732A90 Computational Statistics

Lab 2

Johannes Hedström & Mikael Montén

STIMA
Institutionen för datavetenskap
Linköpings universitet

2024-01-25

Contents

1 Question 1: Optimisation of a two-dimensional function

Consider the function

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

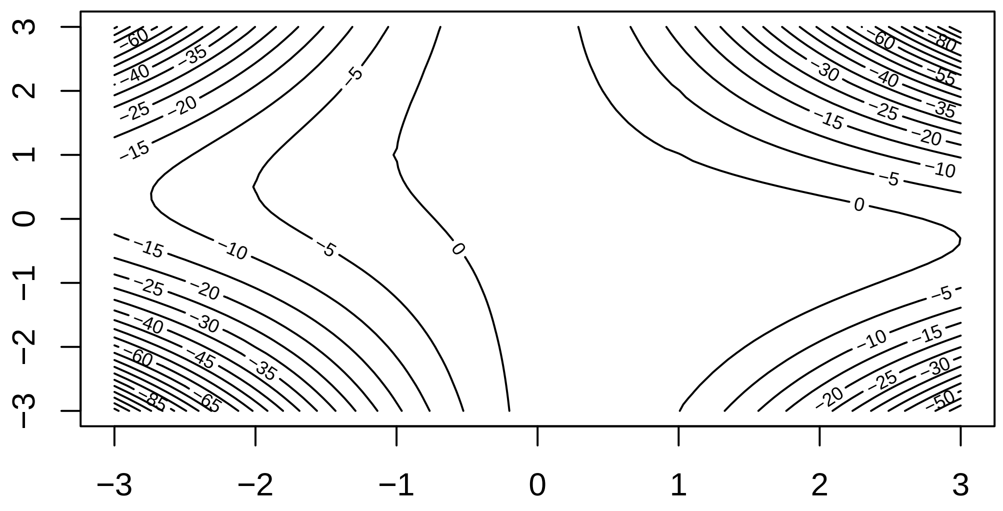
It is desired to determine the point $(x, y), x, y \in [-3, 3]$, where the function is maximized.

1.0.1 a)

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

```r
# Create function
g <- function(x, y){
  -x^2-x^2*y^2-2*x*y+2*x+2
}

# Derive the gradient
deriv_y <- D(expression(-x^2-x^2*y^2-2*x*y+2*x+2), "y")
deriv_x <- D(expression(-x^2-x^2*y^2-2*x*y+2*x+2), "x")

# Prepare contour plot
xgrid <- seq(-3,3,0.1)
ygrid <- seq(-3,3,0.1)
dx1 <- length(xgrid)
dy1 <- length(ygrid)
dx <- dx1*dy1

gx <- matrix(rep(NA, dx), nrow = dx1)
for(i in 1:dx1){
  for(j in 1:dy1){
    x <- xgrid[i]
    y <- ygrid[j]
    gx[i,j] <- g(x,y)
  }
}
mgx <- matrix(gx, nrow = dx1, ncol = dy1)


# Second derivative
deriv2_x <- D(expression(2 - (2 * x + 2 * x * y^2 + 2 * y)), "x")
deriv2_y <- D(expression(-(x^2 * (2 * y) + 2 * x)), "y")
deriv2_xy <- D(expression(2 - (2 * x + 2 * x * y^2 + 2 * y)), "y")

Hessian_df <- data.frame(deparse(deriv2_x), deparse(deriv2_xy), deparse(deriv2_y))
colnames(Hessian_df) <- c("Second derivative of x", "Second derivative of xy", "Second derivative of y")

derivative_df <- data.frame(deparse(deriv_x), deparse(deriv_y))
colnames(derivative_df) <- c("Derivative of x", "Derivative of y")
```

```
# Contour plot
contour(xgrid, ygrid, mgx, nlevels=20)
```



```
knitr::kable(derivative_df, caption = "Derivatives of x, y")
```

Table 1: Derivatives of x, y

Derivative of x	Derivative of y
2 - (2 * x + 2 * x * y^2 + 2 * y)	-(x^2 * (2 * y) + 2 * x)

```
knitr::kable(Hessian_df, caption = "Hessian matrix second derivatives")
```

Table 2: Hessian matrix second derivatives

Second derivative of x	Second derivative of xy	Second derivative of y
-(2 + 2 * y^2)	-(2 * x * (2 * y) + 2)	-(x^2 * 2)

Table 1 shows the derivatives of x and y.

Table 2 shows the second derivatives with regard to x, y and x*y. These are the elements of the Hessian matrix, which is denoted as $H = g''(x,y) = \begin{bmatrix} \delta^2 g/\delta x^2\ (x,y) & \delta^2 g/\delta x \delta y\ (x,y) \\ \delta^2 g/\delta x \delta y\ (x,y) & \delta^2 g/\delta y^2\ (x,y) \end{bmatrix}$

Which in our case is $H = \begin{bmatrix} -(2+2*y^2) & -(2*x*(2*y)+2) \\ -(2*x*(2*y)+2) & -(x^2*2) \end{bmatrix}$.

The contour plot shows the bivariate distribution and it is very hard to interpret where a potential maximum would be geometrically, which showcases the need to optimize the functions numerically.

1.0.2 b)

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

```
# Derivative of function g
dg <- function(x, y){
  c(2 - (2 * x + 2 * x * y^2 + 2 * y), -(x^2 * (2 * y) + 2 * x))
}

# Second derivate of function g
d2g <- function(x, y){
  matrix(c(-(2 + 2 * y^2), -(2 * x * (2 * y) + 2), -(2 * x * (2 * y) + 2), -(x^2 * 2)), ncol = 2)
}

# Newton method for bivariate case
multi_newton <- function(x0, y0, eps = .0001){
  x <- x0
  y <- y0
  xt <- c(x0,y0)
  xt1 <- xt+2
  while(abs(xt[1]-xt1[1]) && abs(xt[2]-xt1[2]) > eps){
    xt1 <- xt
    xt <- xt1 - solve(d2g(xt1[1],xt1[2])) %*% dg(xt1[1],xt1[2])
  }
  xt
}
```

1.0.3 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 $x, y \in [-3,-3]$?

```
# First pair to check
a <- multi_newton(2,0)
knitr::kable(data.frame("Derivatives" = dg(a[1],a[2]), "Eigen" = eigen(d2g(a[1],a[2]))$values),
             digits = 10)
```

Derivatives	Eigen
-1.9e-09	-0.763932
-5.4e-09	-5.236068

The derivative is approximately 0 for both estimates, meaning the algorithm converges to a point for both that either are maximum/minimum. Since the Hessian matrix Eigen values are definite negative, the pair it converged to is a maximum point.

```r
# Second pair
b <- multi_newton(-1,-2)
knitr::kable(data.frame("Derivatives" = dg(b[1],b[2]), "Eigen" = eigen(d2g(b[1],b[2]))$values),
             digits = 23)
```

Derivatives	Eigen
0.0e+00	0.8284271
-2.7e-22	-4.8284271

Again the algorithm converges to a pair that has derivative approximately 0, meaning it also is a potential maximum or minimum. However, since the Hessian matrix Eigen values indefinite - meaning one positive and one negative - the converged pair is a saddle point.

```r
# Third pair
c <- multi_newton(0,1)
knitr::kable(data.frame("Derivatives" = dg(c[1],c[2]), "Eigen" = eigen(d2g(c[1],c[2]))$values),
             digits = 23)
```

Derivatives	Eigen
0	0.8284271
0	-4.8284271

Derivative is exactly equal to zero for both x and y, however the Hessian matrix is indefinitive so the pair is a saddle point.

```r
# Chosen pair
d <- multi_newton(3, -3)
knitr::kable(data.frame("Derivatives" = dg(d[1],d[2]), "Eigen" = eigen(d2g(d[1],d[2]))$values),
             digits = 23)
```

Derivatives	Eigen
4.440892e-16	-0.763932
-9.769963e-15	-5.236068

Again, an approximately zero derivative. This starting pair also converges to a maximum as the Eigen value is negative.

The 4 different pairs resulted in 2 of them converging to a maximum. The value of the function g for these two pars is 4 which we show below.

```
g(a[1],a[2]) # value of function for our proposed maximum for pair a
```

```
## [1] 4
```

```
g(d[1],d[2]) # value of function for our proposed maximum for pair a
```

```
## [1] 4
```

The functions has the same value which means the different pairs have converged to the same points. To check if the resulting pair is a global maximum, we look in the defined matrix above.

```
max(mgx)
```

```
## [1] 4
```

The function value 4 is the highest in the matrix of function values, meaning it is indeed a global maximum.

```
table(mgx = 4)
```

```
## mgx
## 4
## 1
```

To confirm it really is a global maximum we see the amount of cells that has the value 4 in the matrix, and with it being a single observation our result is confirmed.

1.0.4 d)

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

Running a Newton algorithm is more time efficient, however it is very dependent on start values, g(x) is not guaranteed to always increase and you need to calculate the Hessian in each iteration which can be complicated. Running a Steepest ascent might take longer, but it's less prone to human error due to not having to calculate the Hessian, forcing improvements in g(x), and it's guaranteed to always return a global or local maximum, whereas the Newton algorithm also can return minimums. However the Steepest ascent algorithm doesn't relay any information regarding the curvature used.

2 Question 2

Three doses $(0.1, 0.3, 0.9\ g)$ of a drug and placebo $(0\ g)$ are tested in a study. A dose-dependent event is recorded afterwards. The data of $n = 10$ subjects is shown below; x_i is the dose in gram; $y_i = 1$ if the event occurred, $y_i = 0$ otherwise.

```r
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)
n <- 10
```

You should fit a simple logistic regression

$$p(x) = P(Y = 1|x) = \frac{1}{1 + exp(-\beta_0 - \beta_1 x)}$$

to the data, i.e. estimate β_0 and β_1. One can show that the log likelihood is

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

where $\mathbf{b} = (\beta_0, \beta_1)^T$ and the gradient is

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

2.0.1 a)

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 counter the number of function and gradient evaluations.

```r
# The loglikelihood
g <- function(b){
  value <- 0
  for(i in 1:n){
    value <- value + sum(y[i] * log((1+exp(-b[1]-b[2]*x[i]))^(-1)) + (1-y[i])*log((1-(1+exp(-b[1]-b[2]*x
  }
  return(value)
}

# The gradient
dg <- function(b){
  gradient <- c(0,0)
  for(i in 1:n){
    sigm <- 1/(1 + exp(-b[1] - b[2]*x[i]))
    gradient <- gradient + (y[i] - sigm) * c(1,x[i])
  }
  return(gradient)
}

# Steepest ascent algorithm
steepestasc <- function(x0, eps=1e-8, alpha0=1, return = TRUE)
{
  xt   <- x0
  conv <- 999
  g_eval <- 0
  dg_eval <- 0

  alpha <- alpha0
  while(conv>eps)
  {
    if(return == TRUE){
      alpha <- alpha0 # if return is set to FALSE, alpha will never go back to 1
    }
    xt1   <- xt
    xt    <- xt1 + alpha*dg(xt1)

    g_eval <- dg_eval + 1
    while (g(xt)<g(xt1))
    {
      alpha <- alpha/2
      xt    <- xt1 + alpha*dg(xt1)

      dg_eval <- g_eval + 1
    }
    conv <- sum((xt-xt1)*(xt-xt1))

  }
  result <- list("Function evaluations" = as.character(g_eval),
                 "Derivative evaluations" = as.character(dg_eval),
                 "beta_0 estimation" = xt[1], "beta_1 estimation" = xt[2])
  result
}
```

2.0.2 b)

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 evlauation done until convergence

```r
estimator <- data.frame(
  g(b = c(-0.2,1)), # set the starting values in the created function
  dg(b = c(-0.2,1))[1], # set the starting values in the created derivative and index out the first esti
  dg(b = c(-0.2,1))[2]  # set the starting values in the created derivative and index out the second est
)
colnames(estimator) <- c("Function value","Derivate of x","Derivate of y")
knitr::kable(estimator, caption = "Function and derivative values")
```

Table 7: Function and derivative values

Function value	Derivate of x	Derivate of y
-6.581561	0.6449387	0.2859013

```r
steepest <- data.frame(
  as.numeric(unlist(steepestasc(x0 = c(-0.2,1))))
)

colnames(steepest) <- "Steepest ascent result"
knitr::kable(steepest, caption = "Steepest ascent evaluations and estimations")
```

Table 8: Steepest ascent evaluations and estimations

Steepest ascent result
25.0000000
26.0000000
-0.0093162
1.2626558

```r
steepest2 <- data.frame(
  as.numeric(unlist(steepestasc(x0 = c(-0.2,1), return = FALSE)))
)

colnames(steepest2) <- "Steepest ascent result"
knitr::kable(steepest2,
caption = "Steepest ascent evaluations and estimations with different backtracking than previous")
```

Table 9: Steepest ascent evaluations and estimations with different backtracking than previous

Steepest ascent result
3.0000000
2.0000000
-0.0091696
1.2622679

Comparing Table 8 and Table 9 shows great differences in the number of evaluations needed to converge to a nearly identical point estimate for both coefficients. The calculations between the two differ in how the steepest algorithm considers the alpha parameter, where for table 8 the algorithm assigns alpha to 1 at the start of the loop and subsequently halves the value when the conditions are met and for table 9 alpha is only set to 1 at the beginning of the function. This results in an algorithm that has function evaluations dropping from 25 to 3 and derivative evaluations from 26 to 2. The first variant that has to restart the halving of alpha is fast for an optimization problem that inherently requires a lot of iterations due to being able to find the right direction efficiently, but for a problem where large steps easily can be found the second variant is a lot faster.

2.0.3 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**.*

```r
# optim function in r to find maximum of our log likelihood function(fnscale=-1 converts to maximize)
result1 <- optim(par = c(-0.2,1), fn = g,gr=dg,  method = "BFGS", control = list(fnscale=-1))

result2 <- optim(par = c(-0.2,1), fn = g,gr=dg,  method = "Nelder-Mead", control = list(fnscale=-1))

# making a df from the results
BFGS <- unlist(c(result1[1],result1[2],result1[3]))

Nelder_Mead <- unlist(c(result2[1],result2[2],result2[3]))

df_res <- as.data.frame(rbind(BFGS,Nelder_Mead))

# chaning the colnames
colnames(df_res) <- c('Intercept', 'Slope', 'Log likelihood', 'Function eval', 'Gradient eval')

knitr::kable(df_res, caption='Difference between BFGS and Nelder-Mead algorithms')
```

Table 10: Difference between BFGS and Nelder-Mead algorithms

	Intercept	Slope	Log likelihood	Function eval	Gradient eval
BFGS	-0.0093561	1.262813	-6.484279	12	8
Nelder_Mead	-0.0094234	1.262738	-6.484279	47	NA

```
comparison <- as.data.frame(cbind(steepest,steepest2, rbind(df_res[,4], df_res[,5],
                                             df_res[,1], df_res[,2])))
colnames(comparison) <- c("SA1", "SA2", "BFGS", "NM")
rownames(comparison) <- c("Function eval", "Derivative eval", "Intercept", "Slope")
comparison <- t(comparison)
knitr::kable(comparison, digits = 5, caption = "Comparison between different algorithms")
```

Table 11: Comparison between different algorithms

	Function eval	Derivative eval	Intercept	Slope
SA1	25	26	-0.00932	1.26266
SA2	3	2	-0.00917	1.26227
BFGS	12	8	-0.00936	1.26281
NM	47	NA	-0.00942	1.26274

Table 11 shows how the four different algorithms converge. SA1 is the first steepest ascent algorithm that loops alpha to 1 in the iterations and SA2 is the second where alpha keeps decreasing. The second steepest ascent algoritm is the one with the clearly lowest amount of function evaluations, and Nelder-Mead is the algorithm with the highest amount of evalutions, with the two other algorithms spread between them. The same relation holds true for derivative evaluations as well, except the Nelder-Mead not evaluating the derivative and therefore returning NA.

Regarding the estimations down to 5 digits, the algorithms produce similar slopes and intercepts. The biggest intercept difference is between SA2 and NM with a change of $-0.00917-(-0.00942) = 0.00025$, and the biggest slope difference is between SA2 and BFGS with a difference of $1.26227 - 1.26281 = -0.00054$ which both are very slight differences.

2.0.4 d)

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

```
eps_8 <- unlist(steepestasc( c(-0.2,1))[3:4])
eps_10 <- unlist(steepestasc( c(-0.2,1), eps=1e-10)[3:4])
eps_12 <- unlist(steepestasc( c(-0.2,1), eps=1e-12)[3:4])
eps_16 <- unlist(steepestasc( c(-0.2,1), eps=1e-16)[3:4])


GLM <-glm(y~x, family = 'binomial')$coefficients

df <- data.frame(rbind(GLM,eps_8,eps_10,eps_12,eps_16))
colnames(df) <- c('Intercept', 'Slope')
knitr::kable(df, caption = "Comparison of different SA1 criterion vs GLM")
```

Table 12: Comparison of different SA1 criterion vs GLM

	Intercept	Slope
GLM	-0.0093599	1.262823
eps_8	-0.0093162	1.262656
eps_10	-0.0093510	1.262807
eps_12	-0.0093591	1.262822
eps_16	-0.0093598	1.262823

Table 12 shows the difference in estimated intercept and slope depending on how small the stopping criterion for the algorithm is chosen. It's easy to see that the smaller of a criterion you have, the more accurate to the GLM solution the algorithmic approxmiation will be, here shown by eps_16 that has an identical slope and only the smallest difference in intercept.