CompStatLab2

Xuan Wang & Priyarani Patil

2023-11-07

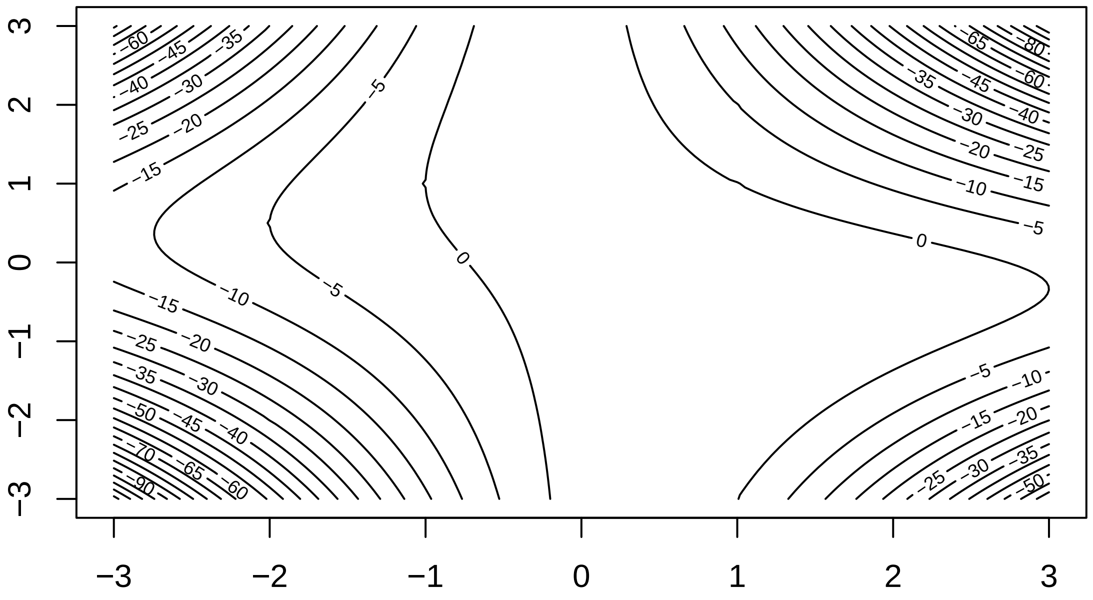
Computational Statistics 732A90 | Computer Lab 1

Question 1: Optimisation of a two-dimensional function

Consider the function $g(x, y) = -x^2 - x^2y^2 - 2xy + 2x + 2$ It is desired to determine the point (x, y), x, y [−3, 3], where the function is maximized.

a.Derive the gradient and the Hessian matrix in dependence of x, y. Produce a contour plot of the function g.

Answer: Please reference Appendix two_dimensional.R, and below are plot of function g.



b.Write an own algorithm based on the Newton method in order to find a local maximum of g

Answer:

Please reference Appendix two_dimensional.R

c.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 [−3, 3]?

Answer:

Please see below R results:

```
## ==========================================================

## 1) using starting points: (x,y) =(2,0), the newton method coverges to point:

##            [,1]
## [1,]  1.0000256
## [2,] -0.9999341

## the corresponding gradient is:

## [1]  2.926312e-05 -8.049134e-05

## the corresponding hessian matrix is:

##           [,1]      [,2]
## [1,] -3.999737  1.999839
## [2,]  1.999839 -2.000102

## This Hessian Matrix is negative definite.

## ==========================================================

## 2) using starting points: (x,y) =(-1,-2)

##              [,1]
## [1,] 1.166791e-11
## [2,] 1.000000e+00

## the corresponding gradient is:

## [1] -2.939771e-08 -2.333581e-11

## the corresponding hessian matrix is:

##      [,1]        [,2]
## [1,]   -4 -2.0000e+00
## [2,]   -2 -2.7228e-22

## This Hessian Matrix is neither positive nor negative definite.

## ==========================================================

## 3) using starting points: (x,y) =(0,1)

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

## the corresponding gradient is:

## [1] 0 0

## the corresponding hessian matrix is:

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

## This Hessian Matrix is neither positive nor negative definite.

## ==========================================================

## 4) using starting points: (x,y) =(1,-1)

##      [,1]
## [1,]    1
## [2,]   -1
## the corresponding gradient is:

## [1] 0 0

## the corresponding hessian matrix is:

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

## This Hessian Matrix is negative definite.
```

According to above R results for 4 different starting points, it's easily to get the conclusion:

1) when starting point is (2,0), the converging point is (1.0000256,-0.9999341) which is local maximum point since the Hessian matrix at that point is negative definite.
2) when starting point is (-1,-2), the converging point is (1.166791e-11,1.000000e+00) which is saddle point since the Hessian matrix at that point is neither positive or negative definite.
3) when starting point is (0,1), the converging point is (0,1) which is saddle point since the Hessian matrix at that point is neither positive or negative definite.
4) when starting point is (1,-1), the converging point is (1,-1) which is local maximum since the Hessian matrix at that point is negative definite.

d. What would be the advantages and disadvantages when you would run a steepest ascent algorithm instead of the Newton algorithm?

Answer:

Please see below R results for steepest ascent algorithm, compared with Newton algorithm,

Advantages:

1) Steepest ascent algorithm only requires the computation of first derivatives, while Newton requires the second derivaties as well;
2) Steepest ascent algorithm can compute more accurately when to get local maximum. For example, when starting point is (-1,-2), newton will converge to (1.166791e-11,1.000000e+00) which is a saddle point, while steepest ascent can converge to (0.99860093,-1.0031322) which is a local maximum. Besides, when starting point is nearly to a saddle point, steepest ascent can compute a local maximum point, while Newton will converge to a saddle point.
3) Steepest ascent algorithm can compute efficiently, it is well-used in machine learning.

Disadvantages:

1) Steepest ascent algorithm generally requires more iterations.
2) Steepest ascent algorithm is less prone to local minima but in case it tends to local minima, it has no noisy step hence it will not be able to come out of it.
3) Steepest ascent algorithm needs more memory to load the complete data into memory at once.

```
## 1) using starting points: (x,y) =(2,0), the steepest ascent method coverges to point:

## [1]  0.998619 -1.003172

## 2) using starting points: (x,y) =(-1,-2), the steepest ascent method coverges to point:

## [1]  0.9986093 -1.0031322

## 3) using starting points: (x,y) =(0,1), the steepest ascent method coverges to point:

## [1] 0 1
## 4) using starting points: (x,y) =(0.000099,0.9999), the steepest ascent method coverges to point:
## [1] 1.019799e-06 9.998010e-01
```

Question 2

a. Write a function for an ML-estimator for $(\beta0, \beta1)$ 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.

Answer:

Please reference Appendix ml_estimator.R

b. Compute the ML-estimator with the function from a. for the data (xi, yi) above. Use a stopping criterion such that you can trust five digits of both parameter estimates for $\beta0$ and $\beta1$. Use the starting value $(\beta0, \beta1)$ = (−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.

Answer:

Please see below R results. It can shows that when alpha using decreasing step size in each iteration, it takes less number of function and gradient evaluations, which means it could be more efficient. While the beta0 is less precise.

```
## =====Steepest ascent algorithm with constant alpha0 (alpha(t+1)=alpha0) when new iteration======
## $coefficients
##        beta0        beta1
## -0.007299539  1.254971983
##
## $counts
##     func_count gradient_count
##             42             21
## =====Steepest ascent algorithm with decreasing alpha (alpha(t+1)=alpha(t)) when new iteration======
## $coefficients
##        beta0        beta1
## -0.003158611  1.244742170
##
## $counts
##     func_count gradient_count
##             34             17
```

c. 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.

Answer:

Please see below R results. It shows that the results computed by both BFGS and Nelder-Mead algorithm are very close, it also quite close to the results computed by Steepest ascent except beta0 is a little smaller when using steepest ascent. Besides, when it refers to the number of function and gradient evaluations, BFGS performs more efficient with less iterations to call function and gradient, while Nelder-Mead requires more iterations.

algorithm	beta0	beta1	count_of_function	count_of_gradient
Steepest Ascent	-0.0072995	1.254972	42	21

algorithm	beta0	beta1	count_of_function	count_of_gradient
BFGS	-0.0093561	1.262813	12	8
Nelder-Mead	-0.0094234	1.262738	47	NA

```
## =====Using BFGS optimal get below results:======
```

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

```
## =====Using Nelder-Mead optimal get below results:======
```

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

d. Use the function glm in R to obtain an ML-solution and compare it with your results before.

Answer:

Please see below R results. It shows that glm function works same as BFGS, it computes the nearly same value for $\beta0$ and $\beta1$.

```
##
## Call:
## glm(formula = y ~ x, family = binomial(link = "logit"))
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.6780  -1.1734   0.7491   1.0251   1.1814
##
## 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
```

Appendix:

two_dimensional.R

```r
# function g to be maximized
g <- function(x,y) {
  return(-x^2 - x^2 * y^2 - 2 * x * y + 2 * x + 2)
}

# partial derivative x for function g
deriv_x <- function(x, y) {
  return(-2 * x - 2 * x * y^2 - 2 * y + 2)
}

# partial derivative y for function g
deriv_y <- function(x, y) {
  return(-2 * x^2 * y - 2 * x)
}

# gradient for function g
gradient <- function (x, y) {
  return(c(deriv_x(x,y), deriv_y(x,y)))
}

# second partial derivative x for function g
deriv_x_11 <- function(x, y) {
  return(-2 - 2 * y^2)
}

# second partial derivative x/y for function g
deriv_x_12 <- function(x, y) {
  return(-4 * x * y - 2)
}

# second partial derivative y for function g
deriv_y_22 <- function(x, y) {
  return(-2 * x^2)
}

# hessian matrix for function g
hessian_g <- function(x, y) {
  return(matrix(c(deriv_x_11(x,y), deriv_x_12(x,y), deriv_x_12(x,y), deriv_y_22(x,y)), nrow = 2, ncol =
}

# produce a contour plot
xgrid <- seq(-3,3, by=0.05)
ygrid <- seq(-3,3, by=0.05)
length_x <- length(xgrid)
length_y <- length(ygrid)
dxy <- length_x * length_y
gxy <- matrix(rep(NA,dxy), nrow = length_x)
for ( i in 1:length_x) {
  for ( j in 1:length_y) {
    gxy[i, j] <- g(xgrid[i], ygrid[j])
  }
}

mgxy <- matrix(gxy, nrow = length_x, ncol = length_y)
contour(xgrid, ygrid, mgxy, nlevels=40)

# newton function
newton_g <- function(x,y, eps=0.0001) {
  xt <- c(x,y)
  xt1 <- c(x,y) + 2

  while( t(xt1 -xt) %*% (xt1 -xt) > eps) {
    xt1 <- xt
    xt <- xt1 - solve(hessian_g(xt[1] , xt[2])) %*% gradient(xt[1],xt[2])
  }
  return(xt)
}
# check definiteness of a hessian matrix
check_definiteness <- function(matrix) {
  eigenvalues <- eigen(matrix)$values
  if (all(eigenvalues > 0)) {
    cat("This Hessian Matrix is positive definite.\n")
  } else if (all(eigenvalues < 0)) {
    cat("This Hessian Matrix is negative definite.\n")
  } else {
    cat("This Hessian Matrix is neither positive nor negative definite.\n")
  }
}

# using different starting points for newton method
cat("=========================================================")
converges_point1 <- newton_g(2,0)
cat("1) using starting points: (x,y) =(2,0), the newton method coverges to point: \n")
converges_point1

grad_point1 <- gradient(converges_point1[1],converges_point1[2])
cat("the corresponding gradient is: \n")
grad_point1

hessian_point1 <- hessian_g(converges_point1[1],converges_point1[2])
cat("the corresponding hessian matrix is: \n")
```

```r
hessian_point1

check_definiteness(hessian_point1)

cat("========================================================")

converges_point2 <- newton_g(-1,-2)
cat("2) using starting points: (x,y) =(-1,-2) \n")
converges_point2

grad_point2 <- gradient(converges_point2[1],converges_point2[2])
cat("the corresponding gradient is: \n")
grad_point2

hessian_point2 <- hessian_g(converges_point2[1],converges_point2[2])
cat("the corresponding hessian matrix is: \n")
hessian_point2

check_definiteness(hessian_point2)

cat("========================================================")

converges_point3 <- newton_g(0,1)
cat("3) using starting points: (x,y) =(0,1) \n")
converges_point3

grad_point3 <- gradient(converges_point3[1],converges_point3[2])
cat("the corresponding gradient is: \n")
grad_point3

hessian_point3 <- hessian_g(converges_point3[1],converges_point3[2])
cat("the corresponding hessian matrix is: \n")
hessian_point3

check_definiteness(hessian_point3)

cat("========================================================")

converges_point4 <- newton_g(1,-1)
cat("4) using starting points: (x,y) =(1,-1) \n")
converges_point4

grad_point4 <- gradient(converges_point4[1],converges_point4[2])
cat("the corresponding gradient is: \n")
grad_point4

hessian_point4 <- hessian_g(converges_point4[1],converges_point4[2])
cat("the corresponding hessian matrix is: \n")
hessian_point4

check_definiteness(hessian_point4)

# Steepest ascent algorithm for g function
```

```r
steepest_ascent_g <- function(x, y, tol=1e-5, alpha0=1) {
  xt <- c(x,y)
  conv <- 999
  while (conv > tol) {
    alpha <- alpha0
    xt1 <- xt
    xt <- xt1 + alpha * gradient(xt1[1],xt1[2])
    while (g(xt[1],xt[2]) < g(xt1[1],xt1[2])) {
      alpha <- alpha/2
      xt <- xt1 + alpha * gradient(xt1[1],xt1[2])
    }
    conv <- sum((xt - xt1) * (xt - xt1))
  }
  return(xt)
}

cat("1) using starting points: (x,y) =(2,0), the steepest ascent method coverges to point: \n")
steepest_ascent_g(2,0)
cat("2) using starting points: (x,y) =(-1,-2), the steepest ascent method coverges to point: \n")
steepest_ascent_g(-1,-2)
cat("3) using starting points: (x,y) =(0,1), the steepest ascent method coverges to point: \n")
steepest_ascent_g(0,1)
cat("4) using starting points: (x,y) =(0.000099,0.9999), the steepest ascent method coverges to point: 
steepest_ascent_g(0.000099,0.9999)
```

ml_estimator.R

```r
# Define the log-likelihood function
log_likelihood <- function(beta, x, y) {
  p <- 1 / (1 + exp(-beta[1] - beta[2]*x))
  sum(y * log(p) + (1 - y) * log(1 - p))
}

# Define the gradient of the log-likelihood function
gradient <- function(beta, x, y) {
  p <- 1 / (1 + exp(-beta[1] - beta[2]*x))
  db0 <- sum(y - p)
  db1 <- sum((y - p) * x)
  c(db0, db1)
}

# Steepest ascent function with constant alpha0 when new iteration
steepest_ascent_constant <- function(beta_start, x, y, tol=1e-5, alpha0=1) {
  beta <- beta_start
  log_likelihood_count <- 0
  gradient_count <- 0
  conv <- 999
  while (conv > tol) {
    alpha <- alpha0
    beta_old <- beta
    beta <- beta_old + alpha * gradient(beta_old, x, y)
    gradient_count <- gradient_count + 1
    log_likelihood_count <- log_likelihood_count + 2
    while (log_likelihood(beta, x , y) < log_likelihood(beta_old, x , y)) {
      alpha <- alpha/2
      beta <- beta_old + alpha * gradient(beta_old, x , y)
      gradient_count <- gradient_count + 1
      log_likelihood_count <- log_likelihood_count + 2
    }
    conv <- sum((beta - beta_old) * (beta - beta_old))
  }
  result <- list(coefficients=c(beta0=beta[1],beta1=beta[2]), counts=c(func_count=log_likelihood_count,
  return(result)
}


# Steepest ascent function with decreasing alpha when new iteration
steepest_ascent_decrease <- function(beta_start, x, y, tol=1e-5, alpha0=1) {
  beta <- beta_start
  log_likelihood_count <- 0
  gradient_count <- 0
  conv <- 999
  alpha <- alpha0
  while (conv > tol) {
    beta_old <- beta
    beta <- beta_old + alpha * gradient(beta_old, x, y)
    gradient_count <- gradient_count + 1
    log_likelihood_count <- log_likelihood_count + 2
    while (log_likelihood(beta, x , y) < log_likelihood(beta_old, x , y)) {
      alpha <- alpha/2
      beta <- beta_old + alpha * gradient(beta_old, x , y)
      gradient_count <- gradient_count + 1
      log_likelihood_count <- log_likelihood_count + 2
    }
    conv <- sum((beta - beta_old) * (beta - beta_old))
  }
  result <- list(coefficients=c(beta0=beta[1],beta1=beta[2]), counts=c(func_count=log_likelihood_count,
  return(result)
}
# drug and placebo data
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)

# Initial values for beta
beta_start <- c(-0.2, 1)

# Compute the ML estimator
steepest_ascent_constant_result <- steepest_ascent_constant(beta_start, x, y)

steepest_ascent_decrease_result <- steepest_ascent_decrease(beta_start, x, y)

# Print the results
cat("=====Steepest ascent algorithm with constant alpha0 (alpha(t+1)=alpha0) when new iteration====== \
print(steepest_ascent_constant_result)
cat("=====Steepest ascent algorithm with decreasing alpha (alpha(t+1)=alpha(t)) when new iteration=====
print(steepest_ascent_decrease_result)
```

```
# BFGS
result_bfgs <- optim(beta_start, log_likelihood, gradient, x=x, y=y, method="BFGS", control = list(fnsca
cat("=====Using BFGS optimal get below results:====== \n")
print(result_bfgs)

# Nelder-Mead
result_nelder <- optim(beta_start, log_likelihood, gradient, x=x, y=y, method="Nelder-Mead", control = l
cat("=====Using Nelder-Mead optimal get below results:====== \n")
print(result_nelder)

# Fit the model
fit <- glm(y ~ x, family=binomial(link="logit"))

# Print the results
print(summary(fit))
```