not converge after", max_step, " steps\n")
  return(NULL)
}

########################## [ 1 c ] ##############################################

# we use the default and max_step.
newton_value_1 <- newton(matrix(c(2, 0), nrow = 2, ncol = 1))
newton_value_2 <- newton(matrix(c(-1, 2), nrow = 2, ncol = 1))
newton_value_3 <- newton(matrix(c(0, -1), nrow = 2, ncol = 1))
newton_value_4 <- newton(matrix(c(0, 2), nrow = 2, ncol = 1))

# we can calculate the Gradient vector and Hessian matrix of g at these points
# using dg and d2g function defined above. The result is as follows.

gradient_1 <- dg(1, -1)
gradient_2 <- dg(-0, 1)

hessian_1 <-  d2g(1, -1)
hessian_2 <-  d2g(0, 1)

check_type <- function(hessian_matrix) {
  eigenvalues <- eigen(hessian_matrix)$values
  if (all(eigenvalues > 0)) {
    cat("Local minimum\n")
  } else if (all(eigenvalues < 0)) {
    cat("Local maximum\n")
  } else {
    cat("Saddle point\n")
  }
}
check_type(hessian_1)
check_type(hessian_2)
#################### Init code for Question 2 ####################################
rm(list = ls())

x <- c(0, 0, 0, 0.1, 0.1, 0.3, 0.3, 0.9, 0.9, 0.9)
y <- c(0, 0, 1,   0,   1,   1,   1,   0,   1,   1)
########################## [ 2 a ] ##############################################
p <- function(beta, x_i) {
  b0 <- beta[1]
  b1 <- beta[2]
  return(1 / (1 + exp(-b0 - b1 * x_i)))
}

# Function g
```

```r
g <- function(beta) {
  n <- length(x)
  ret <- 0
  for (i in 1:n){
    p_val <- p(beta, x[i])
    ret <- ret + y[i] * log(p_val) + (1 - y[i]) * log(1 - p_val)
  }
  return(ret)
}

# First derivative
dg <- function(beta) {
  n <- length(x)

  # init ret
  ret <- matrix(data = c(0, 0), ncol = 1)
  tmp <- 0
  for (i in 1:n){
    p_val <- p(beta, x[i])
    tmp <- tmp + y[i] - p_val
  }
  ret[1, 1] <- tmp
  ret[2, 1] <- tmp * x[i]
  return(ret)
}

distance <- function(beta0, beta1) {
  ret <- t(beta1 - beta0) %*% (beta1 - beta0)
  #ret <- g(beta1) - g(beta0)
  return(ret[1][1])
}

# steepest ascent method 1
# beta_init: the initial value
# alpha_init: the learning rate
# eps: the threshold of the gradient
# max_step: the maximum number of steps
# rate: the rate of change of alpha when g not increase

steepest_ascent_1 <- function(beta_init, alpha_init = 1, eps = 1e-10,
                              max_step = 1000, rate = 0.5) {
  beta0 <- beta_init

  for (i in 1:max_step){
    alpha <- alpha_init

    beta1 <- beta0 + alpha * diag(2) %*% dg(beta0)
    criterion <- distance(beta1, beta0)
    dg0 <- dg(beta0)
    while (g(beta1) < g(beta0)) {
      alpha <-  alpha * rate
      beta1 <- beta0 + alpha * diag(2) %*% dg0
      print(alpha)
```

```
    }
    if (criterion < eps) {
      cat("Converged after ", i, " steps\n")
      return(beta1)
    }
    beta0 <- beta1
  }

  # after max_step steps, we still cannot get the qualified result
  cat("Did not converge after", max_step, " steps\n")
  return(NULL)
}


# steepest ascent method 2
# beta_init: the initial value
# alpha_init: the learning rate
# eps: the threshold of the gradient
# max_step: the maximum number of steps
# rate: the rate of change of alpha when g not increase

steepest_ascent_2 <- function(beta_init, alpha_init = 1, eps = 1e-10,
                              max_step = 1000, rate = 0.5) {
  beta0 <- beta_init
  alpha <- alpha_init
  for (i in 1:max_step){
    beta1 <- beta0 + alpha * diag(2) %*% dg(beta0)
    criterion <- distance(beta1, beta0)
    dg0 <- dg(beta0)
    while (g(beta1) < g(beta0)) {
      alpha <-  alpha * rate
      beta1 <- beta0 + alpha * diag(2) %*% dg0
    }
    if (criterion < eps) {
      cat("Converged after ", i, " steps\n")
      return(beta1)
    }
    beta0 <- beta1
  }

  # after max_step steps, we still cannot get the qualified result
  cat("Did not converge after", max_step, " steps\n")
  return(NULL)
}

########################## [ 2 b ] ###############################################
beta_init <- c(-0.2, 1)

steepest_ascent_1(beta_init)

steepest_ascent_2(beta_init)

########################## [ 2 c ] ###############################################
```

```r
# Use optim function (BFGS) to find the optimal parameters
result_bfgs <- optim(par = beta_init, g, dg, method = 'BFGS',
                     hessian = TRUE, control = list(fnscale = -1))


# Use optim function () to find the optimal parameters
result_nelder_mead <- optim(par = beta_init, g, dg, method = 'Nelder-Mead',
                            hessian = TRUE, control = list(fnscale = -1))
########################## [ 2 d ] ###############################################
x_new <- cbind(1, x)
data <- data.frame(x = x_new, y = y)
fit <- glm(y ~ x, family = binomial(link = 'logit'), data = data)
summary(fit)
```