Homework on Newton's methods

Zhuodiao Kuang(zk2275)

Problem 1:

Design a Golden-Search algorithm to find the MLE in the Example 1.2 (see the lecture notes) and implement it into **R**.

Answer: your answer starts here...

Example 1.2 in the lecture notes is about finding the Maximum Likelihood Estimation (MLE) for a trinomial distribution where the probabilities are determined by a single parameter, θ. The optimization problem involves maximizing a likelihood function expressed in terms of θ. The likelihood function is given by the sum of $(x_1 \log(\theta + 2) + x_2 \log(1 - \theta) + x_3 \log(\theta)$, with θ constrained between 0 and 1.

```
set.seed(229)
# Define the likelihood function for the trinomial distribution
likelihood_function <- function(theta, x1, x2, x3) {
  term1 <- x1 * log(theta + 2)
  term2 <- x2 * log(1 - theta)
  term3 <- x3 * log(theta)
  L<-sum(term1 + term2 + term3)
  return(L)
}

# Golden-Search algorithm
golden_search <- function(f, lower, upper, tol = 0.0001) {
  gconstant <-   (sqrt(5) - 1)/2
  c <- upper - (upper - lower) * gconstant
  d <- lower + (upper - lower) * gconstant
  while (abs(c - d) > tol) {
    if (f(c) < f(d)) {
      lower <- c
    } else {
      upper <- d
    }
    c <- upper - (upper - lower) * gconstant
    d <- lower + (upper - lower) * gconstant
  }
  return((lower + upper) / 2)
}

# Example usage
# Generate Xdata
```

```r
n = 100
theta <- 0.2
Xdata <- rmultinom(n, 40, c((2 + theta)/4, (1 - theta)/2, theta/4))


x1 <- Xdata[1,]
x2 <- Xdata[2,]
x3 <- Xdata[3,]
# Find the MLE of theta
theta_mle <- golden_search(function(theta) likelihood_function(theta, x1, x2, x3), 0, 1)
print(theta_mle)
```

```
## [1] 0.1936355
```

Problem 2:

Design an optmization algorithm to find the minimum of the continuously differentiable function

$$f(x) = -e^{-x}\sin(x)$$

on the closed interval $[0, 1.5]$. Write out your algorithm and implement it into **R**.

Answer: your answer starts here...

Algorithm Steps:

Using Newton's method: - For each iteration, compute $f'(x)$ and $f''(x)$ at the current point x. - Update the guess using a modified Newton step: $x_{new} = x - \frac{f'(x)}{f''(x)}$. - Check for convergence: if $|x_{new} - x| < \epsilon$, then stop and return x_{new} as the minimum. - If the maximum number of iterations is reached without convergence, consider adjusting the initial guess or the damping factor.

```r
f <- function(x) {
  -exp(-x) * sin(x)
}

f_deriv <- function(x) {
  exp(-x) * (sin(x) - cos(x))
}

f_double_deriv <- function(x) {
  2 * cos(x)*exp(-x)
}

newton_method <- function(f_deriv, f_double_deriv,
                          start_point, tol = 1e-5, max_iter = 100) {
  x_current <- start_point
  for (i in 1:max_iter) {
    x_next <- x_current - f_deriv(x_current) / f_double_deriv(x_current)
     print(x_next )
    if (abs(x_next - x_current) < tol) {
```

```
      if (x_next >= 0 && x_next <= 1.5) {

        return(x_next)
      } else {
        break # If outside the interval, break and return NA
      }
    }
    x_current <- x_next
  }
  return(NA) # Return NA if no convergence within the interval or max iterations reached
}

# Choose an initial guess within the interval [0, 1.5]
start_point <- 0.5

# Apply the Newton Method
solution <- newton_method(f_deriv, f_double_deriv, start_point)
```

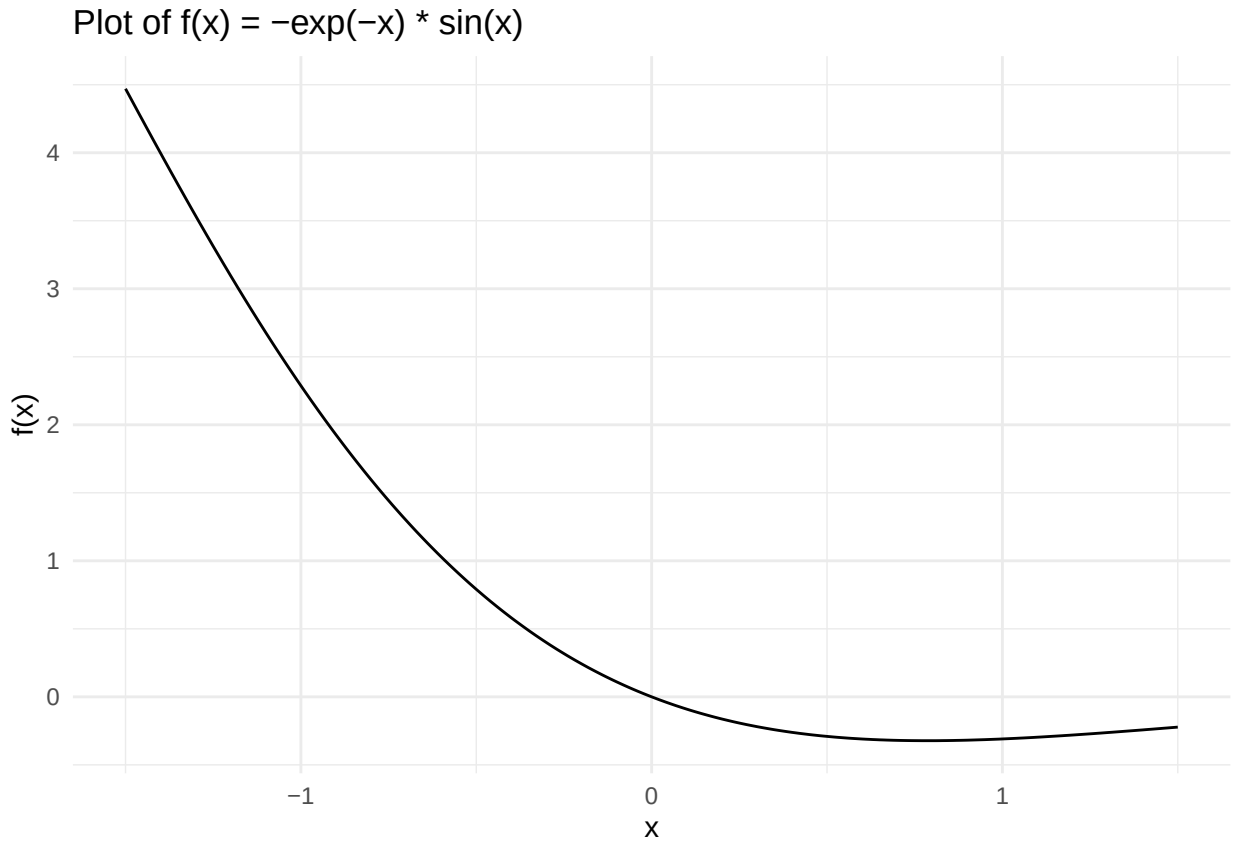
```
## [1] 0.7268488
## [1] 0.7822195
## [1] 0.7853881
## [1] 0.7853982
## [1] 0.7853982
```

```
# Display the solution
if (!is.na(solution)) {
  cat("The minimum is located at x =", solution, "\n")
  cat("The minimum value of the function is f(x) =", f(solution), "\n")
} else {
  cat("The algorithm did not converge to a solution within the interval.\n")
}
```

```
## The minimum is located at x = 0.7853982
## The minimum value of the function is f(x) = -0.3223969
```

```
# Define the function f(x) = -exp(-x) * sin(x)
f <- function(x) {
  -exp(-x) * sin(x)
}

# Create a sequence of x values from 0 to 1.5
x_values <- seq(-1.5, 1.5, by = 0.01)

# Calculate y values using the defined function
y_values <- f(x_values)

# Create a data frame for plotting
data <- data.frame(x = x_values, y = y_values)

# Plot the function using ggplot2
ggplot(data, aes(x = x, y = y)) +
  geom_line() +
```

```
  theme_minimal() +
  ggtitle("Plot of f(x) = -exp(-x) * sin(x)") +
  xlab("x") +
  ylab("f(x)")
```



Problem 3

The Poisson distribution is often used to model "count'' data — e.g., the number of events in a given time period.
The Poisson regression model states that

$$Y_i \sim \text{Poisson}(\lambda_i),$$

where

$$\log \lambda_i = \alpha + \beta x_i$$

for some explanatory variable x_i. The question is how to estimate α and β given a set of independent data $(x_1, Y_1), (x_2, Y_2), \ldots, (x_n, Y_n)$.

1. Modify the Newton-Raphson function from the class notes to include a step-halving step.
2. Further modify this function to ensure that the direction of the step is an ascent direction. (If it is not, the program should take appropriate action.)
3. Write code to apply the resulting modified Newton-Raphson function to compute maximum likelihood estimates for α and β in the Poisson regression setting.

The Poisson distribution is given by

$$P(Y = y) = \frac{\lambda^y e^{-\lambda}}{y!}$$

for $\lambda > 0$.

Answer: your answer starts here...

To address the given tasks, we'll modify the Newton-Raphson method for Poisson regression to include a step-halving step and ensure the step is in an ascent direction for estimating α and β.

The likelihood function for a set of n independent observations given this model is:

$$L(\alpha, \beta | y, x) = \prod_{i=1}^{n} \frac{e^{-\lambda_i} \lambda_i^{y_i}}{y_i!}$$

Taking the logarithm of the likelihood function gives the log-likelihood:

$$\ell(\alpha, \beta | y, x) = \sum_{i=1}^{n} (-\lambda_i + y_i \log(\lambda_i) - \log(y_i!))$$

Substituting $\lambda_i = e^{\alpha + \beta x_i}$ into the log-likelihood gives:

$$\ell(\alpha, \beta | y, x) = \sum_{i=1}^{n} (-e^{\alpha + \beta x_i} + y_i(\alpha + \beta x_i) - \log(y_i!))$$

1. **Include Step-Halving**: To ensure convergence, we'll incorporate a step-halving procedure. If an iteration does not decrease the log-likelihood, we reduce the step size by half and reevaluate.
2. **Ensure Ascent Direction**: Before updating the parameters, we'll check if the step increases the log-likelihood. If not, we adjust the direction or reduce the step size.

```
set.seed(229) # For reproducibility

# Generate  data
n <- 1000
x <- runif(n, 0, 10)
alpha <- 0.5 # True alpha
beta <- 0.1 # True beta

# Calculate lambda for the Poisson distribution
lambda <- exp(alpha + beta * x)
Y <- rpois(n, lambda)

# Check the first few values
head(x)
```

```
## [1] 7.887599 5.782350 4.363164 7.602662 4.431796 2.349943
```

```r
head(Y)
```

```
## [1] 5 0 4 7 3 1
```

```r
# Log-likelihood function for Poisson regression
log_likelihood <- function(alpha, beta, x, Y) {
  lambda <- exp(alpha + beta * x)
  sum(Y * log(lambda) - lambda)
}

# Gradient of the log-likelihood
gradient <- function(alpha, beta, x, Y) {
  lambda <- exp(alpha + beta * x)
  grad_alpha <- sum(Y - lambda)
  grad_beta <- sum((Y - lambda) * x)
  return(c(grad_alpha, grad_beta))
}

# Hessian matrix of the log-likelihood
hessian <- function(alpha, beta, x, Y) {
  lambda <- exp(alpha + beta * x)
  hess <- matrix(c(-sum(lambda), -sum(lambda * x),
                   -sum(lambda * x), -sum(lambda * x^2)), nrow=2)
  return(hess)
}

# Newton-Raphson method with step-halving and ascent direction check
newton_raphson <- function(x, Y, alpha_init, beta_init, tol = 1e-6, max_iter = 100) {
  alpha <- alpha_init
  beta <- beta_init
  for (i in 1:max_iter) {
    grad <- gradient(alpha, beta, x, Y)
    hess <- hessian(alpha, beta, x, Y)
    update <- solve(hess) %*% grad

     # Double check if the direction is an ascent direction
    if (t(grad) %*% update > 0) {
      cat("Iteration", i, ": Direction is not an ascent direction. Adjusting step...\n")
      update <- -update
    }

    # Step-halving
    step_size <- 1
    new_alpha <- alpha - step_size * update[1]
    new_beta <- beta - step_size * update[2]

    # Ensure Ascent Direction
    while (log_likelihood(new_alpha, new_beta, x, Y) < log_likelihood(alpha, beta, x, Y)) {
      # research for directions
      step_size <- step_size / 2
      new_alpha <- alpha - step_size * update[1]
      new_beta <- beta - step_size * update[2]
      if (step_size < tol) break
```

```
    }

    # Check for convergence
    if (sqrt(sum(update^2)) < tol) {
      cat("Converged in", i, "iterations\n")
      break
    }

    alpha <- new_alpha
    beta <- new_beta
  }
  return(c(alpha, beta))
}


alpha_init <- beta_init <- 0
estimates <- newton_raphson(x, Y, alpha_init, beta_init)
```

```
## Converged in 5 iterations
```

```
cat("True alpha:", alpha, "\nTrue beta:", beta, "\n")
```

```
## True alpha: 0.5
## True beta: 0.1
```

```
cat("Estimated alpha:", estimates[1], "\nEstimated beta:", estimates[2], "\n")
```

```
## Estimated alpha: 0.5143557
## Estimated beta: 0.09938042
```

Problem 4: Breast Cancer Diagnosis

Background

The dataset "breast-cancer.csv" consists of 569 rows and 33 columns. The first column, labeled "ID", contains individual breast tissue images. The second column, labeled "Diagnosis", identifies whether the image is from a cancerous or benign tissue (M=malignant, B=benign). There are 357 benign and 212 malignant cases. The other 30 columns represent the mean, standard deviation, and largest values of the distributions of 10 features computed for the cell nuclei.

- radius (mean of distances from center to points on the perimeter)
- texture (standard deviation of gray-scale values)
- perimeter
- area
- smoothness (local variation in radius lengths)
- compactness (perimeter2 / area - 1.0)