Probability, Computation and Simulation Homework 6

Brian Cervantes Alvarez

November 26, 2024

Problem 2.5

I loaded the provided oil spill data into R.

Part A

Given $N_i \sim \text{Poisson}(\lambda_i)$, where $\lambda_i = \alpha_1 b_{i1} + \alpha_2 b_{i2}$.

We can calculate the log-likelihood function as follows,

$$\ell(\alpha_1, \alpha_2) = \sum_i \left[N_i \log(\lambda_i) - \lambda_i - \log(N_i!)\right]$$

Next, we can determine the gradient a.k.a. the score function. It is given below,

$$\frac{\partial \ell}{\partial \alpha_j} = \sum_i \left(\frac{N_i}{\lambda_i} - 1\right) b_{ij}, \quad j = 1, 2$$

Then, we can express the Hessian Matrix by,

$$\frac{\partial^2 \ell}{\partial \alpha_j \partial \alpha_k} = -\sum_i \left(\frac{N_i}{\lambda_i^2}\right) b_{ij} b_{ik}, \quad j, k = 1, 2$$

Hence, we arrive to the Newton–Raphson update rule,

$$\alpha^{(t+1)} = \alpha^{(t)} - \left[\nabla^2 \ell(\alpha^{(t)})\right]^{-1} \nabla \ell(\alpha^{(t)})$$

Part B

We can determine the Fisher Scoring update ruling by finding the expected information matrix,

$$I_{jk}(\alpha) = -\mathrm{E}\left[\frac{\partial^2 \ell}{\partial \alpha_j \partial \alpha_k}\right] = \sum_i \left(\frac{b_{ij} b_{ik}}{\lambda_i}\right), \quad j, k = 1, 2$$

Therefore, our Fisher Scoring update rule is,

$$\alpha^{(t+1)} = \alpha^{(t)} + \left[I(\alpha^{(t)})\right]^{-1} \nabla \ell(\alpha^{(t)})$$

Part C

Newton Raphson Algorithm

```r
newtonRaphson <- function(ni, b1, b2,
                          alphaInit, tol = 1e-8, maxIter = 100) {
  alpha <- alphaInit
  path <- list(alpha)
  for (iter in 1:maxIter) {
    lambdaI <- alpha[1] * b1 + alpha[2] * b2
    grad <- c(
      sum((ni / lambdaI - 1) * b1),
      sum((ni / lambdaI - 1) * b2)
    )
    hessian <- matrix(c(
      -sum(ni * b1^2 / lambdaI^2),
      -sum(ni * b1 * b2 / lambdaI^2),
      -sum(ni * b1 * b2 / lambdaI^2),
      -sum(ni * b2^2 / lambdaI^2)
    ), nrow = 2)
    delta <- solve(hessian, grad)
    alphaNew <- alpha - delta
    path <- append(path, list(alphaNew))
    if (max(abs(alphaNew - alpha)) < tol) break
    alpha <- alphaNew
  }
  list(alpha = alpha, iterations = iter, path = path)
}
```

Fisher Scoring Algorithm

```
fisherScoring <- function(ni, b1, b2,
                          alphaInit, tol = 1e-8, maxIter = 100) {
  alpha <- alphaInit
  path <- list(alpha)
  for (iter in 1:maxIter) {
    lambdaI <- alpha[1] * b1 + alpha[2] * b2
    grad <- c(
      sum((ni / lambdaI - 1) * b1),
      sum((ni / lambdaI - 1) * b2)
    )
    fisherInfo <- matrix(c(
      sum(b1^2 / lambdaI),
      sum(b1 * b2 / lambdaI),
      sum(b1 * b2 / lambdaI),
      sum(b2^2 / lambdaI)
    ), nrow = 2)
    delta <- solve(fisherInfo, grad)
    alphaNew <- alpha + delta
    path <- append(path, list(alphaNew))
    if (max(abs(alphaNew - alpha)) < tol) break
    alpha <- alphaNew
  }
  list(alpha = alpha, iterations = iter, path = path)
}
```

Results

```
Newton-Raphson MLEs: 1.097153 0.9375546

Iterations: 4

Fisher Scoring MLEs: 1.097153 0.9375546

Iterations: 16

[1] 1.0971525 0.9375546

[1] 1.0971525 0.9375546
```

Comparison:

- **Ease of Implementation:** Both methods are similar in code structure.
- **Performance:** The number of iterations to converge may vary. Fisher scoring might require more iterations due to using the *expected information matrix*.

Part D

```r
# Use the Fisher Information Matrix at the MLEs
lambdaI_mle <- nrResult$alpha[1] * b1 + nrResult$alpha[2] * b2
fisherInfoMLE <- matrix(c(
  sum(b1^2 / lambdaI_mle),
  sum(b1 * b2 / lambdaI_mle),
  sum(b1 * b2 / lambdaI_mle),
  sum(b2^2 / lambdaI_mle)
), nrow = 2)

# Covariance Matrix
covMatrix <- solve(fisherInfoMLE)
```

```
Standard Errors: 0.437556 0.6314687
```

Part E

```r
steepestAscent <- function(ni, b1, b2,
                          alphaInit, tol = 1e-8, maxIter = 100) {
  alpha <- alphaInit
  path <- list(alpha)
  for (iter in 1:maxIter) {
    lambdaI <- alpha[1] * b1 + alpha[2] * b2
    grad <- c(
      sum((ni / lambdaI - 1) * b1),
      sum((ni / lambdaI - 1) * b2)
    )
    direction <- grad
    stepSize <- 1
    repeat {
      alphaNew <- alpha + stepSize * direction
      lambdaNew <- alphaNew[1] * b1 + alphaNew[2] * b2
      if (all(lambdaNew > 0)) {
        llOld <- sum(ni * log(lambdaI) - lambdaI)
        llNew <- sum(ni * log(lambdaNew) - lambdaNew)
        if (llNew > llOld) break
      }
      stepSize <- stepSize / 2
      if (stepSize < tol) break
    }
    path <- append(path, list(alphaNew))
    if (max(abs(alphaNew - alpha)) < tol) break
    alpha <- alphaNew
  }
  list(alpha = alpha, iterations = iter, path = path)
}
```

```
Steepest Ascent MLEs: 1.097153 0.9375546
```

```
Iterations: 60
```

Part F

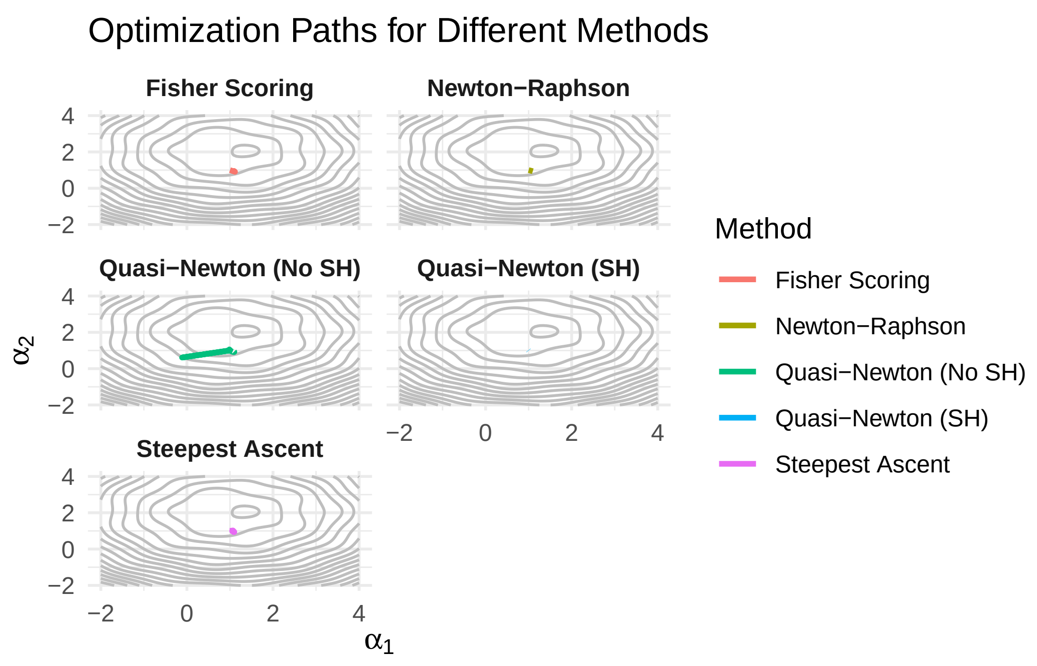
```r
quasiNewton <- function(ni, b1, b2, alphaInit,
                        tol = 1e-8, maxIter = 100, stepHalving = TRUE) {
  alpha <- alphaInit
  BInv <- diag(2)  # Initial inverse Hessian approximation
  path <- list(alpha)
  for (iter in 1:maxIter) {
    lambdaI <- alpha[1] * b1 + alpha[2] * b2
    grad <- c(
      sum((ni / lambdaI - 1) * b1),
      sum((ni / lambdaI - 1) * b2)
    )
    deltaAlpha <- -BInv %*% grad
    stepSize <- 1
    if (stepHalving) {
      repeat {
        alphaNew <- alpha + stepSize * deltaAlpha
        lambdaNew <- alphaNew[1] * b1 + alphaNew[2] * b2
        if (all(lambdaNew > 0)) {
          llOld <- sum(ni * log(lambdaI) - lambdaI)
          llNew <- sum(ni * log(lambdaNew) - lambdaNew)
          if (llNew > llOld) break
        }
        stepSize <- stepSize / 2
        if (stepSize < tol) break
      }
    } else {
      alphaNew <- alpha + deltaAlpha
    }
    s <- alphaNew - alpha
    lambdaI_new <- alphaNew[1] * b1 + alphaNew[2] * b2
    gradNew <- c(
      sum((ni / lambdaI_new - 1) * b1),
      sum((ni / lambdaI_new - 1) * b2)
    )
    y <- gradNew - grad
    rho <- 1 / sum(y * s)
    if (is.finite(rho)) {
      BInv <- (diag(2) - rho * s %*% t(y)) %*% BInv %*%
        (diag(2) - rho * y %*% t(s)) + rho * s %*% t(s)
    }
    path <- append(path, list(alphaNew))
    if (max(abs(alphaNew - alpha)) < tol) break
    alpha <- alphaNew
  }
  list(alpha = alpha, iterations = iter, path = path)
}
```

```
Quasi-Newton MLEs with step halving: 0.9957423 1.013556

Iterations: 3

Quasi-Newton MLEs without step halving: 1.097153 0.9375546

Iterations: 16
```

Comparison:

- **With Step Halving:** Converges reliably.
- **Without Step Halving:** Faster convergence but may risk instability.

Part G


Optimization Paths for Different Methods
Fisher Scoring
Newton−Raphson
Quasi−Newton (No SH)
Quasi−Newton (SH)
Steepest Ascent
4
2
0
−2
−2
0
2
4
α_2
α_1
Method
Fisher Scoring
Newton−Raphson
Quasi−Newton (No SH)
Quasi−Newton (SH)
Steepest Ascent