Fourth assignment$_4$ Course

Mathematical and Computational Statistics

Achladianakis Minas

A report presented for the Course Mathematical and Computational Statistics

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ
UNIVERSITY OF CRETE

Department of Applied Mathematics
University of Crete

Greece, December 15, 2023

Abstract

This assignment is part of my evaluation for the course: Mathematical and Computational Statistics.

List of Figures

Assignment 4

4.1 Part1

The derivatives of log-likelihood of the Cauchy, which are required for the Newton-Raphson procedure to work, are depicted in the attached Thesis (Chapter VII) [Haa69]. For the logistic distribution the corresponding expressions are shown here.

(a) Derive the above-mentioned expressions.

(b) The attached csv contains 1000 independent random observations from the Cauchy and the Logistic distributions. Estimate their location and scale parameters using the Newton-Raphson procedure

(c) Compute confidence intervals for your estimated parameters using the inverse of the Fisher information matrix. Compare these intervals against the ones derived from bootstrap with $B = 200$ samples.

(d) Perform a Monte-Carlo experiment by computing 100 times the above estimators, each time using a different random sample of size N = 100. Report parameter estimates and quantile-based confidence intervals and discuss your findings relative to the previous, bootstrap-based intervals that used the whole sample.

4.1.1 a. The derivatives of log-likelihood of the Cauchy and the logistic distribution

The Cauchy

The Likelihood

$$L(x; m, b) = \prod_{i=1}^{n} \frac{1}{\pi b \left(1 + \frac{(x_i - m)}{b^2}^2\right)}$$

The log-likelihood

$$\log L = -n \log \pi - n \log b - \sum_{i=1}^{n} \log(1 + \frac{(x_i - m)^2}{b^2})$$

For the analytical calculations see Appendix A.

1st Order.

$$\frac{\partial \log L}{\partial m} = 2 \sum_{i=1}^{n} \frac{(x_i - m)}{b^2 + (x_i - m)^2}$$

$$\frac{\partial \log L}{\partial b} = \frac{1}{b} \left[-n + 2 \sum_{i=1}^{n} \frac{(x_i - m)^2}{b^2 + (x_i - m)^2} \right]$$

2nd Order.

$$\frac{\partial^2 \log L}{\partial m^2} = 2 \sum_{i=1}^{n} \frac{(x_i - m)^2 - b^2}{\left[b^2 + (x_i - m)^2\right]^2}$$

$$\frac{\partial^2 \log L}{\partial b \, \partial m} = \frac{\partial^2 \log L}{\partial m \, \partial b} = -4 \sum_{i=1}^{n} \frac{b(x_i - m)}{\left[b^2 + (x_i - m)^2\right]^2}$$

$$\frac{\partial^2 \log L}{\partial b^2} = \frac{1}{b^2} \left[n - 2 \sum_{i=1}^{n} \frac{(x_i - m)^2 \left[(x_i - m)^2 + 3b^2\right]}{\left[b^2 + (x_i - m)^2\right]^2} \right]$$

The Logistic

The Log-Likelihood

$$LL = \frac{n}{\beta}(\mu - \bar{x}) - n \log \beta - 2 \sum_{i=1}^{n} \log \left[1 + e^{-\frac{x_i - \mu}{\beta}} \right]$$

For the analytical calculations see Appendix A.

1st Order.

$$\frac{\partial LL}{\partial \mu} = \frac{1}{\beta} \left[n - 2 \sum_{i=1}^{n} \frac{e^{-\frac{x_i - \mu}{\beta}}}{1 + e^{-\frac{x_i - \mu}{\beta}}} \right]$$

$$\frac{\partial LL}{\partial \beta} = -\frac{n}{\beta^2}(\mu - \bar{x}) - \frac{n}{\beta} - \frac{2}{\beta^2} \sum_{i=1}^{n} \frac{(x_i - \mu) e^{-\frac{x_i - \mu}{\beta}}}{1 + e^{-\frac{x_i - \mu}{\beta}}}$$

2nd Order.

$$\frac{\partial^2 LL}{\partial \mu^2} = -\frac{2}{\beta^2}\sum_{i=1}^{n}\frac{e^{-\frac{x_i-\mu}{\beta}}}{\left[1+e^{-\frac{x_i-\mu}{\beta}}\right]^2}$$

$$\frac{\partial^2 LL}{\partial \beta\,\partial \mu} = \frac{\partial^2 LL}{\partial \mu\,\partial \beta} = -\frac{1}{\beta^2}\left[n - 2\sum_{i=1}^{n}\frac{e^{-\frac{x_i-\mu}{\beta}}}{\left[1+e^{-\frac{x_i-\mu}{\beta}}\right]^2}\right] - \frac{2}{\beta^3}\sum_{i=1}^{n}\frac{(x_i-\mu)e^{-\frac{x_i-\mu}{\beta}}}{\left[1+e^{-\frac{x_i-\mu}{\beta}}\right]^2}$$

$$\frac{\partial^2 LL}{\partial \beta^2} = (\mu-\bar{x})\frac{2n}{\beta^3} + \frac{n}{\beta^2} + \frac{4}{\beta^3}\sum_{i=1}^{n}\frac{(x_i-\mu)e^{-\frac{x_i-\mu}{\beta}}}{1+e^{-\frac{x_i-\mu}{\beta}}} - \frac{2}{\beta^4}\sum_{i=1}^{n}\frac{(x_i-\mu)^2 e^{-\frac{x_i-\mu}{\beta}}}{\left[1+e^{-\frac{x_i-\mu}{\beta}}\right]^2}$$

4.1.2 b. The Newton-Raphson Implementation

The data

The Cauchy data seem as if they were generated also from a logarithmic generative algorithm, as you can see by implementing the below plots.

```
# Read the CSV file
data <- read.csv("NewtonRaphson.csv", header = TRUE)

# Assuming the data is structured with the first column for Logistic and
↪  the second for Cauchy
logistic_data <- data$LOGIS
cauchy_data <- data$CAUCHY

# Plot for Logistic data
plot(logistic_data, col = "blue", main = "Logistic Distribution",
     xlab = "Index", ylab = "Value", pch = 19)


# Plot for Logistic data
plot(sort(logistic_data), col = "blue", main = "Logistic Distribution",
     xlab = "Index", ylab = "Value", pch = 19)

# Plot for Cauchy data
plot(cauchy_data, col = "red", main = "Cauchy Distribution",
     xlab = "Index", ylab = "Value", pch = 19)

# The plot for Cauchy data
plot(sort(cauchy_data), col = "red", main = "Cauchy Distribution",
     xlab = "Index", ylab = "Value", pch = 19)
```

The plots 4.3, 4.4, 4.1, 4.2, show as we previously mentioned that logarithmic curve, even thought we will proceed with our approach having in mind that the implementation could (most probably will) not capture the correct relationship of the data.

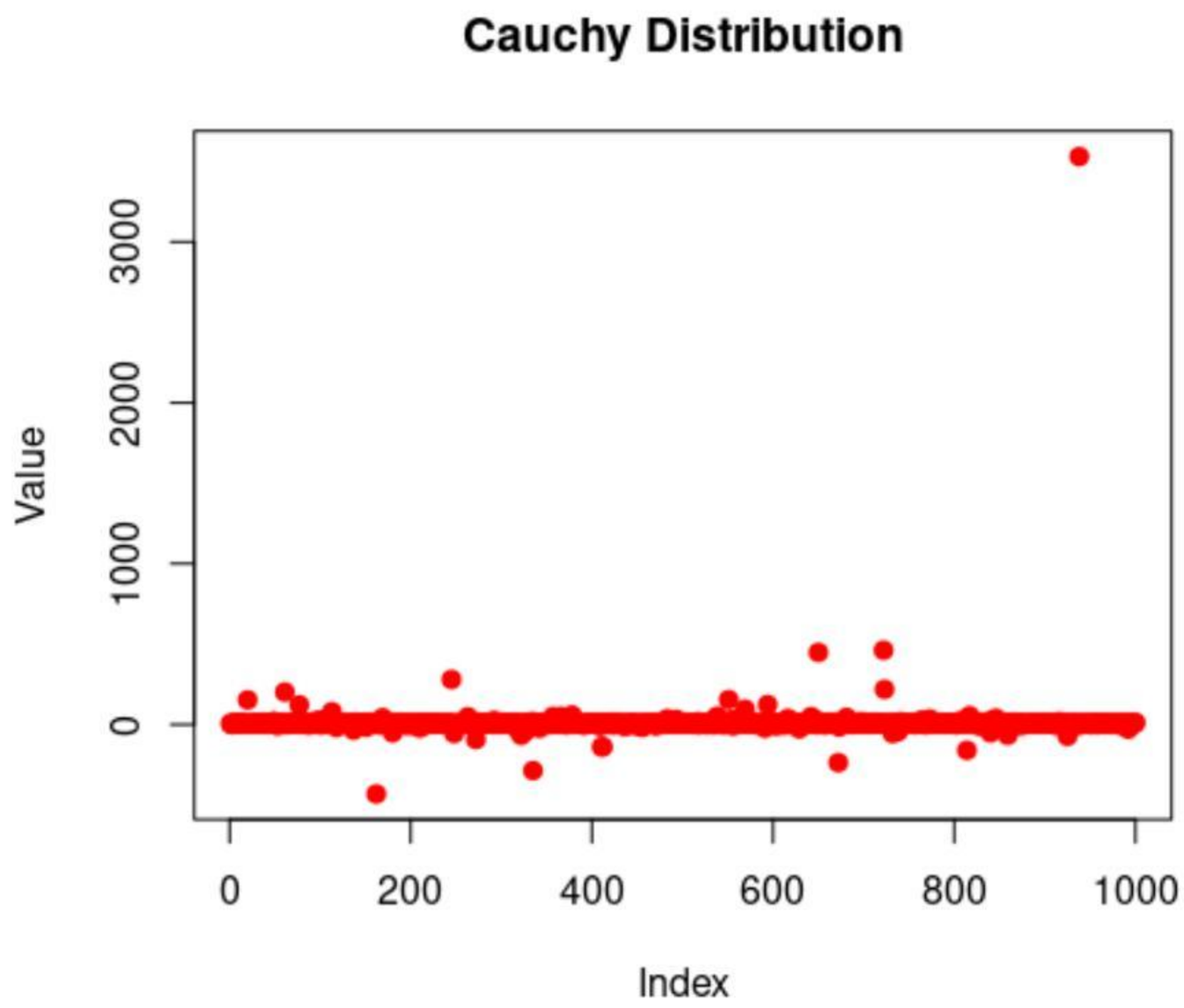

Figure 4.1: The "Cauchy" data

The Cauchy implementation

The main Code

```
cauchy_newton_raphson_regularized <- function(data, start_vals, tol =
↪  1e-12, max_iter = 100000, lambda = 1e-6) {
  m <- start_vals[1]
  b <- start_vals[2]

  for (i in 1:max_iter) {
    # First derivatives
    d1_m <- 2 * sum((data - m) / (b^2 + (data - m)^2))
    d1_b <- (1/b) * (-length(data) + 2 * sum((data - m)^2 / (b^2 + (data -
    ↪  m)^2)))

    # Second derivatives
    d2_mm <- 2 * sum(((data - m)^2 - b^2) / ((b^2 + (data - m)^2)^2))
    d2_mb <- -4 * sum(b * (data - m) / ((b^2 + (data - m)^2)^2))
    d2_bb <- (1/b^2) * (length(data) - 2 * sum((data - m)^2 * ((data -
    ↪  m)^2 + 3*b^2) / ((b^2 + (data - m)^2)^2)))

    # Hessian matrix with regularization
    Hessian <- matrix(c(d2_mm, d2_mb, d2_mb, d2_bb), nrow = 2)
    diag(Hessian) <- diag(Hessian) + lambda  # Adding regularization term

    # Gradient vector
    gradient <- c(d1_m, d1_b)
```

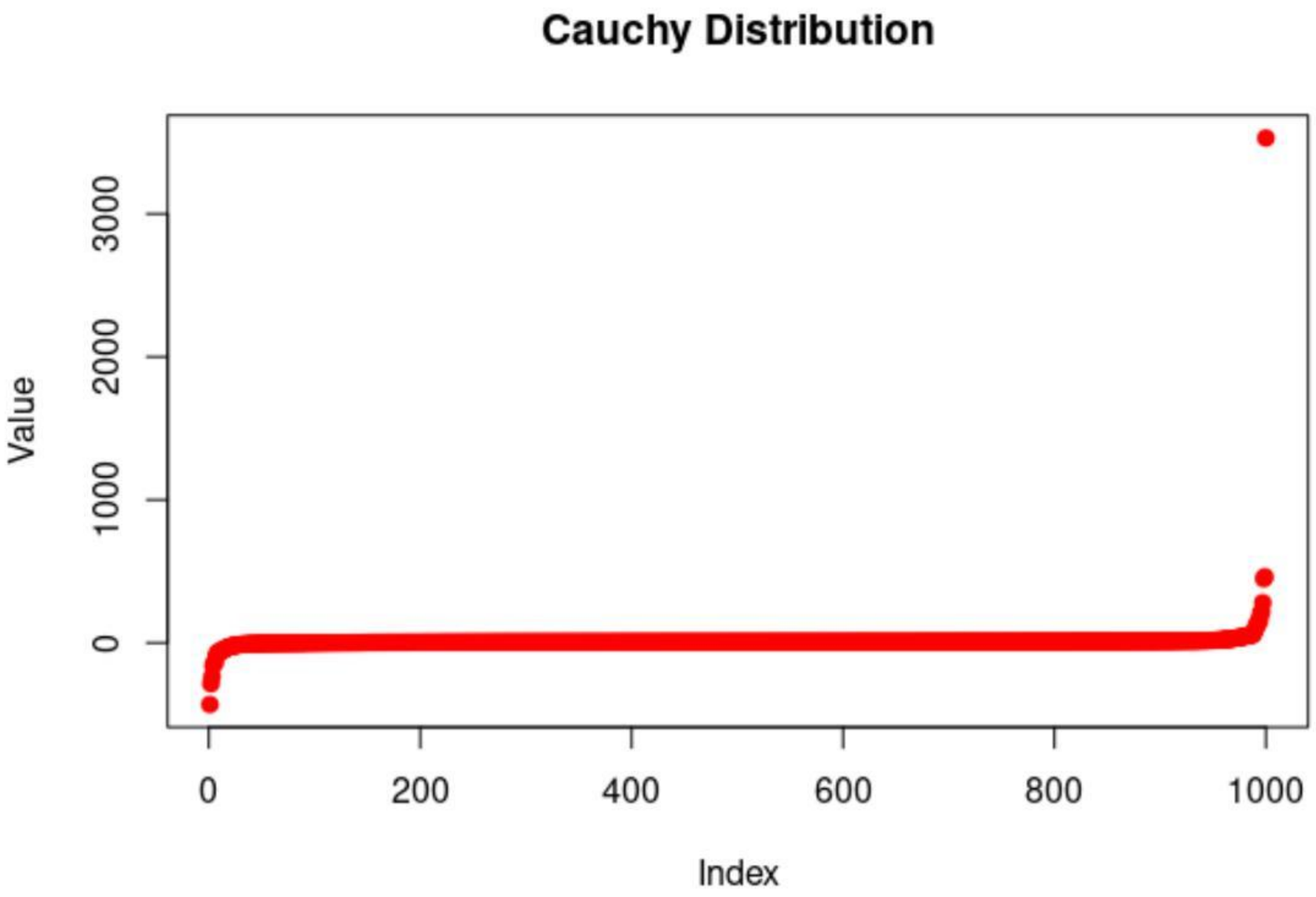

Figure 4.2: The sorted "Cauchy" data

```
  # Newton-Raphson update
  update <- solve(Hessian) %*% gradient
  m <- m - update[1]
  b <- b - update[2]

  # Check for convergence
  if (sqrt(sum(update^2)) < tol) {
    break
  }
}

if (i == max_iter) {
  warning("Algorithm did not converge")
}

  return(list(m = m, b = b, iterations = i))
}
```

The issue of starting values:

There is an extended bibliography regarding the choice of the initial values. In this report, we are presenting 3 approaches. In all of these approaches, m will be equal to the median(data).

- **The instructed approach** where b will be set as median(abs(data-m))
- **The commonly found** one where b= IQR(data)/2

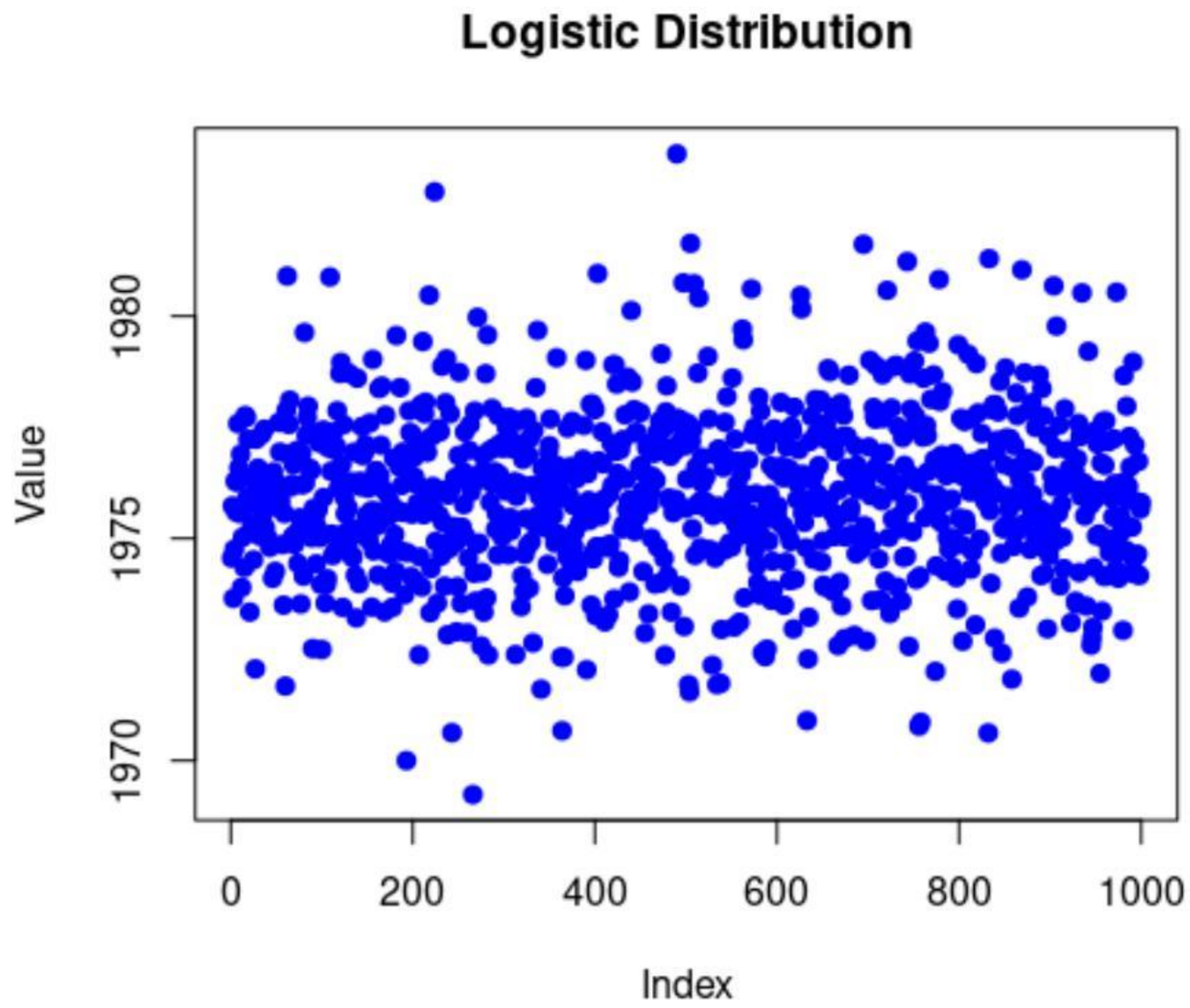

Figure 4.3: The Logistic data

- **The sophisticated one** where for the set value of a (the median) we will search for the appropriate value of b iteratively based on a theorem presented in the Cauchy Thesis

The instructed approach

```
# Find initial values
m_initial <- median(cauchy_data)
b_initial <- median(abs(cauchy_data-m_initial))
# Use these initial values in the Newton-Raphson procedure
start_vals <- c(m_initial, b_initial)
cauchy_estimates_regularized <-
↪  cauchy_newton_raphson_regularized(cauchy_data, start_vals)
```

Resulting in $m = 5.966359$ and $b = 1.882497$ with 5 iterations for convergence, while the initial_b= 1.825662

The commonly found

```
# Find initial values
m_initial <- median(cauchy_data)
# b_initial should be a robust estimate of the scale parameter
b_initial <- IQR(cauchy_data) / 2

start_vals <- c(m_initial, b_initial)
cauchy_estimates_regularized <-
↪  cauchy_newton_raphson_regularized(cauchy_data, start_vals)
```

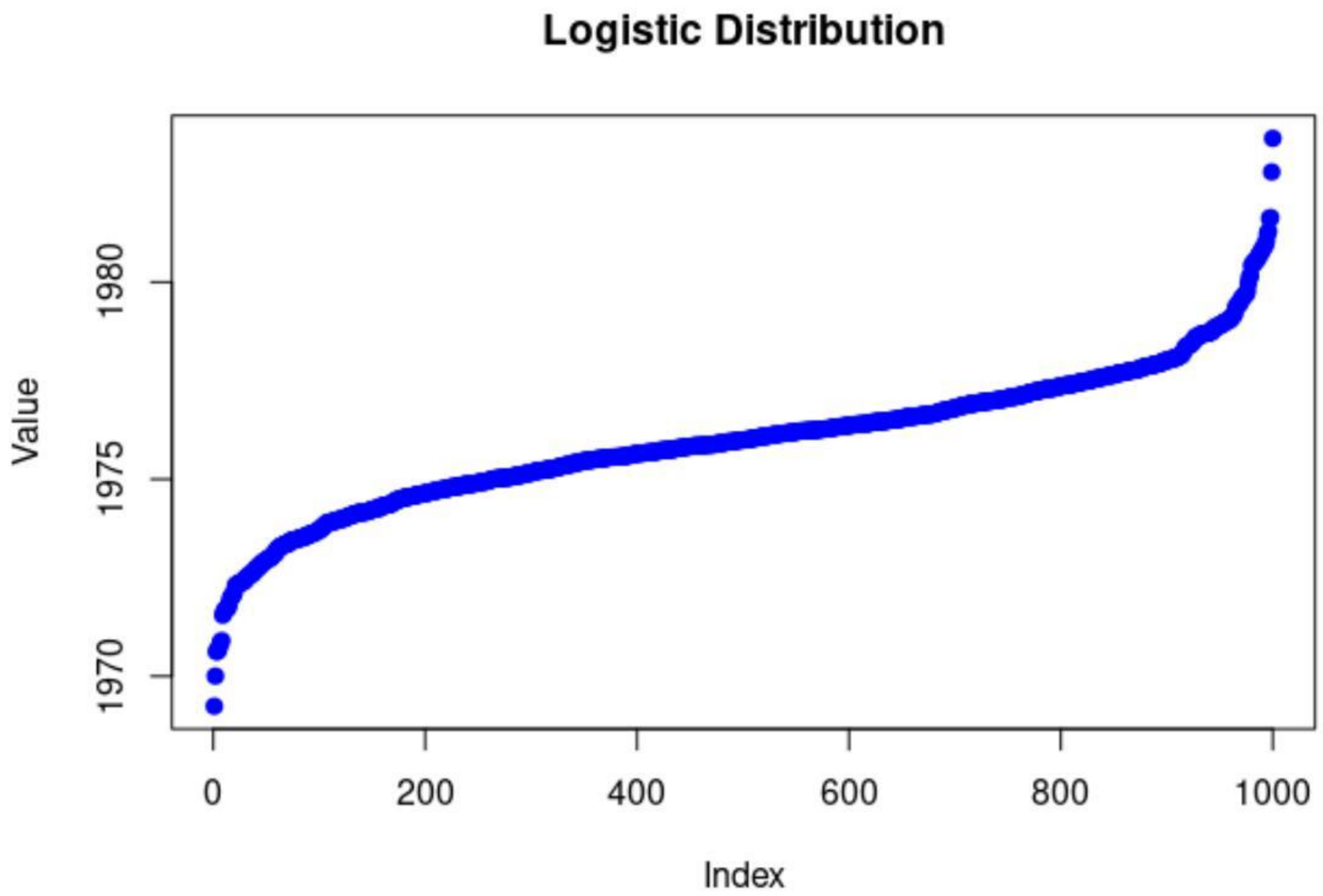

Figure 4.4: The sorted Logistic data

The results as we hoped are the same (also the iterations), although our initial_b value was 1.817715.

The sophisticated one

```
# Function to find the initial estimate of 'b'
find_initial_b <- function(data, m) {
  b <- 0.1  # Start with a small positive value
  last_sign <- sign(first_derivative_b(data, m, b))

  while (TRUE) {
    b <- b + 0.1  # Increment b
    current_sign <- sign(first_derivative_b(data, m, b))

    # Check if the sign of the first derivative has changed
    if (current_sign != last_sign) {
      break
    }
  }
  return(b)
}

# Function for the first derivative of log-likelihood with respect to 'b'
first_derivative_b <- function(data, m, b) {
  return((1/b) * (-length(data) + 2 * sum((data - m)^2 / (b^2 + (data -
  ↪  m)^2))))
}
```

```
# Find initial values
m_initial <- median(cauchy_data)
b_initial <- find_initial_b(cauchy_data, m_initial)

start_vals <- c(m_initial, b_initial)
cauchy_estimates_regularized <-
↪  cauchy_newton_raphson_regularized(cauchy_data, start_vals)
```

Again the same results only now the solution converged in 4 iterations, with the initial_b= 1.9.

My generative data

```
#creating my own data

# Number of data points
n <- 1000

# Location parameter
location <- 10

# Scale parameter
scale <- 5

# Generate Cauchy distributed data
gen_cauchy_data <- rcauchy(n, location, scale)

# Display the first few data points
head(gen_cauchy_data)

# Density plot
plot(density(gen_cauchy_data), main = "Density Plot of Cauchy Data", xlab
↪  = "Value", ylab = "Density")
```

Now the difference between the provided Cauchy data and the above generated is evident by observing the plot 4.5

For those data, we will just implement the sophisticated iterative way of choosing b as described above and then fit as seen below:

```
# Find initial values
m_initial <- median(gen_cauchy_data)
b_initial <- find_initial_b(gen_cauchy_data, m_initial)

# Use these initial values in the Newton-Raphson procedure
start_vals <- c(m_initial, b_initial)
cauchy_estimates_regularized <-
↪  cauchy_newton_raphson_regularized(gen_cauchy_data, start_vals)
```

The results are pretty satisfying with $m = 9.910212$ and $b = 4.914455$ with 4 iterations and initial_b= 5. After some experimentation with the tol, we found that by changing it to $e-15$ the results stayed the same and the fit converged in the 6_{th} iteration, while for tol set to $e-15$ it did not converge at all. For the commonly found and instructed approach the results were the same with convergence on the 5_{th} iteration and initial_b values $4.837359, 4.884573$ respectively.

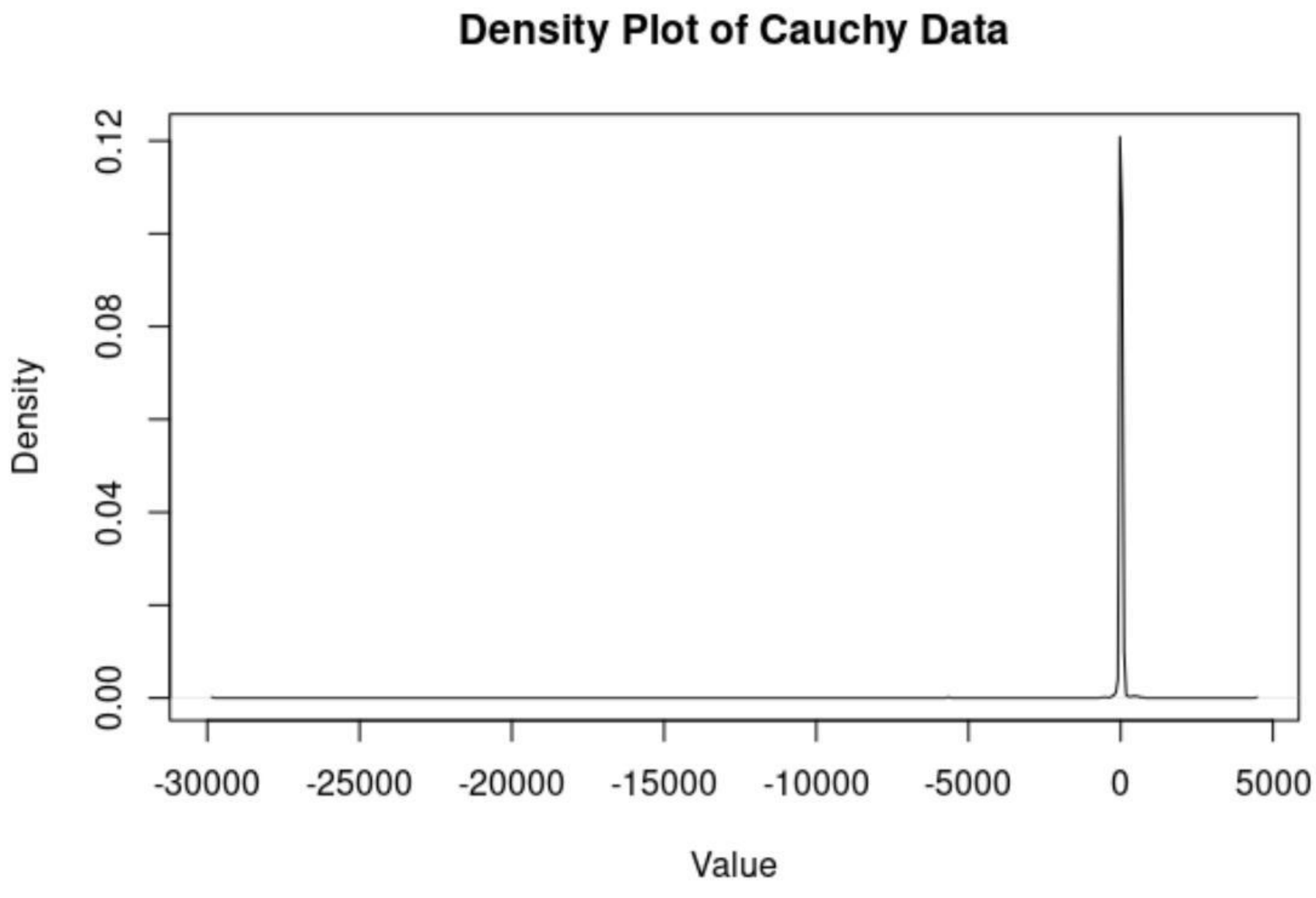

Figure 4.5: The Generated Cauchy data

The Logistic implementation

```
logistic_newton_raphson <- function(data, start_vals, tol = 1e-8, max_iter
↪  = 10000) {
  mu <- start_vals[1]
  beta <- start_vals[2]
  n <- length(data)
  x_bar <- mean(data)

  for (i in 1:max_iter) {
    # First derivatives
    dLL_dmu <- (1/beta) * (n - 2 * sum(exp(-(data - mu)/beta) / (1 +
    ↪  exp(-(data - mu)/beta))))
dLL_dbeta <- -(n / beta^2) * (mu - x_bar) - (n / beta) - (2 / beta^2)
↪  * sum((data - mu) * exp(-(data - mu)/beta) / (1 + exp(-(data -
↪  mu)/beta)))

# Second derivatives
d2LL_dmu2 <- -(2 / beta^2) * sum(exp(-(data - mu)/beta) / ((1 +
↪  exp(-(data - mu)/beta))^2))
d2LL_dmudbeta <- -(1/beta^2) * (length(logistic_data)
↪  -2*sum(exp(-(data - mu)/beta) / ((1 + exp(-(data - mu)/beta))))) -
↪  (2/beta^3) * sum((data - mu) * exp(-(data - mu)/beta) / ((1 +
↪  exp(-(data - mu)/beta))^2))
```

```
d2LL_dbeta2 <- (2 * n / beta^3) * (mu - x_bar) + (n / beta^2) + (4 /
↪  beta^3) * sum((data - mu) * exp(-(data - mu)/beta) / (1 +
↪  exp(-(data - mu)/beta))) - (2 / beta^4) * sum(((data - mu)^2) *
↪  exp(-(data - mu)/beta) / ((1 + exp(-(data - mu)/beta))^2))

# Hessian matrix
Hessian <- matrix(c(d2LL_dmu2, d2LL_dmudbeta, d2LL_dmudbeta,
↪  d2LL_dbeta2), nrow = 2)

# Gradient vector
  gradient <- c(dLL_dmu, dLL_dbeta)

  # Newton-Raphson update
  update <- solve(Hessian, gradient)
  mu <- mu - update[1]
  beta <- beta - update[2]

  # Check for convergence
  if (sqrt(sum(update^2)) < tol) {
    break
  }
}

if (i == max_iter) {
  warning("Algorithm did not converge")
}

  return(list(mu = mu, beta = beta, iterations = i))
}
```

For initial values, we will use the data mean and standard deviation as presented below:

```
# Example initial values
mu_initial <- mean(logistic_data)
beta_initial <- sd(logistic_data)

# Example data (using rlogis to generate logistic distribution data)
logistic_data <- rlogis(n = 100, location = mu_initial, scale =
↪  beta_initial)

# Run the Newton-Raphson algorithm
estimates <- logistic_newton_raphson(logistic_data, c(mu_initial,
↪  beta_initial))
```

The results are $\mu = 1976.454$ and $\beta = 1.812591$ and the convergence took 5 iterations.

4.1.3 c. The confidence Intervals

CI using Fisher Information

For the logistic:
Given the complexity of the computations and our inability to solve some integrals that were produced (see Appendix A) we will compute the confidence intervals by approximation the Fisher Information Matrix: $I(\theta) \approx -l''(\theta)$. By using the below code we get the estimated CI, the Fisher Information matrix and it's inverse.

```
# Function to compute the Fisher information matrix for the logistic
↪  distribution
compute_fisher_info_logistic <- function(data, mu, beta) {
  n <- length(data)

  d2LL_dmu2 <- (2 / beta^2) * sum(exp(-(data - mu)/beta) / ((1 +
  ↪  exp(-(data - mu)/beta))^2))
  d2LL_dmudbeta <- (1/beta^2) *(n -2*sum(exp(-(data - mu)/beta) / ((1 +
  ↪  exp(-(data - mu)/beta))^2))) + (2/beta^3) * sum((data - mu) *
  ↪  exp(-(data - mu)/beta) / ((1 + exp(-(data - mu)/beta))^2))
  d2LL_dbeta2 <- -(2 * n / beta^3) * (mu - mean(data)) - (n / beta^2) - (4
  ↪  / beta^3) * sum((data - mu) * exp(-(data - mu)/beta) / (1 +
  ↪  exp(-(data - mu)/beta))) + (2 / beta^4) * sum(((data - mu)^2) *
  ↪  exp(-(data - mu)/beta) / ((1 + exp(-(data - mu)/beta))^2))

  fisher_info <- matrix(c(d2LL_dmu2, d2LL_dmudbeta, d2LL_dmudbeta,
  ↪  d2LL_dbeta2), nrow = 2)
  return(fisher_info)
}

# Compute the inverse Fisher information matrix and confidence intervals
compute_conf_intervals_logistic <- function(data, mu, beta, conf_level =
↪  0.95) {
  fisher_info <- compute_fisher_info_logistic(data, mu, beta)
  fisher_info_inv <- solve(fisher_info)

  # Standard errors of the estimates
  std_errors <- sqrt(diag(fisher_info_inv))

  # Compute the confidence intervals
  z_val <- qnorm(1 - (1 - conf_level) / 2)
  conf_intervals_mu <- c(mu - z_val * std_errors[1], mu + z_val *
  ↪  std_errors[1])
  conf_intervals_beta <- c(beta - z_val * std_errors[2], beta + z_val *
  ↪  std_errors[2])

  return(list(mu = conf_intervals_mu, beta = conf_intervals_beta))
}

# Function to compute and print Fisher information matrix and its inverse
↪  for logistic distribution
print_fisher_info_logistic <- function(data, mu, beta) {
```

```r
fisher_info <- compute_fisher_info_logistic(data, mu, beta)
fisher_info_inv <- solve(fisher_info)

cat("Fisher Information Matrix:\n")
print(fisher_info)
cat("\nInverse Fisher Information Matrix:\n")
print(fisher_info_inv)

# Compute confidence intervals
conf_intervals <- compute_conf_intervals_logistic(data, mu, beta)

  cat("\nConfidence Intervals:\n")
  cat("mu: ", conf_intervals$mu, "\n")
  cat("beta: ", conf_intervals$beta, "\n")
}

# The fit to our data and estimates
print_fisher_info_logistic(logistic_data, estimates$mu, estimates$beta)
```

The results where:

```
Fisher Information Matrix:
         [,1]     [,2]
[1,] 10.45956 20.14396
[2,] 20.14396 41.94914

Inverse Fisher Information Matrix:
           [,1]       [,2]
[1,]  1.2715454 -0.6105954
[2,] -0.6105954  0.3170460

Confidence Intervals:
mu:  1974.243 1978.664
beta:  0.708997 2.916184
```

For the Cauchy

The logic is the same as before, by using the below code we get the results.

```r
# Function to compute the Fisher information matrix for the Cauchy
↪  distribution
compute_fisher_info_cauchy <- function(data, m, b) {
  n <- length(data)

  d2LL_dm2 <- -2 * sum(((data - m)^2 - b^2) / ((b^2 + (data - m)^2)^2))
  d2LL_dmb <- 4 * sum(b * (data - m) / ((b^2 + (data - m)^2)^2))
  d2LL_db2 <- -(1/b^2) * (n - 2 * sum(((data - m)^2 * ((data - m)^2 + 3 *
  ↪  b^2)) / ((b^2 + (data - m)^2)^2)))

  fisher_info <- matrix(c(d2LL_dm2, d2LL_dmb, d2LL_dmb, d2LL_db2), nrow =
  ↪  2)
  return(fisher_info)
```

```
}

# Function to compute and print Fisher information matrix and its inverse
↪  for Cauchy distribution
print_fisher_info_cauchy <- function(data, m, b, conf_level = 0.95) {
  fisher_info <- compute_fisher_info_cauchy(data, m, b)
  fisher_info_inv <- solve(fisher_info)

  cat("Fisher Information Matrix for Cauchy Distribution:\n")
  print(fisher_info)
  cat("\nInverse Fisher Information Matrix:\n")
  print(fisher_info_inv)

  # Standard errors of the estimates
  std_errors <- sqrt(diag(fisher_info_inv))

  # Compute the confidence intervals
  z_val <- qnorm(1 - (1 - conf_level) / 2)
  conf_intervals_m <- c(m - z_val * std_errors[1], m + z_val *
  ↪  std_errors[1])
  conf_intervals_b <- c(b - z_val * std_errors[2], b + z_val *
  ↪  std_errors[2])

  cat("\nConfidence Intervals:\n")
  cat("m: ", conf_intervals_m, "\n")
  cat("b: ", conf_intervals_b, "\n")
}

# CI and Fisher Info for the Cauchy data and estimates
print_fisher_info_cauchy(cauchy_data, 5.966359, 1.882497)
```

with results:

```
Fisher Information Matrix for Cauchy Distribution:
             [,1]        [,2]
[1,] 144.2594746  -0.3729062
[2,]  -0.3729062 137.9238945

Inverse Fisher Information Matrix:
              [,1]         [,2]
[1,] 6.932002e-03 1.874212e-05
[2,] 1.874212e-05 7.250426e-03

Confidence Intervals:
m:  5.803175 6.129543
b:  1.715607 2.049387
```

The Bootstrapping

By using the below code we get our Bootstrapping confidence Intervals

```
# Bootstrap function for Logistic distribution parameters
```

```r
bootstrap_logistic <- function(data, B = 200) {
  n <- length(data)
  bootstrap_estimates <- matrix(NA, nrow = B, ncol = 2) # Matrix to store
  ↪  bootstrap estimates

  for (i in 1:B) {
    # Sampling with replacement
    bootstrap_sample <- sample(data, size = n, replace = TRUE)

    # Estimate parameters for the bootstrap sample using Newton-Raphson or
    ↪  another method
    estimates <- logistic_newton_raphson(bootstrap_sample, c(mu_initial,
    ↪  beta_initial))
    mu_bootstrap <- estimates$mu
    beta_bootstrap <- estimates$beta

    # Store the estimates
    bootstrap_estimates[i, ] <- c(mu_bootstrap, beta_bootstrap)
}

# Compute confidence intervals
conf_intervals_mu <- quantile(bootstrap_estimates[, 1], probs = c(0.025,
↪  0.975))
conf_intervals_beta <- quantile(bootstrap_estimates[, 2], probs =
↪  c(0.025, 0.975))

  return(list(mu = conf_intervals_mu, beta = conf_intervals_beta))
}

# Fitting for our Logistic data
bootstrap_results_logistic <- bootstrap_logistic(logistic_data)
print(bootstrap_results_logistic)
```

with results

```
$mu
    2.5%    97.5%
1975.445 1976.585

$beta
    2.5%    97.5%
1.449980 1.922973
```

As we expected the Bootstrapping procedure produces better (smaller) confidence intervals.

For the Cauchy

```r
# Bootstrap function for Cauchy distribution parameters
cauchy_bootstrap <- function(data, B) {
  n <- length(data)
  bootstrap_estimates <- matrix(nrow = B, ncol = 2)  # To store m and b
  ↪  estimates
```

```
for (i in 1:B) {
  # Sampling with replacement
  bootstrap_sample <- sample(data, size = n, replace = TRUE)

  # Initial estimates for m and b
  m_initial <- median(bootstrap_sample)
  b_initial <- median(abs(bootstrap_sample - m_initial))

  # Estimate parameters (m and b) for the bootstrap sample using
  ↪  Newton-Raphson Regularized
  m_bootstrap <- estimates[1]
  b_bootstrap <- estimates[2]

  # Store the estimates
  bootstrap_estimates[i, ] <- c(m_bootstrap, b_bootstrap)
}

# Compute confidence intervals
conf_intervals_m <- quantile(bootstrap_estimates[, 1], probs = c(0.025,
↪  0.975))
conf_intervals_b <- quantile(bootstrap_estimates[, 2], probs = c(0.025,
↪  0.975))

return(list(conf_intervals_m = conf_intervals_m, conf_intervals_b =
↪  conf_intervals_b))
}
# Fitting our Cauchy data
bootstrap_results <- cauchy_bootstrap(cauchy_data, 200)
print(bootstrap_results)
```

with results:

```
$conf_intervals_m
    2.5%    97.5%
5.828928 6.129143

$conf_intervals_b
    2.5%    97.5%
1.720611 2.062911
```

As we can see there is little difference with the CI derived from the Fisher Information matrix. However, the bootstrapping confidence intervals are of less length one's more.

4.1.4 Monte Carlo

For the Cauchy

```
monte_carlo_experiment <- function(N, B, num_experiments) {
  # Initialize matrices to store estimates and confidence intervals
  estimates_matrix <- matrix(nrow = num_experiments, ncol = 2)
```

```
ci_matrix <- matrix(nrow = num_experiments, ncol = 4)  # 2 parameters
↪  each with 2 CI bounds

for (i in 1:num_experiments) {
  # Generate a new random sample
  sample_data <- rcauchy(N)

  # Estimate parameters for the new sample
  m_estimate <- median(sample_data)
  b_estimate <- median(abs(sample_data - m_estimate))
  estimates_matrix[i, ] <- c(m_estimate, b_estimate)

  # Perform bootstrap to get confidence intervals
  bootstrap_results <- cauchy_bootstrap(sample_data, B)
  ci_matrix[i, 1:2] <- bootstrap_results$conf_intervals_m
  ci_matrix[i, 3:4] <- bootstrap_results$conf_intervals_b
}

# Calculate the mean and standard deviation of the estimates
estimates_summary <- colMeans(estimates_matrix)
estimates_sd <- apply(estimates_matrix, 2, sd)

# Calculate the mean and standard deviation of the confidence intervals
ci_summary <- colMeans(ci_matrix)
ci_sd <- apply(ci_matrix, 2, sd)

  return(list(estimates = list(mean = estimates_summary, sd =
  ↪  estimates_sd),
              ci = list(mean = ci_summary, sd = ci_sd)))
}

# Initializations
N = 100
B = 200
num_experiments = 100
results <- monte_carlo_experiment(N, B, num_experiments)

# Results contain mean and standard deviation for parameter estimates and
↪  confidence intervals
print(results)
```

with results:

```
$estimates$mean
[1] 0.01488739 1.03663574

$estimates$sd
[1] 0.1555433 0.1608390

$ci
$ci$mean
[1] -0.2772009  0.3071475  0.7694425  1.3392676
```

```
$ci$sd
[1] 0.1500612 0.1533920 0.1109658 0.1959637
```

For the Logistic

For the Monte Carlo to work we had to revise the original function that estimates Newton-Raphson using the addition of a small constant in the Hessian to avoid singularity, though the results indicate pathogenicity in our data or in our derivatives, given the chance for similar future work on this part, we should revise the initial Newton-Raphson estimation algorithm, though it's probably a bug in the code, nevertheless, a revision on the derivatives should be priority number one, and then a reconstruction of the Monte Carlo experiment.

```
monte_carlo_logistic <- function(N, B, mu_true, beta_true) {
  monte_carlo_results <- matrix(NA, nrow = B, ncol = 6) # Matrix to store
  ↪  results: mu_est, beta_est, mu_low, mu_high

  for (i in 1:B) {
    # Simulate a dataset from the logistic distribution
    simulated_data <- rlogis(N, location = mu_true, scale = beta_true)

    # Estimate parameters using Newton-Raphson or another method
    estimates <- logistic_newton_raphson(simulated_data, c(mu_initial,
    ↪  beta_initial))

    # Compute bootstrap confidence intervals
    bootstrap_results <- bootstrap_logistic(simulated_data)

    # Store the results
    monte_carlo_results[i, 1:2] <- c(estimates$mu, estimates$beta)
    monte_carlo_results[i, 3:6] <- c(bootstrap_results$mu[1],
    ↪  bootstrap_results$mu[2], bootstrap_results$beta[1],
    ↪  bootstrap_results$beta[2])
  }

  colnames(monte_carlo_results) <- c("mu_estimate", "beta_estimate",
  ↪  "mu_low", "mu_high")
  return(monte_carlo_results)
}

# Example usage
mu_true <-1975   # True value of mu (replace with the actual value)
beta_true <- 1.5  # True value of beta (replace with the actual value)
N <- 100       # Sample size for each simulation
B <- 100       # Number of simulations


results <- monte_carlo_logistic(N, B, mu_true, beta_true)
```

with troubling results:

```
$estimates$mean
[1] 0.01488739 1.03663574

$estimates$sd
[1] 0.1555433 0.1608390


$ci
$ci$mean
[1] -0.2772009  0.3071475  0.7694425  1.3392676

$ci$sd
[1] 0.1500612 0.1533920 0.1109658 0.1959637
```

4.2 Part 2

Consider the density function defined as

$$f(x) = \frac{1 - \cos(x - \theta)}{2\pi} \quad \text{on} \quad 0 \leq x \leq 2\pi,$$

where θ is a parameter between $-\pi$ and π. The following i.i.d. data arise from this density: $3.91, 4.85, 2.28, 4.06, 3.70, 4.04, 5.46, 3.53, 2.28, 1.96, 2.53, 3.88, 2.22, 3.47, 4.82, 2.46, 2.99, 2.54, 0.52, 2.50$. We wish to estimate θ.

(a) Graph the log likelihood function between $-\pi$ and π.

(b) Find the method of moments estimator of θ.

(c) Find the MLE for θ using the Newton-Raphson method, using the result from (b) as the starting value. What solution do you find when you start at -2.7 and 2.7?

(d) Repeat part (c) using 200 equally spaced starting values between $-\pi$ and π. Partition the interval between $-\pi$ and π into sets of attraction. In other words, divide the set of starting values into separate groups, with each group corresponding to a separate unique outcome of the optimization (a local mode). Discuss your results.

(e) Find two starting values, as nearly equal as you can, for which the Newton-Raphson method converges to two different solutions.

4.2.1 a. The Graph

This part is pretty straightforward, we just create our plot (4.6) using the code:

```
#a. Graph the log likelihood function between -π and π
data <- c(3.91, 4.85, 2.28, 4.06, 3.70, 4.04, 5.46, 3.53, 2.28, 1.96,
          2.53, 3.88, 2.22, 3.47, 4.82, 2.46, 2.99, 2.54, 0.52, 2.50)

log_likelihood <- function(theta, data) {
```

```
  sum(log((1 - cos(data - theta)) / (2 * pi)))
}


theta_vals <- seq(-pi, pi, length.out = 100)
log_likelihood_vals <- sapply(theta_vals, log_likelihood, data = data)

plot(theta_vals, log_likelihood_vals, type = 'l', xlab =
↪  expression(theta),
     ylab = "Log Likelihood", main = "Log Likelihood Function")
```

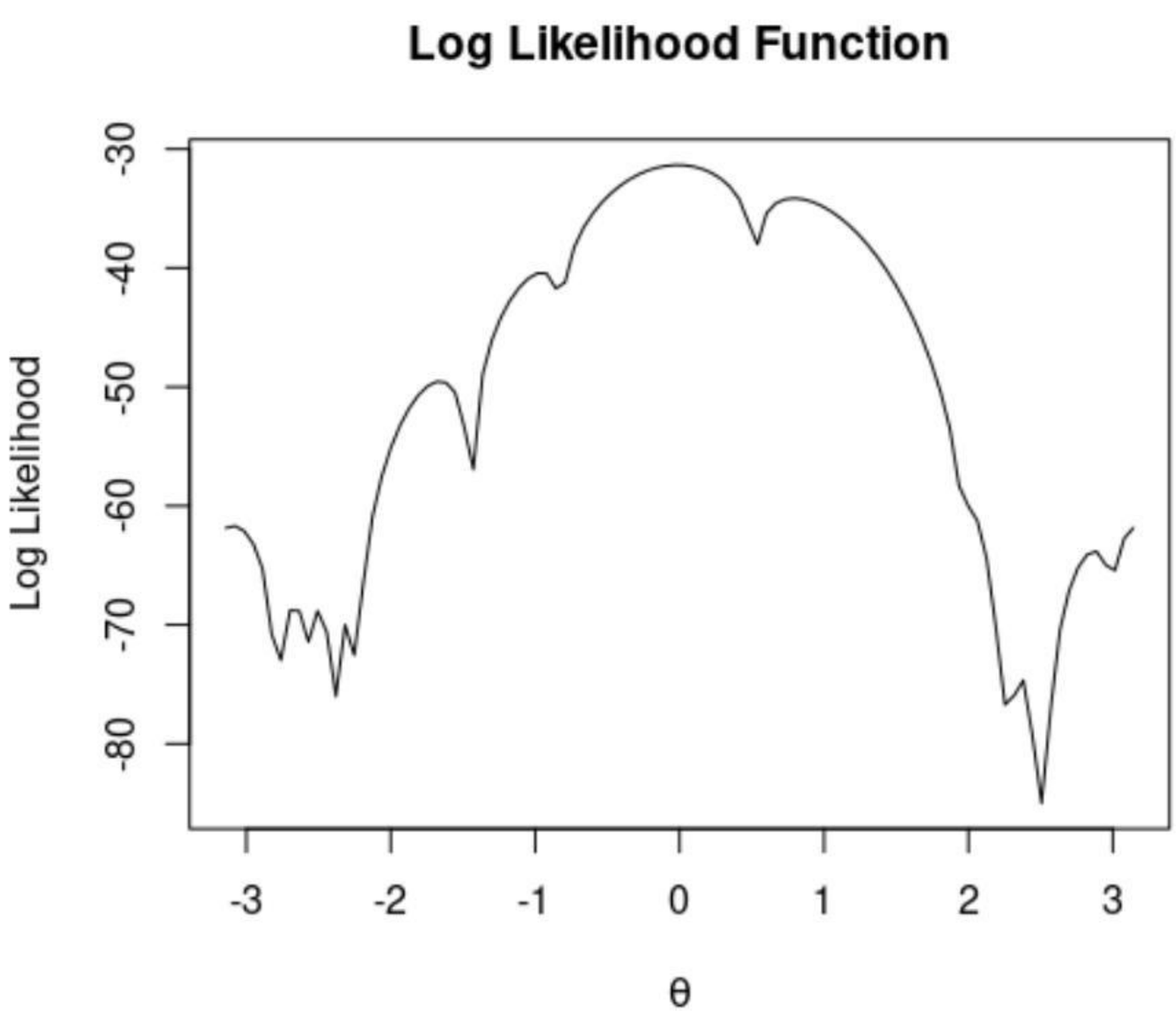

Figure 4.6: The log-likelihood function

Note there is a variety of local maxima. Thought the true maximum should be somewhere in the interval $[0, 0.5]$

4.2.2 b. The method of Moments

Supposing we are not interested in Variance as it does not relate to our distribution (it's not a variable of our distribution), we can just use the first moment a.k.a the mean to estimate our θ as shown in the code below:

```
#b. The method of moments
# Method of Moments Estimator for theta
mom_estimator <- function(data) {
  mean(data)
}

theta_mom <- mom_estimator(data)
```

The result was 3.2 so we expect later our algorithmic approximation of θ to yied a local maximum near our starting 3.2 value.

4.2.3 c.The MLE using N-R

To use Newton-Raphson we need to calculate first the derivatives of our log-likelihood up to the second order.

The derivatives

Supposing $(X_1, \ldots, X_n) \sim$ the distribution with density $f(x) = \frac{1-cos(x-\theta)}{2\pi}$. The log-likelihood is:

$$log(L(\theta)) = -n\log(2\pi) + \sum_{i=1}^{n} \log(1 - cos(x - \theta))$$

, with derivatives:

1st Order.

$$\frac{d\log(L)}{d\theta} = -\sum_{i=1}^{n} \frac{sin(x-\theta)}{1 - cos(x-\theta)}$$

2nd Order.

$$\frac{d^2\log(L)}{d\theta^2} = \sum_{i=1}^{n} \frac{cos(x-\theta) - 1}{[1 - cos(x-\theta)]^2}$$

For more information on the computations see Appendix A.

The implementation using the mean as an initial guess

```
#c. The MLE for \theta using Newton-Raphson
# The first derivative of the log-likelihood
first_derivative <- function(theta, data) {
  -sum(sin(data - theta) / (1 - cos(data - theta)))
}

# The second derivative of the log-likelihood
second_derivative <- function(theta, data) {
  sum((cos(data - theta) - 1) / (1 - cos(data - theta))^2)
}

# Newton-Raphson method to find the MLE of theta
newton_raphson <- function(data, theta_init, max_iter = 100, tol = 1e-6) {
  theta <- theta_init
  for (i in 1:max_iter) {
    theta_old <- theta
    theta <- theta - first_derivative(theta, data) /
    ↪  second_derivative(theta, data)
    # Check for convergence
```

```
  if (abs(theta - theta_old) < tol) {
    break
  }
}

if (i == max_iter) {
  warning("Maximum number of iterations reached without convergence")
}

return(theta)
}

# Initial guess for theta per request
theta_init <- mean(data)
theta_mle <- newton_raphson(data, theta_init)

# Print the MLE of theta
print(theta_mle)
```

As we imagined the result was 3.190094 which could be the last local maximum depicted in our plot (4.6), though upon further examination it's not, as the max(theta_vals) was 3.141593.

The implementation using the 2.7 and -2.7 as an initial guess

```
# Initial guess for theta could be the 2.7 or -2.7 of the data
theta_init_p <- 2.7
theta_mle_p <- newton_raphson(data, theta_init_p)

theta_init_n <- -2.7
theta_mle_n <- newton_raphson(data, theta_init_n)
```

The results where theta_mle_p= 2.873095 and theta_mle_n=-2.6667, which makes sense, as both appear in the form of local maxima (see 4.6) and are located near the initial guess.

4.2.4 d.Repeat c for 200 equally spaced starting values

Again a pretty straightforward procedure realized by the following R code:

```
# the 200 equally spaced starting values between -pi and pi
starting_points <- seq(-pi, pi, length.out = 200)

# Appling Newton-Raphson method to each
results <- sapply(starting_points, run_newton_raphson, data = data)
```

Below we present the results from which the method's sensitivity, to initial conditions, is evident from the convergence behavior observed. Specifically, the algorithm converged to different values of θ even when initiated from starting points that were nearby cause of the presence of multiple local maxima.

```
>print(results)
  [1] -3.0930917 -3.0930917 -3.0930917 -3.0930917 -3.0930917 -3.0930917
   ↪  -3.0930917 -3.0930917 -3.0930917
 [10] -3.0930917 -3.0930917 -2.7861668 -2.7861668 -2.6666999 -2.6666999
 ↪  -2.6666999 -2.6666999 -2.6666999
 [19] -2.5076132 -2.5076132 -2.5076132 -2.5076132 -2.5076132 -2.5076132
 ↪  -2.3882005 -2.2972562 -2.2972562
 [28] -2.2972562 -2.2972562 -2.2321673 -1.6582832 -1.6582832 -1.6582832
 ↪  -1.6582832 -1.6582832 -1.6582832
 [37] -1.6582832 -1.6582832 -1.6582832 -1.6582832 -1.6582832 -1.6582832
 ↪  -1.6582832 -1.6582832 -1.6582832
 [46] -1.6582832 -1.6582832 -1.6582832 -1.6582832 -1.6582832 -1.6582832
 ↪  -1.6582832 -1.6582832 -1.6582832
 [55] -1.4474788 -0.9533363 -0.9533363 -0.9533363 -0.9533363 -0.9533363
 ↪  -0.9533363 -0.9533363 -0.9533363
 [64] -0.9533363 -0.9533363 -0.9533363 -0.9533363 -0.9533363 -0.9533363
 ↪  -0.9533363 -0.9533363 -0.9533363
 [73] -0.9533363 -0.9533363 -0.0119720 -0.0119720 -0.0119720 -0.0119720
 ↪  -0.0119720 -0.0119720 -0.0119720
 [82] -0.0119720 -0.0119720 -0.0119720 -0.0119720 -0.0119720 -0.0119720
 ↪  -0.0119720 -0.0119720 -0.0119720
 [91] -0.0119720 -0.0119720 -0.0119720 -0.0119720 -0.0119720 -0.0119720
 ↪  -0.0119720 -0.0119720 -0.0119720
[100] -0.0119720 -0.0119720 -0.0119720 -0.0119720 -0.0119720 -0.0119720
↪  -0.0119720 -0.0119720 -0.0119720
[109] -0.0119720 -0.0119720 -0.0119720 -0.0119720 -0.0119720 -0.0119720
↪  -0.0119720 -0.0119720  0.7906013
[118]  0.7906013  0.7906013  0.7906013  0.7906013  0.7906013  0.7906013
↪  0.7906013  0.7906013  0.7906013
[127]  0.7906013  0.7906013  0.7906013  0.7906013  0.7906013  0.7906013
↪  0.7906013  0.7906013  0.7906013
[136]  0.7906013  0.7906013  0.7906013  0.7906013  0.7906013  0.7906013
↪  0.7906013  0.7906013  0.7906013
[145]  0.7906013  0.7906013  0.7906013  0.7906013  0.7906013  0.7906013
↪  0.7906013  0.7906013  0.7906013
[154]  0.7906013  0.7906013  0.7906013  0.7906013  0.7906013  0.7906013
↪  0.7906013  0.7906013  0.7906013
[163]  2.0036449  2.0036449  2.0036449  2.0036449  2.0036449  2.0036449
↪  2.0036449  2.0036449  2.2362194
[172]  2.2362194  2.3607182  2.3607182  2.3607182  2.3607182  2.3607182
↪  2.3607182  2.4753736  2.5135932
[181]  2.8730945  2.8730945  2.8730945  2.8730945  2.8730945  2.8730945
↪  2.8730945  2.8730945  2.8730945
[190]  2.8730945  2.8730945  2.8730945  2.8730945  2.8730945  2.8730945
↪  3.1900936  3.1900936  3.1900936
[199]  3.1900936  3.1900936
```

To further enable us to understand the results we created a set of unique results to partitions for a plot-visual aid that better describes the situation using the code below:

```
# Partition based on unique outcomes
```

```r
unique_results <- unique(results)
partitions <- lapply(unique_results, function(x)
↪  starting_points[which(results == x)])

# Print the partitions
print(partitions)

library(ggplot2)
# Data frame for plotting
plot_data <- data.frame(starting_points = starting_points, results =
↪  results)

ggplot(plot_data, aes(x = starting_points, y = results)) +
  geom_point(alpha = 0.6, color = "blue") +
  ggtitle("Convergence of Newton-Raphson for Different Starting Values") +
  xlab("Starting Value of Theta") +
  ylab("Converged Value of Theta") +
  theme_minimal()

# Data frame for plotting with index
plot_data_index <- data.frame(index = seq_along(starting_points), results
↪  = results)

ggplot(plot_data_index, aes(x = index, y = results)) +
  geom_point(alpha = 0.6, color = "blue") +
  ggtitle("Convergence of Newton-Raphson for Different Starting Indices")
↪  +
  xlab("Index of Starting Value") +
  ylab("Converged Value of Theta") +
  theme_minimal()
```

The resulted plots are presented below (4.7, 4.8)

4.2.5 e. The two close starting values that converge to different solutions

Here we decided to look in the interval $[-1, 0.5]$ for which (we can see from the plot 4.7) on the one hand the values near -1 converge to the -1 solution, while the near enough values around -0.8 converge to 0 which seem to be the actual maximum. We decided to use that visual aid repeatedly upon spacing a continuously enclosed interval as seen in the code below:

```r
#e. Find two values close to each other that converge to different
↪  values:
# Figure value based on step
get_starting_value <- function(index) {
  return(starting_points[index])
}

# Define the function to Run the Newton-Raphson repeatedly
run_newton_raphson_range <- function(data, start, end, n_points) {
```

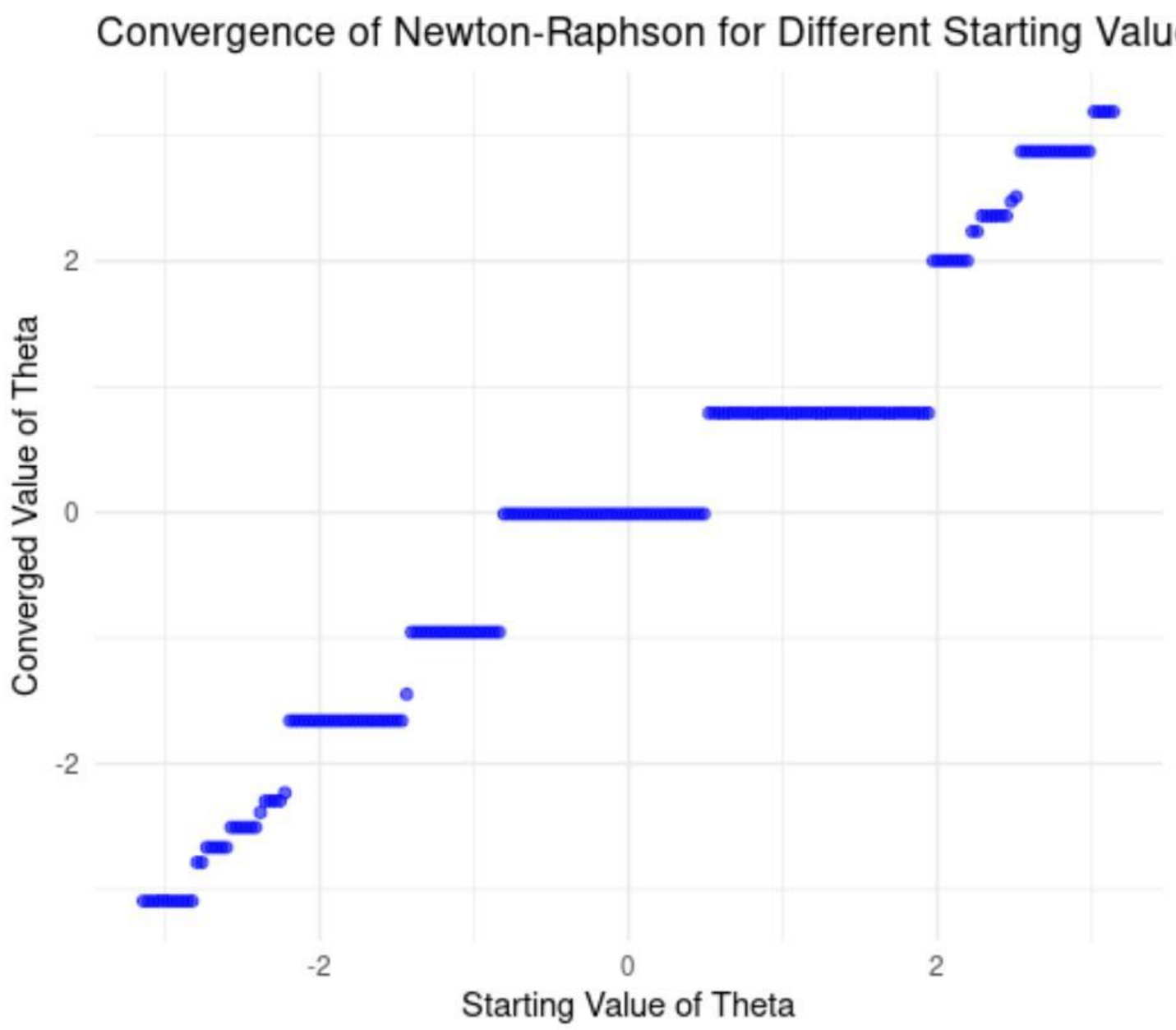

Figure 4.7: Convergence plot based on value

```
subrange_values <- seq(start, end, length.out = n_points)
subrange_results <- sapply(subrange_values, run_newton_raphson, data =
↪  data)

# Data frame for plotting
plot_data_subrange <- data.frame(starting_value = subrange_values,
↪  converged_theta = subrange_results)

p <- ggplot(plot_data_subrange, aes(x = starting_value, y =
↪  converged_theta)) +
  geom_point(alpha = 0.6, color = "blue") +
  ggtitle("Newton-Raphson Convergence for a Range of Starting Values") +
  xlab("Starting Value of Theta") +
  ylab("Converged Value of Theta") +
  theme_minimal()

  return(p)
}
s=get_starting_value(73)
e=get_starting_value(77)
p<-run_newton_raphson_range(data,start=s,end=e, n_points = 1000)
```

Based on this initial run we could observe my Starting values through the plot 4.9 and then further repeatably close our interval by changing the initial values used for the spacing. After a satisfactory difference of $-1.828537e-13$ was created between the different starting values we concluded, using the below code:

```
s=-0.8231863071856829
```

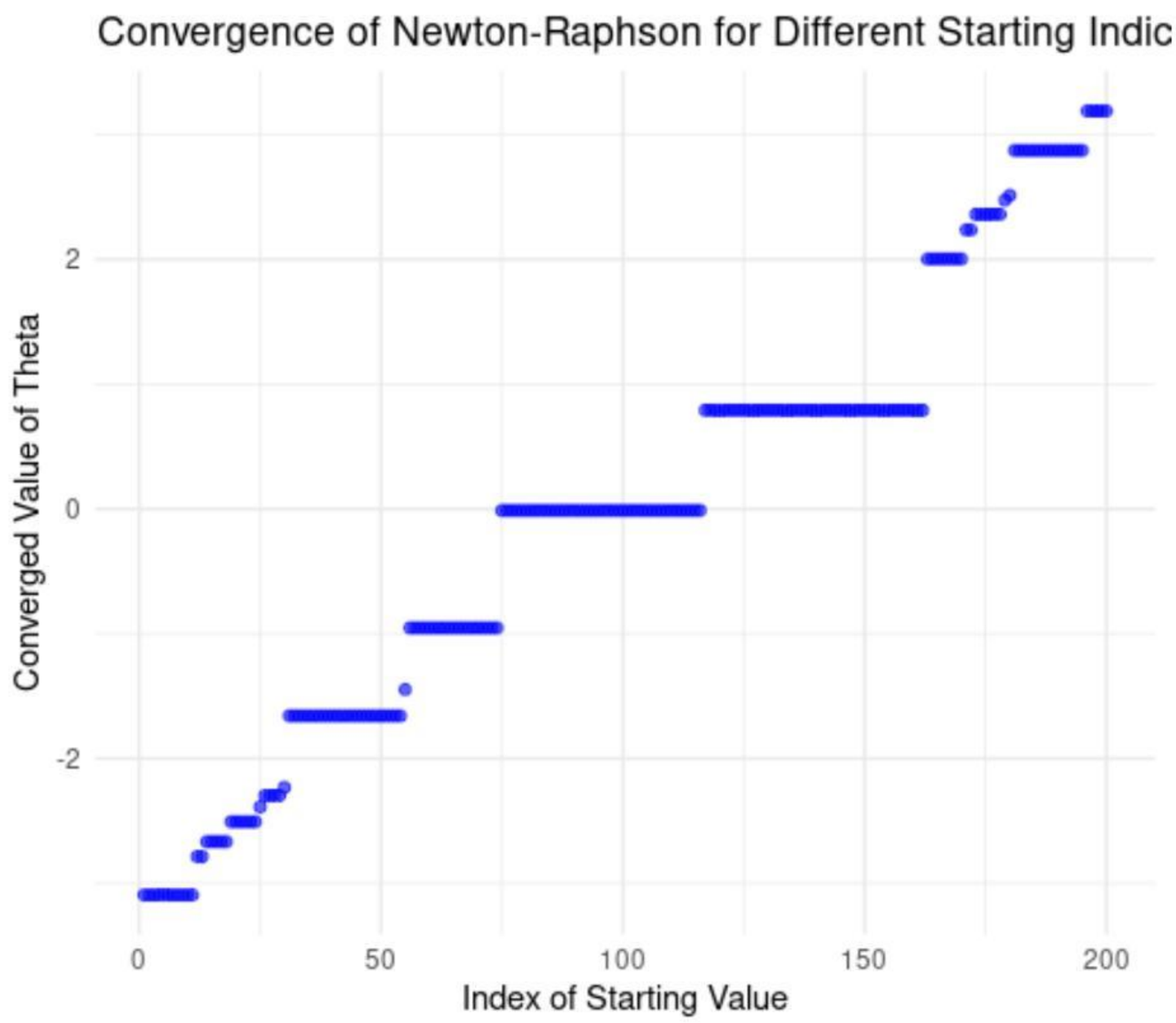

Figure 4.8: Convergence plot based on partition index

```
e=-0.8231863071855
p<-run_newton_raphson_range(data,start=s,end=e, n_points = 1000)
p

#The result is
s-e
#[1] -1.828537e-13
```

This code provided also the convergence plot 4.10. Upon further enclosing the interval a new convergence value was created at -0.82, for which we could either continue to enclose the $[-1, -0.8]$ or the $[-0.8, -0.5]$, we chose to further enclose the first interval.

4.3 Part 3

Modify the code displayed for logistic regression (binomial responses) to create a Fisher-Scoring algorithm that produces parameter estimates and confidence intervals for the Poisson regression model.

(a) Are there differences between Newton-Raphson and Fisher-Scoring in this case?

(b) Compare your results versus the ones obtained from the glm function for the following data:

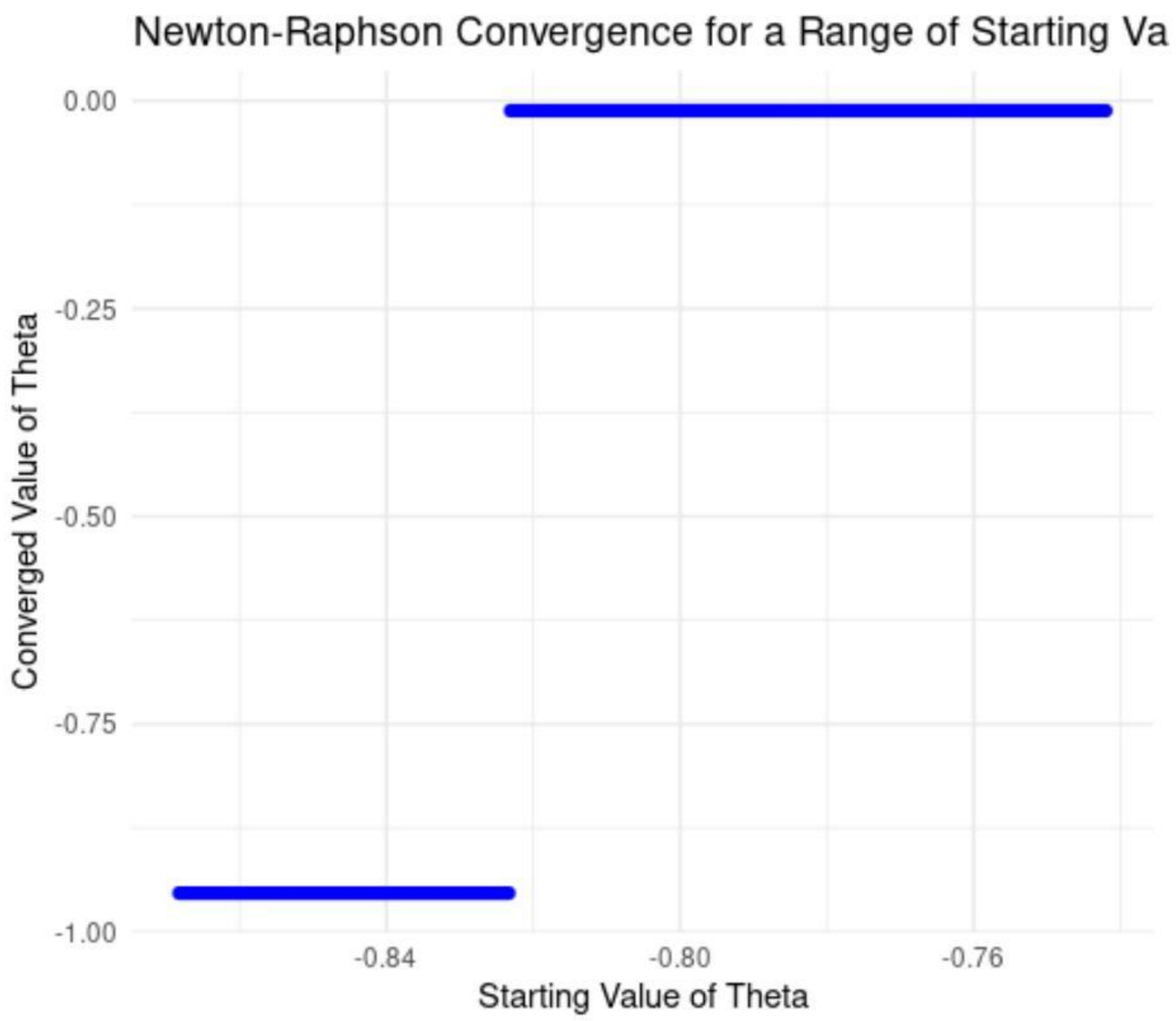

Figure 4.9: Initial 2 Converging values plot

4.4 a) Newton Raphson vs Fisher Scoring

The Data

```
counts <- c(18, 17, 15, 20, 10, 20, 25, 13, 12)
outcome <- gl(3, 1, 9)
treatment <- gl(3, 3)
d.AD <- data.frame(treatment, outcome, counts)
```

Fisher Scoring

```
fisher_scoring <- function(formula, data, start = NULL, max_iter = 25, eps
↪  = 1e-8, conf_level = 0.95) {
  # The model matrix and response variable
  X <- model.matrix(formula, data)
  y <- model.response(model.frame(formula, data))

  # Initializing coefficients
  if (is.null(start)) {
    beta <- rep(0, ncol(X))
  } else {
    beta <- start
  }

  # Interational variables
  iter <- 1
  conv <- FALSE
```

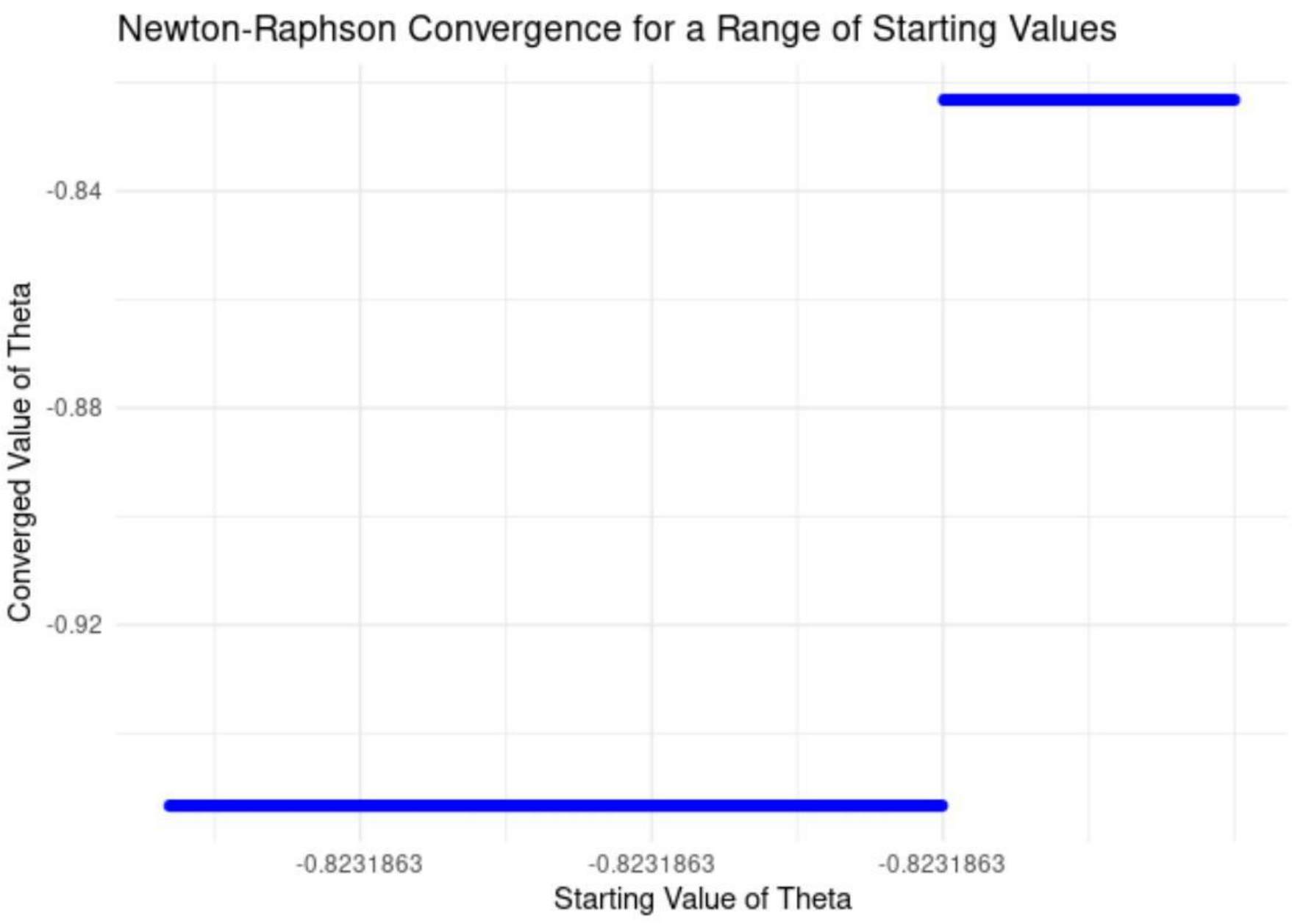

Figure 4.10: Final 2 Converging values plot

```r
while (iter <= max_iter && !conv) {
  # The linear predictor and mean
  eta <- X %*% beta
  mu <- exp(eta)

  # Score function
  score <- t(X) %*% (y - mu)

  # Fisher Information matrix
  W <- diag(as.vector(mu))
  fisher_info <- t(X) %*% W %*% X

  # Update coefficients using the Fisher Scoring update rule
  beta_new <- beta + solve(fisher_info, score)

  # Check convergence
  if (max(abs(beta_new - beta)) < eps) {
    conv <- TRUE
  }

  # Update beta for the next iteration
  beta <- beta_new
  iter <- iter + 1
}
```

```r
if (iter > max_iter) {
  warning('Fisher Scoring algorithm did not converge')
}

# After convergence, standard errors
fisher_info_inv <- solve(fisher_info)
std_errors <- sqrt(diag(fisher_info_inv))

# Confidence intervals
z_val <- qnorm(1 - (1 - conf_level) / 2)
  conf_intervals <- cbind(beta - z_val * std_errors, beta + z_val *
  ↪  std_errors)

  # The return list
  return(list(coefficients = beta, std.errors = std_errors, conf.intervals
  ↪  = conf_intervals))
}
# The fit for our given data
fit <- fisher_scoring(counts ~ outcome + treatment, data = d.AD)
```

yielding results:

```
> fit$coefficients
                    [,1]
(Intercept)  3.044522e+00
outcome2    -4.542553e-01
outcome3    -2.929871e-01
treatment2   2.063846e-16
treatment3  -1.906872e-16
> fit$conf.intervals
                   [,1]        [,2]
(Intercept)  2.7095672  3.37947764
outcome2    -0.8505027 -0.05800787
outcome3    -0.6707552  0.08478093
treatment2  -0.3919928  0.39199280
treatment3  -0.3919928  0.39199280
```

Based on these variables the Intercept dominates in magnitude all other predictors and a variable elimination procedure could provide us with a more parsimonious model nevertheless, that is not our goal in this exercise. Now to answer if there are differences between Newton-Raphson and Fisher-Scoring in this case.

In this case, we used the approximation of the Fisher Information Matrix we presented in Part 1. The Fisher information matrix is approximated by the Hessian which means that in this approximating framework, NR and Fisher-Scoring would yield the same results. Though an actual estimation of the Fisher Information matrix could be made by integration of the mean value, either using a Monte Carlo-like procedure or solving analytically the computational cost and time will be extensive, that is why we omit those calculations (see Appendix.A)

4.4.1 b. Comparing with glm function results

```
# The glm function
glm.D93 <- glm(counts ~ outcome + treatment, family = poisson(), data =
↪  d.AD)

# The confidence intervals for the glm model
glm_conf_intervals <- confint(glm.D93, level = 0.95)
```

with resulted coefficients and Confidence Intervals:

```
#Coefficients taken from Summary
Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept)  3.045e+00  1.709e-01  17.815   <2e-16 ***
outcome2    -4.543e-01  2.022e-01  -2.247   0.0246 *
outcome3    -2.930e-01  1.927e-01  -1.520   0.1285
treatment2   1.189e-15  2.000e-01   0.000   1.0000
treatment3   8.438e-16  2.000e-01   0.000   1.0000

>glm_conf_intervals
                 2.5 %      97.5 %
(Intercept)  2.6958215  3.36655581
outcome2    -0.8577018 -0.06255840
outcome3    -0.6753696  0.08244089
treatment2  -0.3932548  0.39325483
treatment3  -0.3932548  0.39325483
```

The results in both the glm and Fisher are pretty similar with no strong indications of one being better than the other.

4.5 Part 4

The attached csv files yvec and Xmat correspond, respectively to a response variable and a matrix of predictors. The dataset relates to a real application that we tackle these days.

(a) Your task is to derive a predictive model for yvec, which performs well in terms of MAE

(b) Report the alternative models you developed for the above task and their MAE performance in a 50-fold CV experiment.

Feel free to include interaction terms. Prioritize methodologies that were discuss in class, but feel free to use any alternative method that you know and you believe is worth examining for this task

4.5.1 The Implementations

All second-order interactions of predictors where omitted cause there was no apparent relationship of the predictors from the heatmap 4.11 (to choose some) and the computations below, using all the second-order predictors, would have been impossible, thus all the models are fitted on the initial Xmat columns.

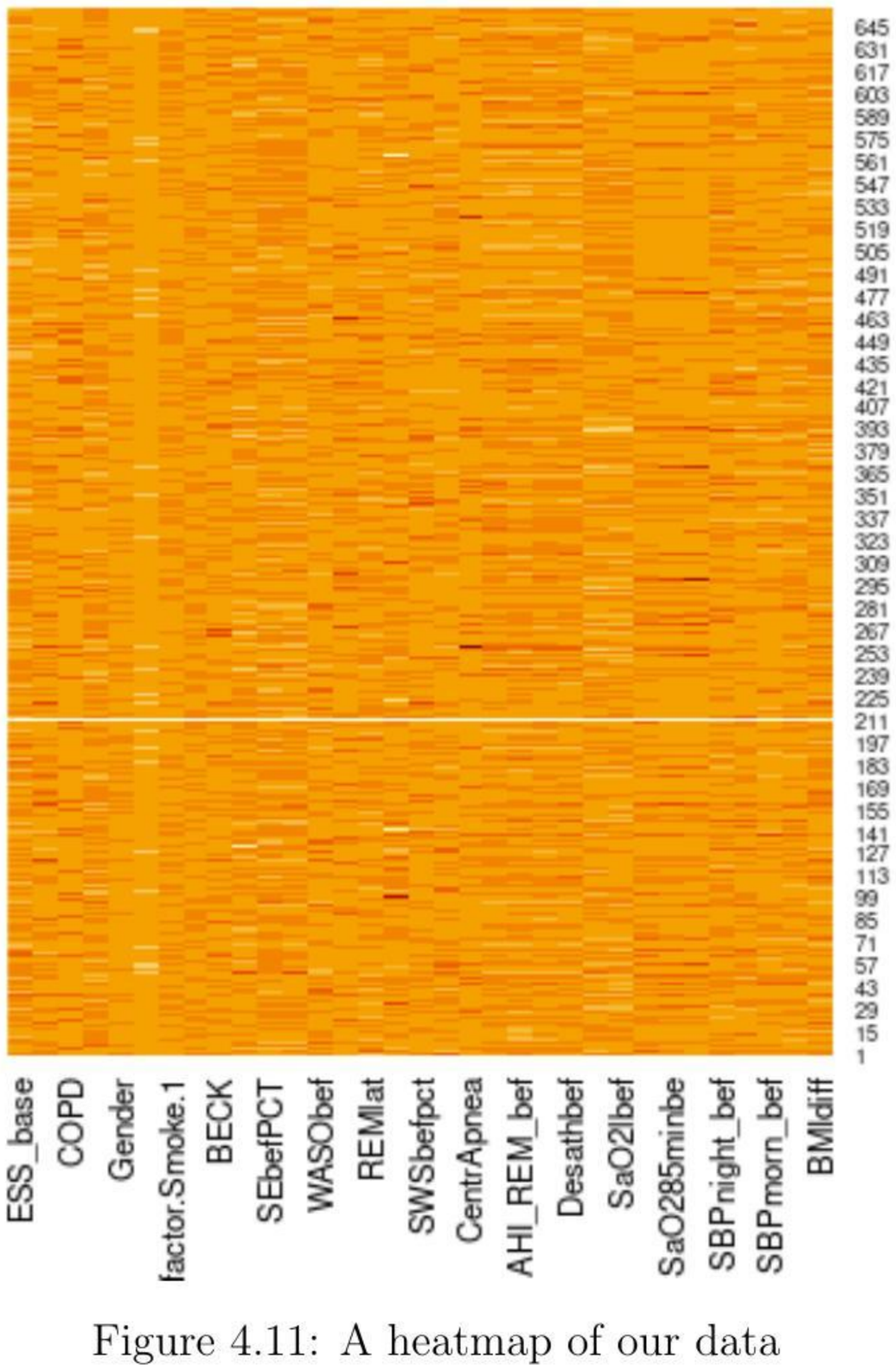

Figure 4.11: A heatmap of our data

LASSO

```
#lasso
library(glmnet)

yvec_vector <- yvec$target
Xmat_matrix <- as.matrix(Xmat)


set.seed(123)
cv_model <- cv.glmnet(Xmat_matrix, yvec_vector, alpha = 1, family =
↪  "gaussian")


best_lambda <- cv_model$lambda.min
lasso_model <- glmnet(Xmat_matrix, yvec_vector, alpha = 1, lambda =
↪  best_lambda)


lasso_coefficients <- coef(lasso_model, s = best_lambda)

# Storing the coefficients in a named vector for easy reference
```

```
lasso_coefficients_vector <- as.numeric(lasso_coefficients[-1])  # Exclude
↪  the intercept
names(lasso_coefficients_vector) <- rownames(lasso_coefficients)[-1])
```

The best LASSO model was found to have the below coefficients:

```
> lasso_coefficients_vector
   (Intercept)       ESS_base            BMI           COPD     HoursPweek
    ↪  Gender
 -7.4386682998   0.9919919249   0.0000000000 -1.0467091397   0.0243070885
↪  -0.1286604997
     Marital factor.Smoke.1 factor.Smoke.2           BECK       TRT_bef
      ↪  SEbefPCT
 1.2884012840   0.0000000000   0.0000000000 -0.0099450632   0.0031734960
↪  0.0022458783
      TSTbef       WASObef          N2bef        REMlat       NREMbef
       ↪  SWSbefpct
 0.0000000000   0.0000000000 -0.0051931074   0.0000000000   0.0000000000
↪  0.0000000000
  REM_befpct     CentrApnea        AHI_bef    AHI_REM_bef     X.rousalnd
   ↪  Desathbef
 0.0139493116 -0.1353469861   0.0000000000 -0.0018613470   0.0000000000
↪  0.0000000000
    SaO2mbef       SaO2lbef     SaO2minbef    SaO285minbe    SaO280minbe
     ↪  SBPnight_bef
  0.0000000000   0.0373119894 -0.0030818215   0.0000000000 -0.0276225128
  ↪  -0.0486847433
   DBPnight_be    SBPmorn_bef    DBPmorn_bef         BMIdiff
  0.0235599928 -0.0009092007   0.0031016962   0.0091384014
```

the intercept term had the greatest magnitude -7.4386682998 and as we can see based on LASSO it affects our model the most.

Ridge

```
# Load the required library
library(glmnet)

Xmat_matrix <- as.matrix(Xmat)
yvec_vector <- yvec$target

# Fit the Ridge
ridge_model <- glmnet(Xmat_matrix, yvec_vector, alpha = 0)

# CV for the best lambda
cv_ridge_model <- cv.glmnet(Xmat_matrix, yvec_vector, alpha = 0)

# The coefficients for the best lambda
ridge_coefficients <- coef(cv_ridge_model, s = "lambda.min")
```

with resulted coefficients:

```
> print(ridge_coefficients)
34 x 1 sparse Matrix of class "dgCMatrix"
                          s1
(Intercept)     2.6411585409
ESS_base        0.9354978516
BMI            -0.0278733220
COPD           -1.1356422195
HoursPweek      0.0309309730
Gender         -0.3989954980
Marital         1.9350019239
factor.Smoke.1 -0.0343499440
factor.Smoke.2 -0.1384327852
BECK           -0.0176945311
TRT_bef         0.0089313976
SEbefPCT        0.0257562011
TSTbef         -0.0045440966
WASObef        -0.0007956993
N2bef          -0.0106239128
REMlat          0.0015512008
NREMbef        -0.0123977225
SWSbefpct      -0.0112371658
REM_befpct      0.0543476313
CentrApnea     -0.2016378628
AHI_bef         0.0114955758
AHI_REM_bef    -0.0148972302
X.rousalnd      0.0022176170
Desathbef       0.0078105833
SaO2mbef       -0.1354295588
SaO2lbef        0.0601207171
SaO2minbef     -0.0052483164
SaO285minbe    -0.0033489867
SaO280minbe    -0.0255234467
SBPnight_bef   -0.0498488904
DBPnight_be     0.0313847236
SBPmorn_bef    -0.0193932386
DBPmorn_bef     0.0229684016
BMIdiff         0.0423171912
```

As we can see all of the predictors survived the elimination as we expected from the Ridge, with the most dominant being the intercept with magnitude 2.6411585409, and the predictor that seemed to affect our model the most was the "Martial" with a magnitude of 1.9350019239

Best Subset

```
library(leaps)
library(glmnet)

yvec_vector <- yvec#£target  # Assuming yvec is a dataframe
Xmat_matrix <- as.matrix(Xmat)  # Convert Xmat to a matrix if it's not
↪  already
```

```
# Fit best subset regression
best_subset_fit <- regsubsets(yvec ~ ., data = Xmat, nvmax = 10) # Adjust
↪  'nvmax' as needed

# Summarize the best models for each number of variables
subset_summary <- summary(best_subset_fit)

# Identify the best model based on the lowest BIC
best_model_bic <- which.min(subset_summary$bic)

# Coefficients of the best model
best_model_bic_coefs <- coef(best_subset_fit, id = best_model_bic)

# Store the coefficients
best_subset_bic_coefs <- best_model_coefs
```

with remaining predictors:

```
> best_subset_bic_coefs
 (Intercept)     ESS_base         COPD   HoursPweek  SaO280minbe
 ↪  SBPnight_bef
-0.01829338   1.02762821  -1.71592677   0.03091128  -0.05273760
 ↪  -0.05389675
```

F-B_ AIC and BIC

```
Xmat_matrix <- as.matrix(Xmat)
yvec_vector <- yvec$target

combined_data <- cbind(Xmat, yvec)

forward_aic_fit <- step(lm(yvec ~ 1, data = combined_data),
                        scope = list(lower = ~1, upper = ~.),
                        direction = "forward",
                        trace = FALSE,
                        k = 2)
forward_aic_coefs <- coef(forward_aic_fit)

forward_bic_fit <- step(lm(yvec ~ 1, data = combined_data),
                        scope = list(lower = ~1, upper = ~.),
                        direction = "forward",
                        trace = FALSE,
                        k = log(nrow(combined_data)))
forward_bic_coefs <- coef(forward_bic_fit)

backward_aic_fit <- step(lm(yvec ~ ., data = combined_data),
                         direction = "backward",
                         trace = FALSE,
                         k = 2)
backward_aic_coefs <- coef(backward_aic_fit)
```

```
backward_bic_fit <- step(lm(yvec ~ ., data = combined_data),
                         direction = "backward",
                         trace = FALSE,
                         k = log(nrow(combined_data)))
backward_bic_coefs <- coef(backward_bic_fit)
```

with results:

```
> forward_aic_coefs
(Intercept)
   3.703647

> forward_bic_coefs
(Intercept)
   3.703647

> backward_aic_coefs
 (Intercept)     ESS_base         COPD   HoursPweek      Marital
 ↪  TRT_bef     SEbefPCT
  1.21579948   1.01995773  -1.39408586   0.03128183   1.86063670
  ↪  0.01797355   0.11444807
      TSTbef      AHI_bef  AHI_REM_bef     SaO2mbef     SaO2lbef
       ↪  SaO2minbef  SaO280minbe
 -0.02055101   0.02324273  -0.01804969  -0.21900877   0.08519438
  ↪  -0.00800668  -0.03478945
SBPnight_bef  DBPnight_be
 -0.06931650   0.04448021

> backward_bic_coefs
 (Intercept)     ESS_base         COPD   HoursPweek  SaO280minbe
  ↪  SBPnight_bef
 -0.01829338   1.02762821  -1.71592677   0.03091128  -0.05273760
  ↪  -0.05389675
```

4.5.2 b. The 50-fold CV

Everything is in the below code:

```
# Necessary libraries
library(caret)
library(glmnet)
library(leaps)

# The data prep
yvec_vector <- yvec  # Assuming yvec is a dataframe
Xmat_matrix <- as.matrix(Xmat)  # Convert Xmat to a matrix if it's not
 ↪  already
combined_data <- cbind(Xmat, yvec = yvec_vector)
folds <- createFolds(yvec_vector, k = 50)
```

```r
# Function to calculate MAE
calc_mae <- function(predicted, actual) {
  mean(abs(predicted - actual))
}

# A vector to store the mean MAE of each model
model_names <- c("lasso", "ridge", "best_subset", "forward_aic",
↪  "forward_bic", "backward_aic", "backward_bic")
mean_maes <- setNames(numeric(length(model_names)), model_names)

# Loop over each fold for CV
for (model_name in model_names) {
  fold_maes <- numeric(length(folds))

  for (i in seq_along(folds)) {
    # Splitting the data
    train_indices <- setdiff(seq_len(nrow(Xmat_matrix)), folds[[i]])
    test_indices <- folds[[i]]
    X_train <- Xmat_matrix[train_indices, ]
    y_train <- yvec_vector[train_indices]
    X_test <- Xmat_matrix[test_indices, ]
    y_test <- yvec_vector[test_indices]

    # Fiting the model based on name
    if (model_name == "lasso") {
      cv_model <- cv.glmnet(X_train, y_train, alpha = 1)
      best_lambda <- cv_model$lambda.min
      fit_model <- glmnet(X_train, y_train, alpha = 1, lambda =
      ↪  best_lambda)
      predictions <- predict(fit_model, newx = X_test, s = best_lambda)
    } else if (model_name == "ridge") {
      cv_ridge_model <- cv.glmnet(X_train, y_train, alpha = 0)
      best_lambda <- cv_ridge_model$lambda.min
      fit_model <- glmnet(X_train, y_train, alpha = 0, lambda =
      ↪  best_lambda)
      predictions <- predict(fit_model, newx = X_test, s = best_lambda)
    } else if (model_name == "best_subset") {
      best_subset_fit <- regsubsets(yvec ~ ., data =
      ↪  combined_data[train_indices, ], nvmax = 10)
      best_model <- which.min(summary(best_subset_fit)$bic)
      fit_model <- lm(formula = as.formula(paste("yvec ~",
      ↪  paste(names(coef(best_subset_fit, id = best_model))[-1],
      ↪  collapse = "+"))),
                      data = combined_data[train_indices, ])
      predictions <- predict(fit_model, newdata =
      ↪  combined_data[test_indices, ])
    } else if (model_name == "forward_aic" || model_name == "forward_bic")
    ↪  {
      direction <- "forward"
      k_value <- ifelse(model_name == "forward_aic", 2,
      ↪  log(nrow(combined_data)))
```

```
    stepwise_fit <- step(lm(yvec ~ 1, data =
    ↪  combined_data[train_indices, ]),
                         direction = direction,
                         scope = list(lower = ~1, upper = ~.),
                         trace = FALSE,
                         k = k_value)
    predictions <- predict(stepwise_fit, newdata =
    ↪  combined_data[test_indices, ])
  } else if (model_name == "backward_aic" || model_name ==
  ↪  "backward_bic") {
    direction <- "backward"
    k_value <- ifelse(model_name == "backward_aic", 2,
    ↪  log(nrow(combined_data)))
    stepwise_fit <- step(lm(yvec ~ ., data =
    ↪  combined_data[train_indices, ]),
                         direction = direction,
                         trace = FALSE,
                         k = k_value)
    predictions <- predict(stepwise_fit, newdata =
    ↪  combined_data[test_indices, ])
  }

  # MAE for this fold
  fold_maes[i] <- calc_mae(predictions, y_test)
}

  # The mean MAE for this model
  mean_maes[model_name] <- mean(fold_maes, na.rm = TRUE)
}
```

with resulted Mean MAE over the 50-folds CV:

```
> print(mean_maes)
      lasso       ridge  best_subset  forward_aic  forward_bic
      ↪  backward_aic backward_bic
   3.970452    3.976690    3.974529    5.852230    5.852230
   ↪  4.023168    3.962912
```

4.5.3 a. The best MAE performing predictive model

Based on MAE the Backward_bic is the best, though LASSO came close. The B_bic additionally provides a better relation to the remaining predictors enabling us to assess the situation quickly and get a pretty accurate initial guess. Though there might be more parsimonious second-order models we failed to optically observe.

Appendix A

Appendix Title

A.

The following handwritten notes illustrate the computation of the derivatives of the Cauchy densities log-likelihood (see A.1 and A.2), the logistic densities log-likelihood (see A.3 and A.4), the part2 densities log-likelihood (see A.4) and the Fisher Information Matrix calculations (see A.4, A.5 and A.6)

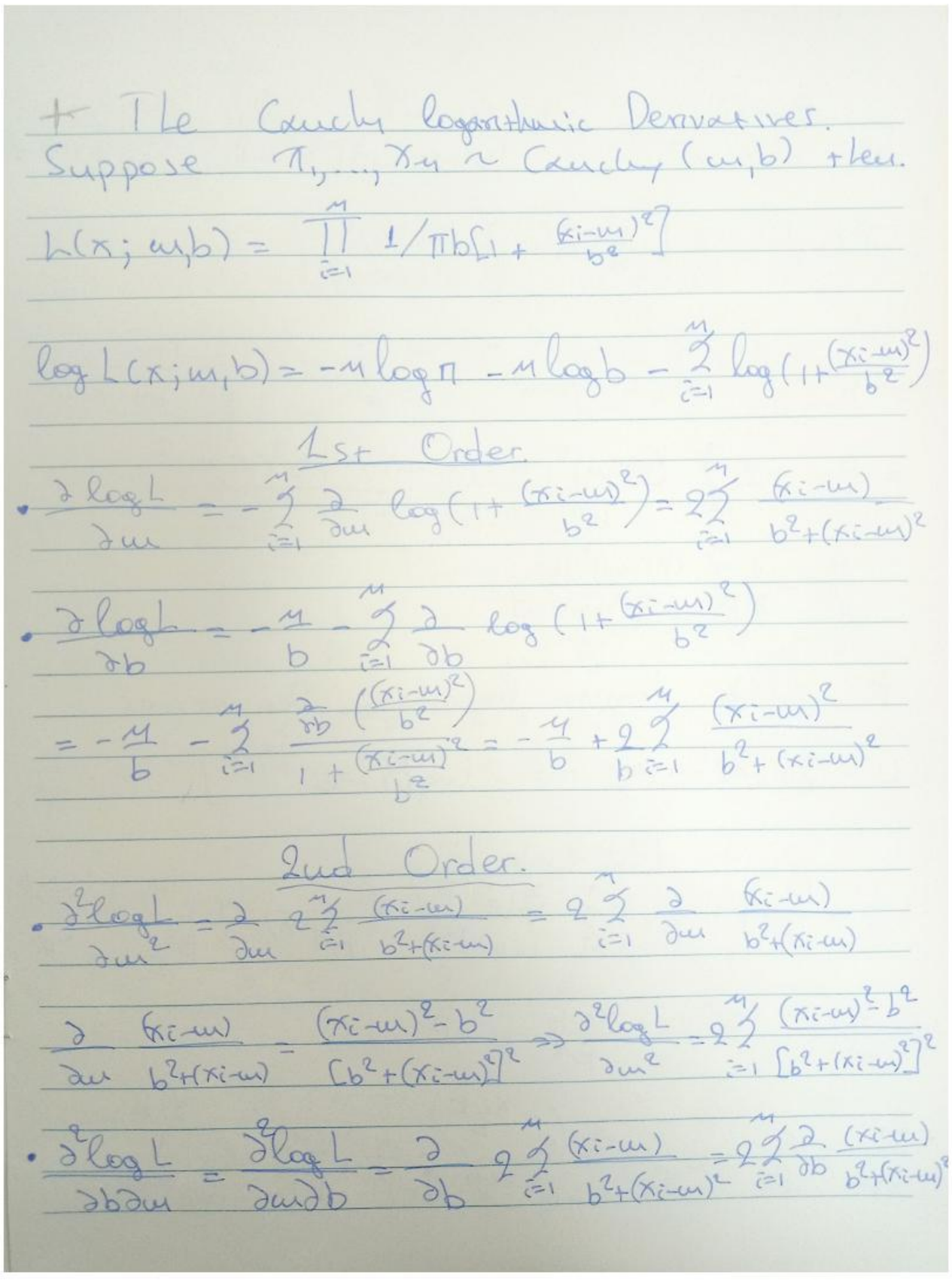

Figure A.1: The Cauchy Derivatives 1

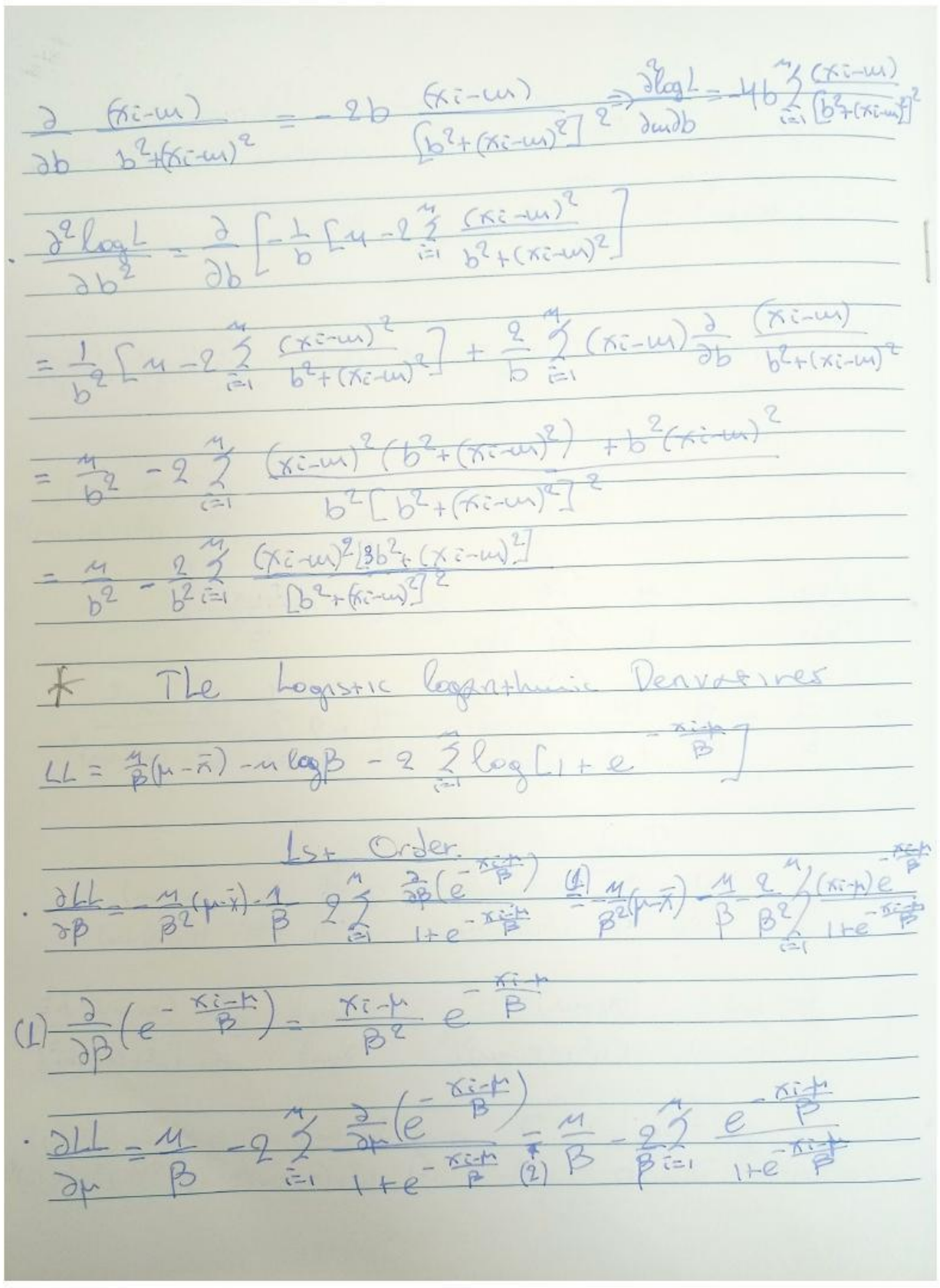

Figure A.2: The Cauchy Derivatives 2

B. The code for estimating the integral for the logistic $I(\theta)_{1,1}$

```
from sympy import symbols, integrate, exp

# Define the symbols
x, mu, beta = symbols('x mu beta')

# Define the expression to integrate
expression = exp(-2*(x-mu)/beta) / (1 + exp(-(x-mu)/beta))**4

# Perform the integration
integral_result = integrate(expression, x)

integral_result
```

The code for estimating the integral for the logistic $I(\theta)_{1,2}$

```
from sympy import symbols, integrate, exp

# Define the symbols
x, mu, beta = symbols('x mu beta')
```

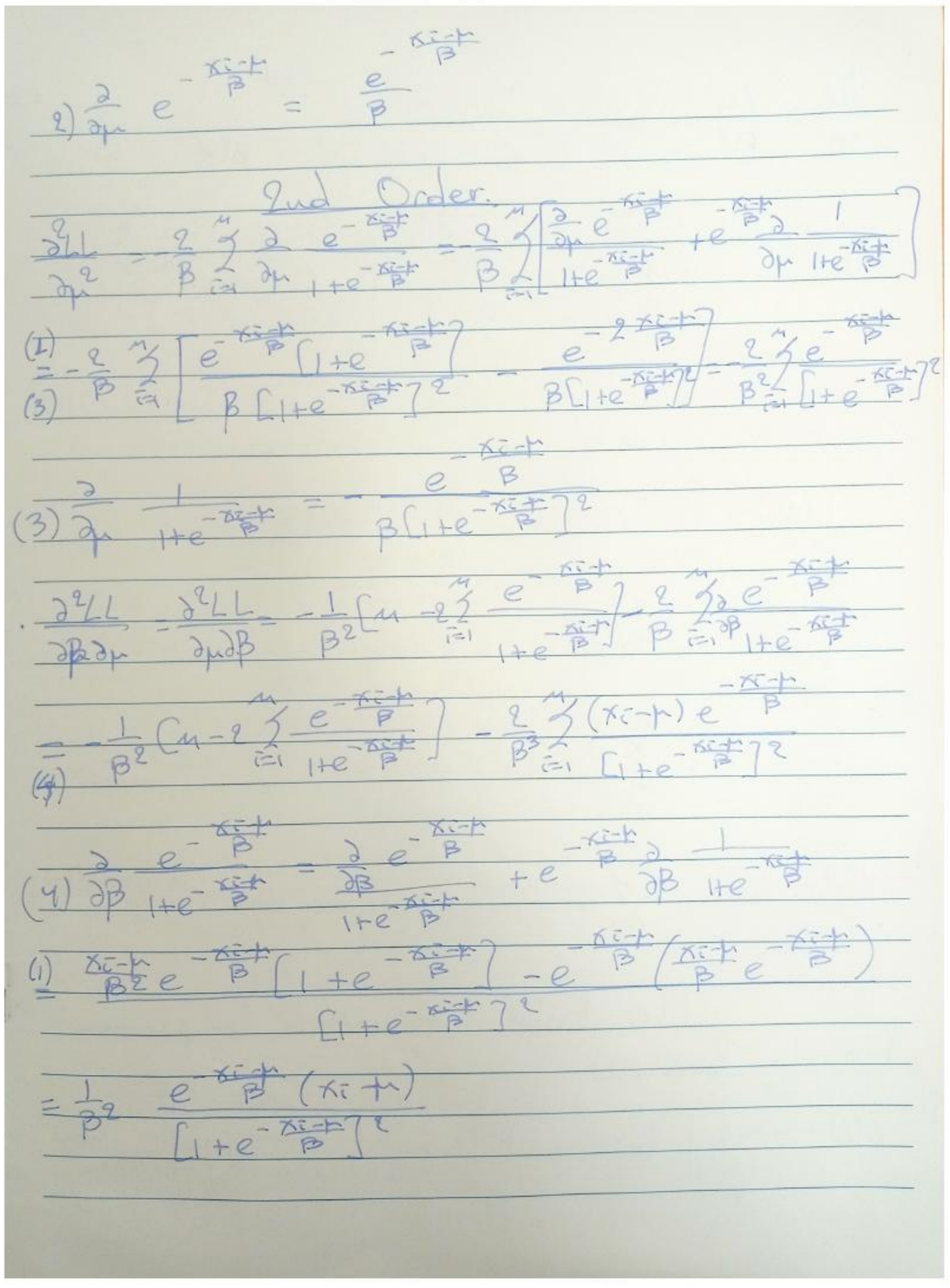

Figure A.3: The Logistic Derivatives 1

```
# Define the expression to integrate
expression = exp(-2*(x-mu)/beta) / (1 + exp(-(x-mu)/beta))**3

# Perform the integration
integral_result = integrate(expression, x)

integral_result
```

```
#and the again


# Define the symbols
x, mu, beta = symbols('x mu beta')

# Define the expression to integrate
expression = (x-mu)*exp(-2*(x-mu)/beta) / (1 + exp(-(x-mu)/beta))**4
```

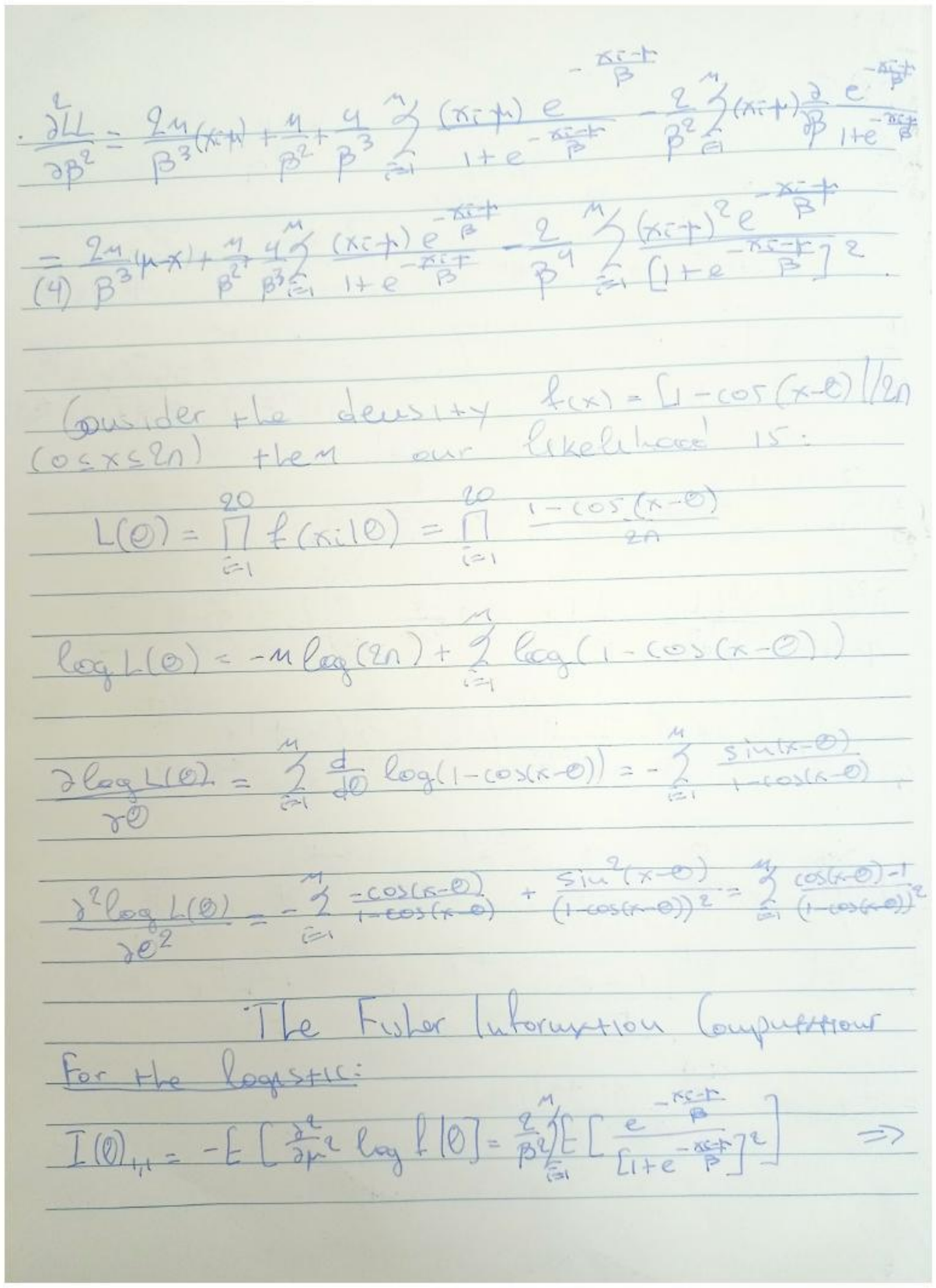

Figure A.4: The part2 densities derivatives

```
# Perform the integration
integral_result = integrate(expression, x)

integral_result
```

Similarly for $I_{2,2}$

The code for the pair and scatter plots for predictability

```
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
import pandas as pd

df=pd.read_csv("Xmat.csv")
sns.pairplot(df)


sns.pairplot(df,hue="target")
```

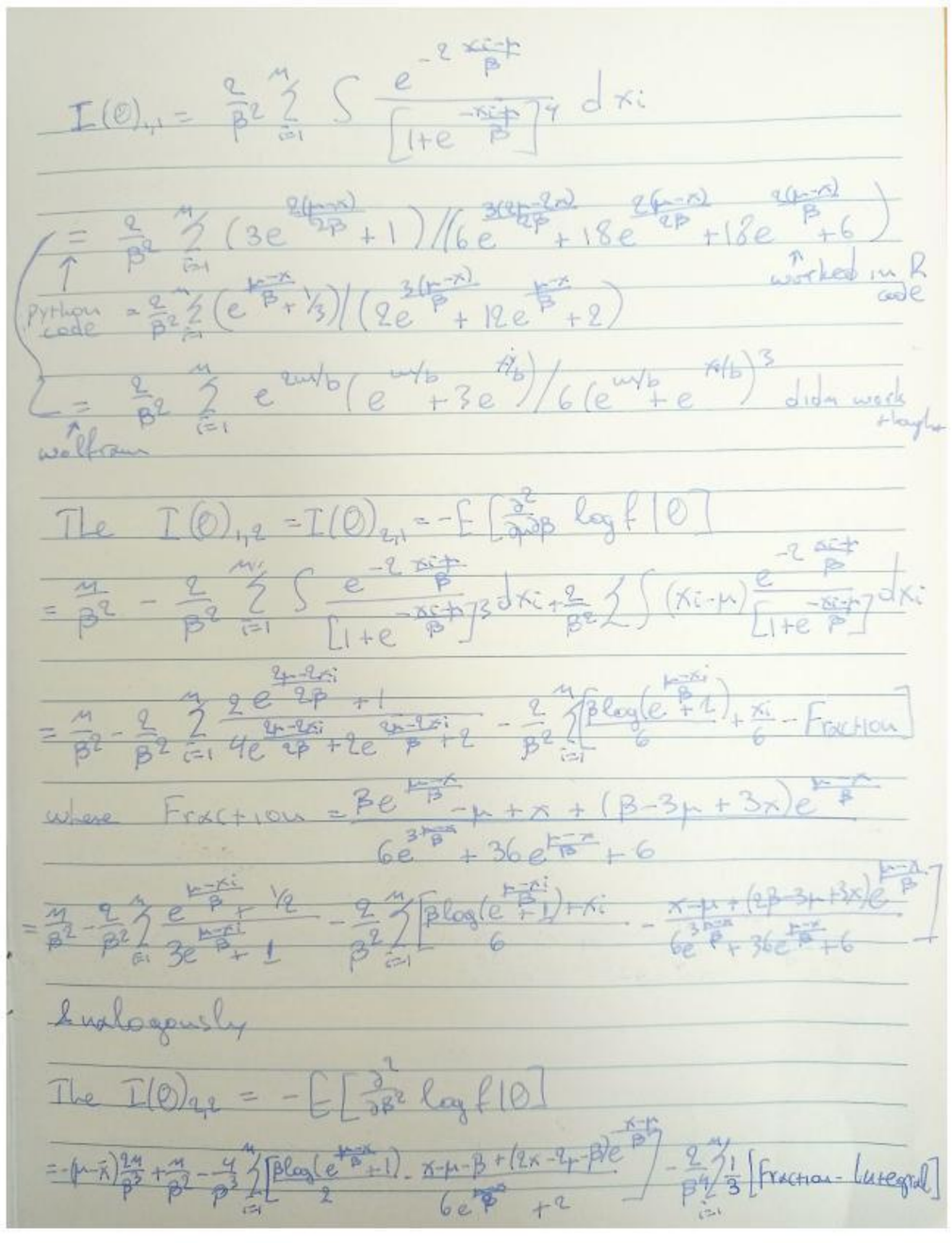

Figure A.5: The Fisher Information matrix computations 1

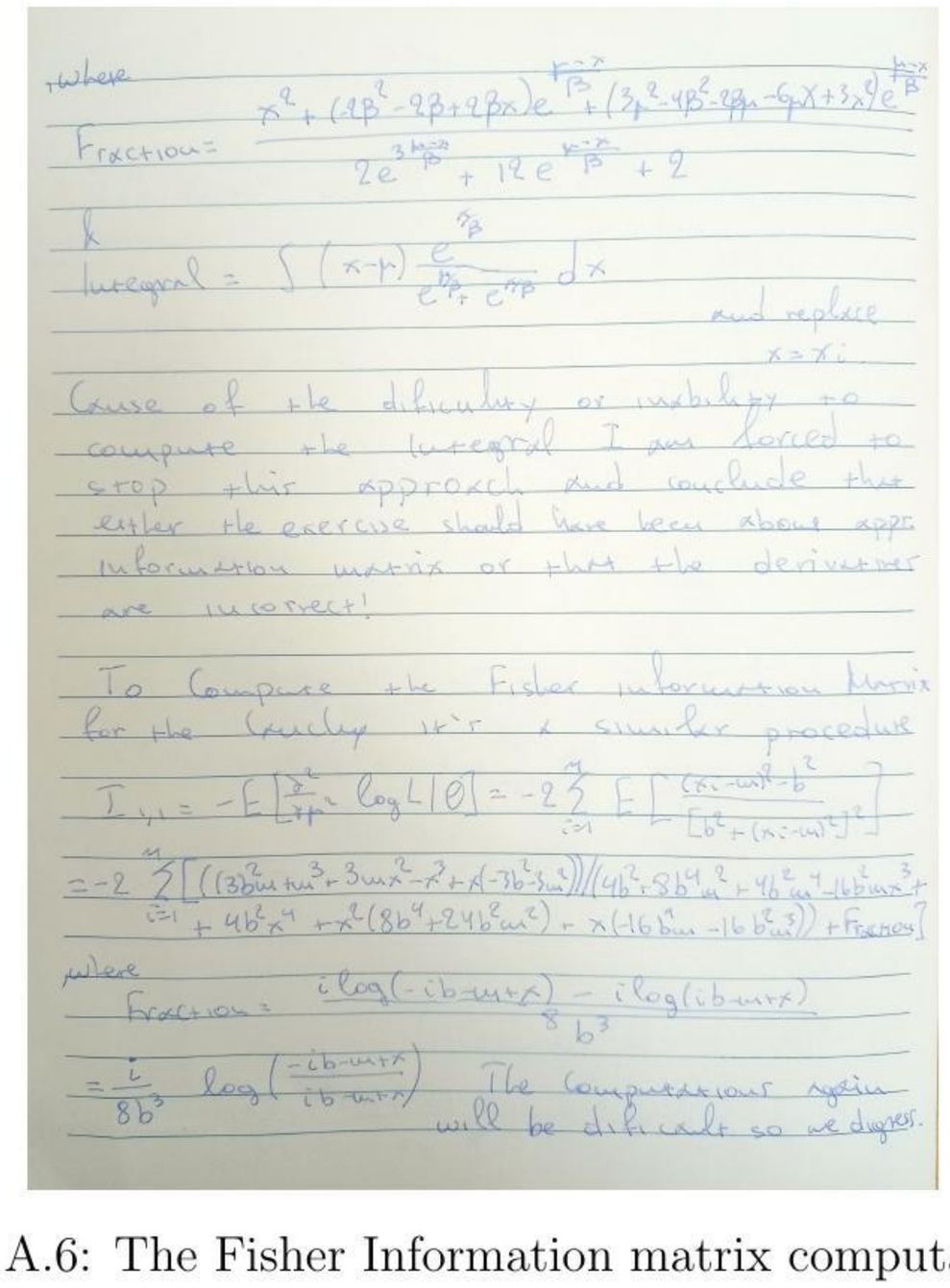

Figure A.6: The Fisher Information matrix computations 2