Computational & Mathematical Statistics - Fall 2023
Assignment 4

Newton-Raphson optimization; Fisher scoring; Maximum likelihood estimation

Ioannis Maris, math1p0004

University of Crete,
Department of Mathematics & Applied Mathematics,
Department of Computer Science,
FORTH - Foundation for Research & Technology - Institute of Applied and Computational Mathematics, Institute of Computer Science.

January 2024

Assignment 4 Part A

- 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). For the logistic distribution the corresponding expressions are shown in here
  - Derive the above-mentioned expressions.
  - 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.
  - 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.
  - 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.

MLE for Cauchy Distribution

Given a sample $X = \{x_1, x_2, \ldots, x_n\}$ from a Cauchy distribution with the location parameter m and scale parameter b known and set to 1 for simplicity, the likelihood function is given by:

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

The log-likelihood function is:

$$\ell(x; m) = \sum_{i=1}^{n} \log \left[\frac{1}{\pi(1 + (x_i - m)^2)} \right] = -n \log \pi - \sum_{i=1}^{n} \log[1 + (x_i - m)^2] \tag{2}$$

Taking the derivative leads to:

$$\frac{\partial}{\partial m} \ell(x; m) = \sum_{i=1}^{n} \frac{2(x_i - m)}{1 + (x_i - m)^2} \tag{3}$$

$$\frac{\partial^2}{\partial m^2} \ell(x; m) = \sum_{i=1}^{n} \frac{-2\left(1 - (x_i - m)^2\right)}{\left(1 + (x_i - m)^2\right)^2} \tag{4}$$

MLE for Logistic Distribution

For the Logistic distribution, the pdf for a single observation x is:

$$f(x;\mu,\beta) = \frac{e^{-\frac{x-\mu}{\beta}}}{\beta\left(1+e^{-\frac{x-\mu}{\beta}}\right)^2} \tag{5}$$

The log-likelihood function for a sample X is the sum of the logarithms of the individual PDFs:

$$\ell(\mu,\beta) = \sum_{i=1}^{n} \log\left(\frac{e^{-\frac{x_i-\mu}{\beta}}}{\beta\left(1+e^{-\frac{x_i-\mu}{\beta}}\right)^2}\right) \tag{6}$$

The partial derivatives of $\ell(\mu,\beta)$ with respect to μ and β are required to find the MLEs. These derivatives are:

$$\frac{\partial}{\partial\mu}\ell(\mu,\beta) = \sum_{i=1}^{n} \frac{\frac{x_i-\mu}{\beta^2}}{\left(1+e^{-\frac{x_i-\mu}{\beta}}\right)\left(1+e^{\frac{x_i-\mu}{\beta}}\right)} \tag{7}$$

MLE for Logistic Distribution

$$\frac{\partial}{\partial\beta}\ell(\mu,\beta) = \sum_{i=1}^{n}\left(\frac{\frac{x_i-\mu}{\beta^2}e^{-\frac{x_i-\mu}{\beta}}}{\left(1+e^{-\frac{x_i-\mu}{\beta}}\right)^2} - \frac{1}{\beta} + \frac{2e^{-\frac{x_i-\mu}{\beta}}}{\beta\left(1+e^{-\frac{x_i-\mu}{\beta}}\right)^2}\right) \tag{8}$$

$$\frac{\partial^2}{\partial\beta^2}\ell(\mu,\beta) = \sum_{i=1}^{n}\left(\frac{(x_i-\mu)^2\exp\left(\frac{x_i-\mu}{\beta}\right)}{\beta^4\left(1+\exp\left(\frac{x_i-\mu}{\beta}\right)\right)^2} - \frac{1}{\beta^2}\right) \tag{9}$$

$$\frac{\partial^2}{\partial\mu^2}\ell(\mu,\beta) = -\sum_{i=1}^{n}\left(\frac{\exp\left(\frac{x_i-\mu}{\beta}\right)}{\beta^2\left(1+\exp\left(\frac{x_i-\mu}{\beta}\right)\right)^2}\right) \tag{10}$$

Numerical methods such as the Newton-Raphson algorithm are typically used to solve these equations for the MLEs, as they do not have closed-form solutions.

Fitting Logistic Parameters via MLE

Based on source, the log-likelihood function for the Logistic distribution for the sample $\{x_1, \ldots, x_n\}$ is:

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

$$= \frac{n}{\beta}\bar{x} - \frac{n}{\beta}\mu - 2\sum_{i=1}^{n} \ln\left[1 + \exp\left(-\frac{x_i - \mu}{\beta}\right)\right] - n\ln(\beta) \tag{11}$$

To find the maximum value we need to solve the following equations simultaneously:

$$\beta = h(\beta) = \bar{x} - \mu - \frac{2}{n}\sum_{i=1}^{n} \frac{(x_i - \mu)\exp\left(-\frac{x_i-\mu}{\beta}\right)}{1 + \exp\left(-\frac{x_i-\mu}{\beta}\right)} \tag{12}$$

$$0 = g(\mu) = n - 2\sum_{i=1}^{n} \frac{\exp\left(-\frac{x_i-\mu}{\beta}\right)}{1 + \exp\left(-\frac{x_i-\mu}{\beta}\right)} \tag{13}$$

Fitting Logistic Parameters via MLE

We can use a fixed point iteration to find a better version of beta for a given value of mu, namely

$$\beta_{j+1} = h(\beta_j) \tag{14}$$

For any given value of beta, we can use Newton's method to find a better version of mu,

$$\mu_{j+1} = \mu_j - \frac{g(\mu_j)}{g'(\mu_j)} \tag{15}$$

where

$$g'(\mu) = -\frac{2}{\beta} \sum_{i=1}^{n} \frac{\exp\left(-\frac{x_i-\mu}{\beta}\right)}{\left[1+\exp\left(-\frac{x_i-\mu}{\beta}\right)\right]^2} \tag{16}$$

MLE for Logistic and Cauchy distributions

Summary of Methodologies

- **Logistic Distribution**
  - **Fixed-Point Iteration for β:**
    - Initial guess: Median of data.
    - Iterative update based on the current μ and data.
  - **Newton-Raphson for μ:**
    - Initial guess: Median of data.
    - Update using first and second derivatives of log-likelihood.
- **Cauchy Distribution**
  - **Newton-Raphson for m:**
    - Initial guess: Median of data.
    - Update using first and second derivatives of log-likelihood

Derivatives of log-likelihood for Logistic & Cauchy in R

```
# Log-likelihood for Logistic distribution
LL_logistic <- function(mu, beta, data) {
  LL <- -sum((data - mu)/beta)-2*sum(log(1+exp(-(data-mu)/beta)))-length(data)*log(beta)
  return(LL)
}
# Derivative of log-likelihood with respect to mu for Logistic distribution
dLL_logistic_mu <- function(mu, beta, data) {
  dLL_mu <- sum((data-mu)/(beta^2*(1+exp(-(data-mu)/beta))*(1+exp((data-mu)/beta))))
  return(dLL_mu)
}

d2LL_logistic_mu <- function(mu, beta, data) {
  return(-sum(exp((data - mu) / beta) / (beta^2 * (1 + exp((data - mu) / beta))^2)))
}
# Derivative of log-likelihood with respect to beta for Logistic distribution
dLL_logistic_beta <- function(mu, beta, data) {
  # This computes the sum of the derivatives with respect to beta for all data points
  dLL_beta <- sum((data-mu)/(beta^2)*exp(-(data-mu)/beta)/((1 + exp(-(data-mu)/beta))^2) -\
              1/beta + 2 * exp(-(data-mu)/beta)/(beta*(1 + exp(-(data-mu)/beta))^2))
  return(dLL_beta)
}
```

Derivatives of log-likelihood for Logistic & Cauchy in R

```
# Second-order partial derivative with respect to beta for the Logistic
d2LL_logistic_beta <- function(mu, beta, data) {
  return(sum(((data-mu)^2*exp((data-mu)/beta)/(beta^4*(1+exp((data-mu)/beta))^2))-(1/beta^2)))
}

# Log-likelihood for Cauchy distribution
LL_cauchy <- function(params, data) {
  m <- params[1]
  LL <- -length(data) * log(pi) - sum(log(1 + (data - m)^2))
  return(LL)
}

# Derivative of log-likelihood for Cauchy distribution
dLL_cauchy_m <- function(m, data) {
  dLL_m <- 2 * sum((data - m) / (1 + (data - m)^2))
  return(dLL_m)
}

# Second derivative
d2LL_cauchy_m <- function(m, data) {
  d2LL_m <- sum(-2 * (1 - (data - m)^2) / (1 + (data - m)^2)^2)
  return(d2LL_m)
}
```

Implementation of Newton-Raphson and fixed-point in R

```r
fixed_point_logistic_beta <- function(mu, data, tol = 1e-8, max_iter = 100) {
  beta <- median(data)  # Initial guess for beta
  for (i in 1:max_iter) {
    # Calculate beta_new
    exp_terms <- exp(-(data - mu)/beta)
    # Limit exp_terms to avoid Inf or NaN
    exp_terms <- ifelse(abs(exp_terms) > 1e8, 1e8, exp_terms)
    beta_new <- mean(data)-mu-(2/length(data))*sum((data-mu)*exp_terms/(1+exp_terms))
    if (is.na(beta_new) || is.nan(beta_new) || is.infinite(beta_new)) {
      return(NA)
    }
    if (abs(beta - beta_new) < tol) {
      return(beta_new)
    }
    beta <- beta_new
  }
  return(beta)
}
# Newton-Raphson for estimating mu in logistic distribution
newton_raphson_logistic_mu <- function(beta, data, tol = 1e-8, max_iter = 100) {
  mu <- median(data)  # Initial guess for mu
  for (i in 1:max_iter) {
    mu_new <- mu - dLL_logistic_mu(mu, beta, data)/dLL_logistic_beta(mu, beta, data)
    if (abs(mu - mu_new) < tol) {
      return(mu_new)
    }
    mu <- mu_new
  }
  return(mu)
}
```

Implementation of Newton-Raphson and fixed-point in R

```r
# Estimate parameters for logistic distribution
estimate_logistic_params <- function(data, tol = 1e-8, max_iter = 100) {
  mu <- median(data)  # Initial guess for mu
  beta <- fixed_point_logistic_beta(mu, data, tol, max_iter)  # Init FP iter for beta
  for (j in 1:max_iter) {
    # NR iteration for mu
    mu <- newton_raphson_logistic_mu(beta, data, tol, max_iter)
    # FP iteration for beta GIVEN new mu
    beta <- fixed_point_logistic_beta(mu, data, tol, max_iter)
  }
  return(c(mu = mu, beta = beta))
}
# NR for the location parameter 'm' of the Cauchy distribution
newton_raphson_cauchy <- function(data, tol = 1e-8, max_iter = 100) {
  m <- median(data) # A reasonable initial guess

  for (i in 1:max_iter) {
    # Newton-Raphson update
    m_new <- m - dLL_cauchy_m(m, data) / d2LL_cauchy_m(m, data)

    # Check for convergence
    if (abs(m - m_new) < tol) {
      return(m_new)
    }
    m <- m_new
  }
  return(m)
}
```

Implementation of Newton-Raphson and fixed-point in R

Example (Generate data for Cauchy and Logistic using the $\mathcal{U}(0,1)$)

$$\text{If } X \sim \mathcal{U}(0,1) \text{ then } \mu + \beta(\log(X) - \log(1-X)) \sim \text{Logistic}(\mu, \beta).$$

$$\text{If } X \sim \mathcal{U}(0,1) \text{ then } \tan\left(\pi\left(X - 0.5\right)\right) \sim \text{Cauchy}(0,1).$$

```
set.seed(666)
# Generate data from un1form(0,1) -> Cauchy(0,1)
X <- runif(1000)
generate_cauchy <- tan( pi * (X - 0.5) ) # Cauchy(0,1)
# Generate data from un1form(0,1) -> logistic(mu, beta)
mu <- 5
beta <- 2
X <- runif(1000)
generate_log <- mu + beta * (log(X) - log(1 - X))


# NR For the generated data (cauchy(0,1) and logistic(mu=5, beta=2))
logistic_params <- estimate_logistic_params(generate_log, max_iter=20)
cauchy_m <- newton_raphson_cauchy(generate_cauchy, max_iter=20)
# Check methods
logistic_params # Should be close to 5 and 2
cauchy_m # Should be close to 0

>> mu: 5.07959177895596  beta: 1.956210057359
>> m: 0.00754160277996369
```

Implementation of Newton-Raphson and fixed-point in R

Results for NewtonRaphson.csv

```
data <- read.csv("NewtonRaphson.csv", header = TRUE)
# Separate the data into logistic and cauchy observations
cauchy_data <- data[, 2]
logistic_data <- data[, 1]

summary(cauchy_data)
>>    Min.  1st Qu.   Median     Mean   3rd Qu.     Max.
>> -431.492    4.192    5.957   10.056    7.827 3530.204

summary(logistic_data)
>> Min. 1st Qu.  Median    Mean  3rd Qu.    Max.
>> 1969    1975    1976    1976    1977    1984

# NR For the .csv data
logistic_params <- estimate_logistic_params(logistic_data, max_iter=20)
cauchy_m <- newton_raphson_cauchy(cauchy_data, max_iter=20)

logistic_params
cauchy_m

>> mu: 1976.0058106772  beta: 0.988052188422498
>> m: 5.95604108987387
```

Hence, for the logistic data: $\hat{\mu}_{\mathtt{MLE}} \approx 1976$[1], $\hat{\beta}_{\mathtt{MLE}} \approx 1$.

For the Cauchy data: $\hat{m}_{\mathtt{MLE}} \approx 6$, we fixed $b = 1$, for convenience.

[1] Very close to the median.

Second-order derivatives for the log-likelihoods

For the **Logistic Distribution**:

- Second derivative with respect to μ:

$$\frac{\partial^2}{\partial \mu^2}\ell(\mu,\beta) = \sum_{i=1}^{n} \frac{2\exp\left(\frac{x_i-\mu}{\beta}\right)}{\beta^2\left(1+\exp\left(\frac{x_i-\mu}{\beta}\right)\right)^2\left(1+\exp\left(-\frac{x_i-\mu}{\beta}\right)\right)^2}$$

- Second-order mixed partial derivative with respect to μ and β:

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

According to **Clairaut's theorem**, if a function is continuously differentiable, then the mixed partial derivatives are equal. Therefore, in the case of the logistic distribution's log-likelihood function:

$$\frac{\partial^2}{\partial \mu \partial \beta}\ell(\mu,\beta) = \frac{\partial^2}{\partial \beta \partial \mu}\ell(\mu,\beta)$$

Fisher information matrix

Fisher Information Matrix for Logistic Distribution

The Fisher Information Matrix $\mathcal{I}(\theta)$ for parameters μ and β is given by:

$$\mathcal{I}(\mu,\beta) = \begin{pmatrix} -\mathbb{E}\left[\frac{\partial^2}{\partial\mu^2}\ell(\mu,\beta)\right] & -\mathbb{E}\left[\frac{\partial^2}{\partial\mu\partial\beta}\ell(\mu,\beta)\right] \\ -\mathbb{E}\left[\frac{\partial^2}{\partial\beta\partial\mu}\ell(\mu,\beta)\right] & -\mathbb{E}\left[\frac{\partial^2}{\partial\beta^2}\ell(\mu,\beta)\right] \end{pmatrix}$$

Fisher Information Matrix for Cauchy Distribution

For the Cauchy distribution with the location parameter m, the Fisher Information Matrix $\mathcal{I}(m)$ is given by:

$$\mathcal{I}(m) = -\mathbb{E}\left[\frac{\partial^2}{\partial m^2}\ell(m)\right]$$

Confidence intervals using the Fisher information matrix

1. Compute the Fisher Information Matrix for the parameter vector θ, denoted $\mathcal{I}(\theta)$.
2. Invert the Fisher Information Matrix to obtain $\mathcal{I}(\theta)^{-1}$. The inverse provides the variance-covariance matrix of the parameter estimates.
3. Calculate the standard errors (SE) of the parameter estimates. The standard error for each parameter is the square root of the corresponding diagonal element of $\mathcal{I}(\theta)^{-1}$.
4. Construct the confidence intervals for each parameter. For a parameter estimate $\hat{\theta}$ and its standard error $SE(\hat{\theta})$, a 95% confidence interval is given by $\hat{\theta} \pm$ [2] $1.96 \times SE(\hat{\theta})$.

[2] Equals to `qnorm(0.975)`. `qnorm()` essentially allows us to find the quantile (or the inverse of the cdf) for a given probability in a normal distribution.

Confidence intervals using the Fisher information in R

```
# Second-order mixed partial derivative with respect to mu and beta for Logistic distribution
d2LL_logistic_mu_beta <- function(mu, beta, data) {
  sum_terms <- -sum((data-mu)*exp(-(data-mu)/beta)/(beta^3*(1+exp(-(data-mu)/beta))^2)-2*\
              (data-mu)*exp(-(data-mu)/beta)/(beta^2*(1+exp(-(data-mu)/beta))^3))
  return(sum_terms)
}
d2LL_logistic_beta_mu <- function(mu, beta, data) {
  d2LL_logistic_mu_beta(mu, beta, data)  # Same as d2LL_log_mu_beta
}

# Fisher information matrix for Logistic Distribution
compute_fisher_matrix_logistic <- function(mu, beta, data) {
  matrix(c(
    -d2LL_logistic_mu(mu, beta, data), -d2LL_logistic_mu_beta(mu, beta, data),
    -d2LL_logistic_beta_mu(mu, beta, data), -d2LL_logistic_beta(mu, beta, data)
  ), nrow = 2)
}

# Calculate confidence intervals for Logistic parameters
compute_ci_logistic <- function(mu, beta, data) {
  fisher_matrix <- compute_fisher_matrix_logistic(mu, beta, data)
  inv_fisher <- solve(fisher_matrix)
  se <- sqrt(diag(inv_fisher))

  ci_mu <- mu + c(-1, 1) * qnorm(0.975) * se[1]
  ci_beta <- beta + c(-1, 1) * qnorm(0.975) * se[2]

  return(list(ci_mu = ci_mu, ci_beta = ci_beta))
}
```

Confidence intervals using the Fisher information in R

```
estimated_mu <- as.numeric(logistic_params)[1]
estimated_beta <- as.numeric(logistic_params)[2]
cat("Estimated mu, beta:", logistic_params, '\n')
logistic_ci <- compute_ci_logistic(estimated_mu, estimated_beta, logistic_data)
print(logistic_ci)
>> Estimated mu, beta: 1976.006 0.9880522
>> $ci_mu
>> [1] 1975.853 1976.159
>>
>> $ci_beta
>> [1] 0.9174593 1.0586450
```

- For the logistic distribution:
  - 95% CI for $\hat{\mu}_{MLE} \approx 1976.006$: $[1975.853, 1976.159]$
  - 95% CI for $\hat{\beta}_{MLE} \approx 0.9880522$: $[0.9174593, 1.0586450]$

Confidence intervals using the Fisher information in R

```
# Function to calculate confidence intervals for Cauchy parameter
compute_ci_cauchy <- function(m, data) {
  fisher_info <- -d2LL_cauchy_m(m, data)
  inv_fisher <- 1 / fisher_info
  se <- sqrt(inv_fisher)
  ci_m <- m + c(-1, 1) * qnorm(0.975) * se

  return(ci_m)
}

# Example usage for Cauchy Distribution
estimated_m <- cauchy_m
cauchy_ci <- compute_ci_cauchy(estimated_m, cauchy_data)
cat("Estimated m:", cauchy_m, '\n')
cat("CI for m:", cauchy_ci)
>> Estimated m: 5.956041
>> CI for m: 5.830927 6.081156
```

- For the Cauchy distribution:
  - 95% CI for $\hat{m}_{MLE} \approx 5.956041$: $[5.830927, 6.081156]$
  - b is fixed.

Bootstrap confidence intervals for MLE

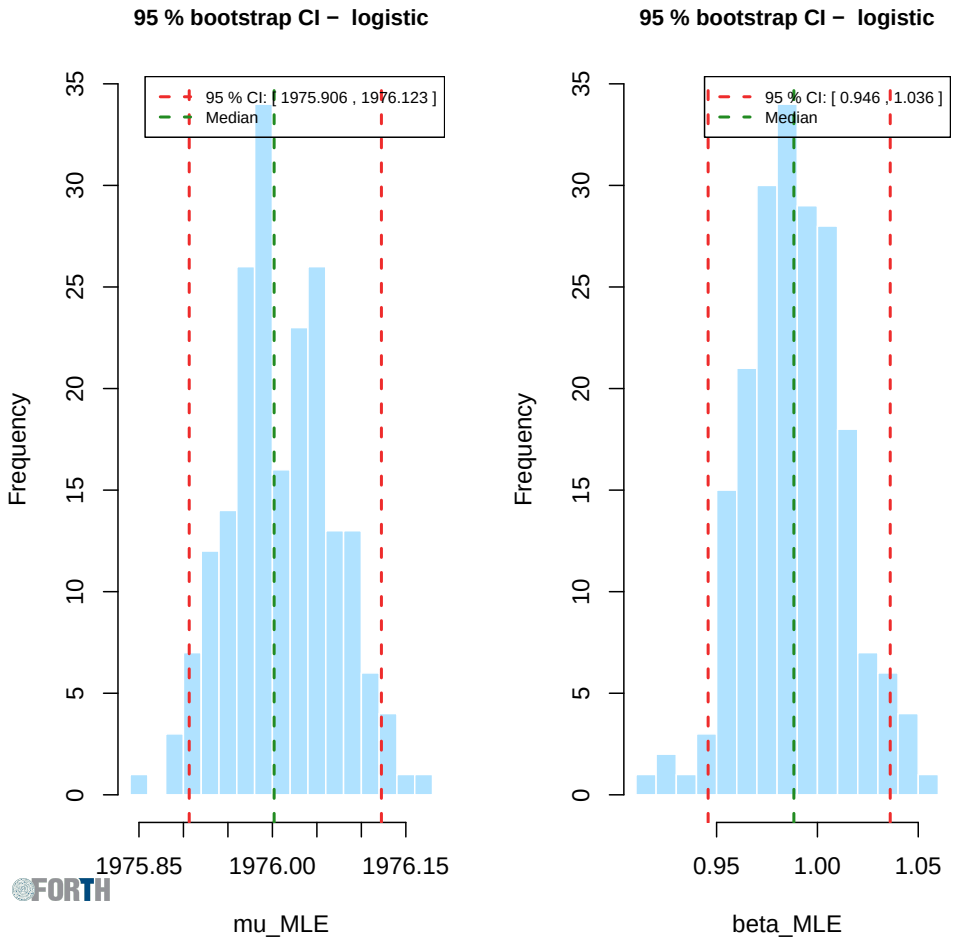
```
get.bootstrapCI <- function(B=100, data, name="distribution", alpha = 0.05,
                            plot.histogram = FALSE, bins = 16) {

  if (name == "logistic") {
      boot_stats <- matrix(NA, nrow = B, ncol = 2)
      colnames(boot_stats) <- c("mu_MLE", "beta_MLE")
  } else if (name == "cauchy") {
      boot_stats <- matrix(NA, nrow = B, ncol = 1)
      colnames(boot_stats) <- c("m_MLE")
  }
  pb <- txtProgressBar(min = 0, max = B, style = 3) # Progress bar setup
  # Bootstrap loop
  for(b in 1:B) {
    setTxtProgressBar(pb, b)
    # Set a random state for each bootstrap sample
    set.seed(b)
    boot_data <- data[sample(nrow(data), replace = TRUE), ]
    # Get the MLE.
    if (name == "logistic") {
        MLE <- estimate_logistic_params(boot_data, max_iter=20)
        MLE <- t(as.data.frame(MLE))
        boot_stats[b, ] <- c(MLE[1], MLE[2])
    } else if (name == "cauchy") {
        MLE <- newton_raphson_cauchy(boot_data, max_iter=20)
        MLE <- t(as.data.frame(MLE, 'm'))
        boot_stats[b, ] <- MLE
    }
  }
  close(pb)
```

Bootstrap confidence intervals for MLE

```
  if (!plot.histogram) {
    # Calc. the (1-0.05)*100% CI (set alpha=0.05 for 95% CI)
    ci_lower <- apply(boot_stats, 2, function(x) quantile(x, probs = alpha/2))
    ci_upper <- apply(boot_stats, 2, function(x) quantile(x, probs = 1 - alpha/2))
    CI <- data.frame(
      LowerCI = ci_lower, #lower
      UpperCI = ci_upper  #upper
    )
    return(CI)
  } else { # Get the histograms
    par(mfrow=c(1, 2))
    for (i in 1:ncol(boot_stats)) {
      metric <- colnames(boot_stats)[i]
      metric_values <- boot_stats[, i]
      ci <- quantile(metric_values, probs=c(alpha/2, 1-alpha/2))
      x_lim <- range(c(metric_values, ci[1], ci[2])) # CI in the range
      x_lim <- c(x_lim[1] - diff(x_lim) * 0.1, x_lim[2] + diff(x_lim) * 0.1)
      hist(metric_values, main=paste((1-alpha)*100, "% bootstrap CI - ", name),
           xlab=metric, col='lightskyblue1', border='white', breaks=bins, cex.main = 0.956)
       # CI and Median lines
      abline(v=ci[1], col="firebrick2", lwd=2, lty=2)
      abline(v=median(metric_values), col="forestgreen", lwd=2, lty=2)
      abline(v=ci[2], col="firebrick2", lwd=2, lty=2)
      # Add a legend with the CI and Median
      legend("topright", legend=c(paste((1-alpha)*100,
             "% CI: [", round(ci[1], 3), ',', round(ci[2], 3), "]"), "Median"),
             col=c("firebrick2", "forestgreen"), lwd=2, lty=2, cex=0.7)
    }
    par(mfrow=c(1, 1))
  }
}
```

Comparing Fisher vs bootstrap 95% CI for MLE

```
get.bootstrapCI(B=200,
     data=as.data.frame(logistic_data),
     name='logistic',
     plot.histogram=TRUE)
```



Fisher vs Bootstrap CI:

- Fisher:

$$\hat{\mu}_{\texttt{MLE}} : [1975.853, 1976.159]$$

$$\hat{\beta}_{\texttt{MLE}} : [0.9174593, 1.0586450]$$

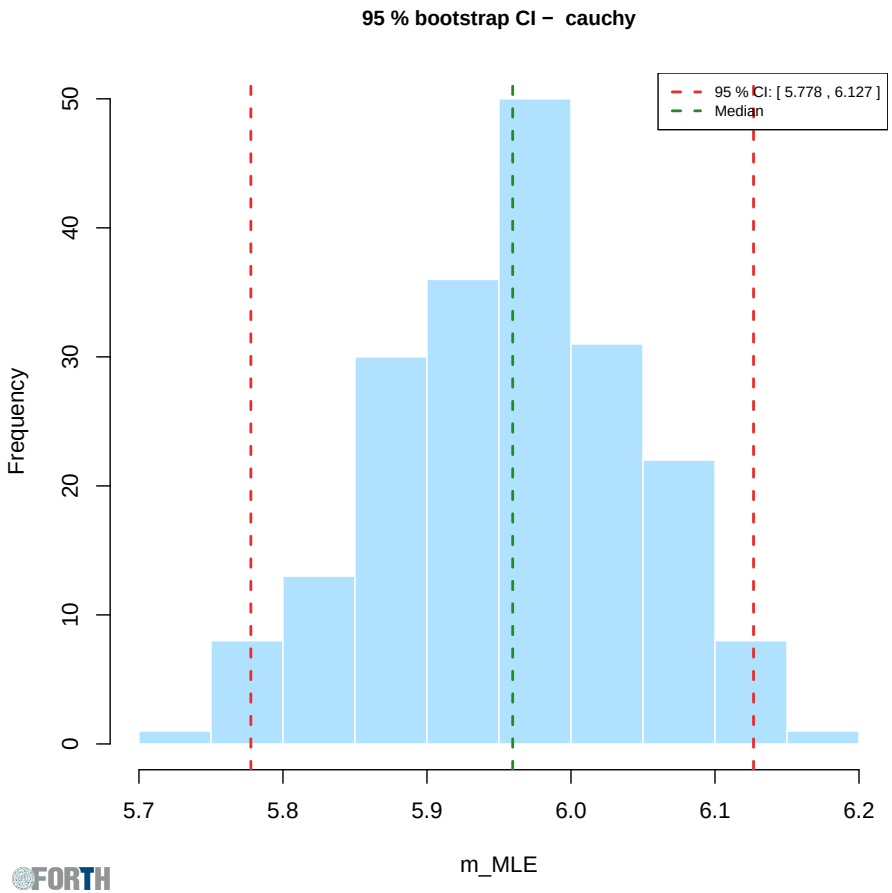
- Bootstrap:

$$\hat{\mu}_{\texttt{MLE}} : [1975.906, 1976.123]$$

$$\hat{\beta}_{\texttt{MLE}} : [0.946, 1.036]$$

Comparing Fisher vs bootstrap 95% CI for MLE

```
get.bootstrapCI(B=200,
     data=as.data.frame(cauchy_data),
     name='cauchy',
     plot.histogram=TRUE)
```



Fisher vs Bootstrap CI:

- Fisher:

$$\hat{m}_{\mathtt{MLE}} : [5.830927, 6.081156]$$

- Bootstrap:

$$\hat{m}_{\mathtt{MLE}} : [5.778, 6.127]$$

Confidence intervals based on Monte Carlo simulations

A Monte Carlo experiment can be utilized to establish a confidence interval. By generating data from both the Cauchy and logistic distributions R times, using the approximated MLE parameters previously computed, we can calculate the corresponding MLE for each dataset. Subsequently, we can report the confidence interval (based on quantiles). To accomplish this in R, we can slightly modify the previously used bootstrap function.

Confidence intervals based on Monte Carlo simulations

```
generate_log.data <- function (n, mu, beta, rstate=42) {
    set.seed(rstate)
    X <- runif(n)
    return (mu + beta * (log(X) - log(1 - X) ))
}
get.MC_CI <- function (R=100, name="distribution", alpha=0.05, plot.histogram=FALSE,
                       bins = 16, m=5.956, mu=1976.006, beta=0.99) {
    if (name == "logistic") {
        boot_stats <- matrix(NA, nrow = R, ncol = 2)
        colnames(boot_stats) <- c("mu_MLE", "beta_MLE")
    } else if (name == "cauchy") {
        boot_stats <- matrix(NA, nrow = R, ncol = 1)
        colnames(boot_stats) <- c("m_MLE")
    }
    pb <- txtProgressBar(min = 0, max = R, style = 3) # Progress bar

    for(r in 1:R) {
        setTxtProgressBar(pb, r)
        if (name == "logistic") {
            MC_data <- generate_log.data(n=n, mu=mu, beta=beta, rstate=r)
            MLE <- as.numeric(estimate_logistic_params(MC_data, max_iter=20)) # Gen. log. data
            boot_stats[r, ] <- MLE[1:2]
        } else if (name == "cauchy") {
            MC_data <- rcauchy(n, location=m, scale=1) # Generate Cauchy data
            MLE <- newton_raphson_cauchy(MC_data, max_iter=20)
            boot_stats[r, ] <- MLE
        }
    }
    close(pb)
```

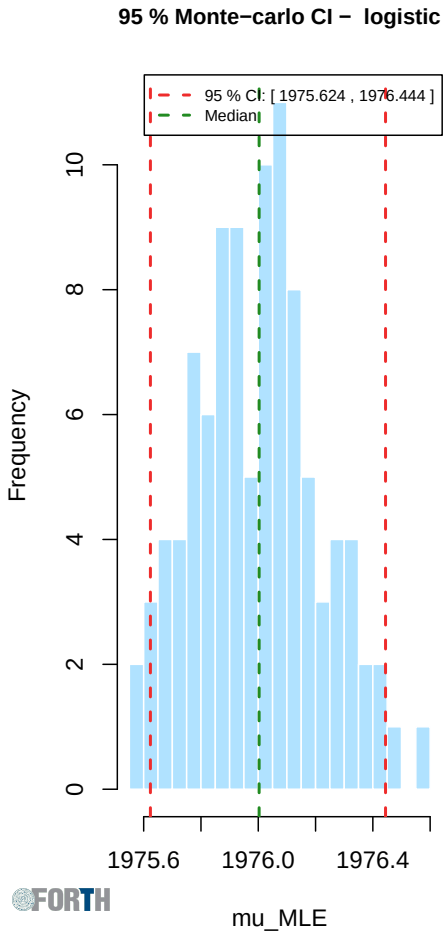
Confidence intervals based on Monte Carlo simulations

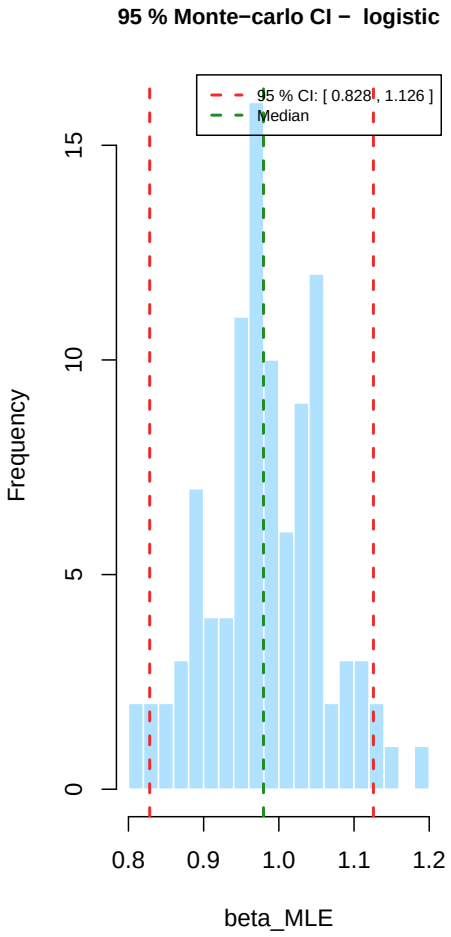
```
  if (!plot.histogram) {
    ci_lower <- apply(boot_stats, 2, function(x) quantile(x, probs = alpha/2))
    ci_upper <- apply(boot_stats, 2, function(x) quantile(x, probs = 1 - alpha/2))
    CI <- data.frame(
      LowerCI = ci_lower,  #lower
      UpperCI = ci_upper)  #upper
    return(CI)
  } else {
    par(mfrow=c(1, 2))
    for (i in 1:ncol(boot_stats)) {
      metric <- colnames(boot_stats)[i]
      metric_values <- boot_stats[, i]
      ci <- quantile(metric_values, probs=c(alpha/2, 1-alpha/2))
      x_lim <- range(c(metric_values, ci[1], ci[2])) # CI in the range
      x_lim <- c(x_lim[1] - diff(x_lim) * 0.1, x_lim[2] + diff(x_lim) * 0.1)
      hist(metric_values, main=paste((1-alpha)*100, "% Monte-carlo CI - ", name),
           xlab=metric, col='lightskyblue1', border='white',
           breaks=bins, cex.main = 0.956)
       # CI and Median lines
      abline(v=ci[1], col="firebrick2", lwd=2, lty=2)
      abline(v=median(metric_values), col="forestgreen", lwd=2, lty=2)
      abline(v=ci[2], col="firebrick2", lwd=2, lty=2)
      legend("topright", legend=c(paste((1-alpha)*100,
                                  "% CI: [", round(ci[1], 3),
                                  ',', round(ci[2], 3), "]"), "Median"),
             col=c("firebrick2", "forestgreen"), lwd=2, lty=2, cex=0.7)
    }
    par(mfrow=c(1, 1))
  }
}
```

Comparing Fisher; bootstrap and Monte Carlo 95% CI

Logistic MLE

```
get.MC_CI(R=100, name='logistic',
          plot.histogram=TRUE)
```





Fisher vs Bootstrap vs MC CI:

- Fisher:

$$\hat{\mu}_{\mathtt{MLE}} : [1975.853, 1976.159]$$

$$\hat{\beta}_{\mathtt{MLE}} : [0.9174593, 1.0586450]$$

- Bootstrap:

$$\hat{\mu}_{\mathtt{MLE}} : [1975.906, 1976.123]$$

$$\hat{\beta}_{\mathtt{MLE}} : [0.946, 1.036]$$

- Monte Carlo:

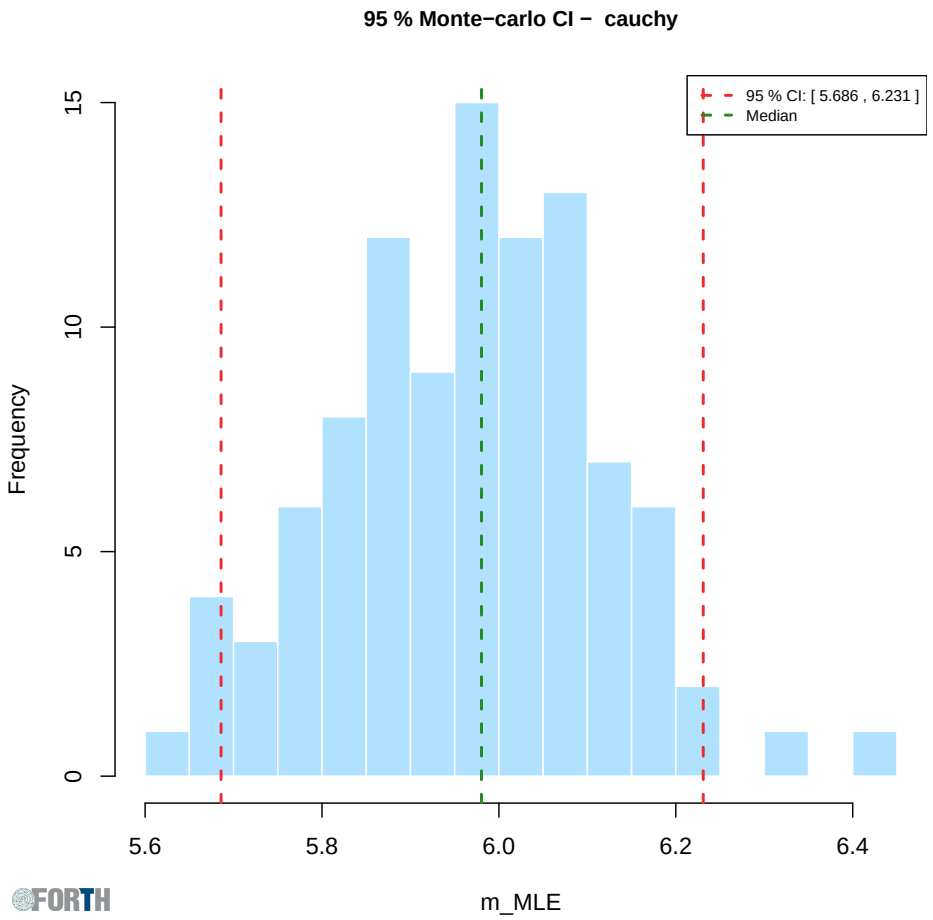
$$\hat{\mu}_{\mathtt{MLE}} : [1975.624, 1976.444]$$

$$\hat{\beta}_{\mathtt{MLE}} : [0.828, 1.126]$$

Comparing Fisher; bootstrap and Monte Carlo 95% CI

Cauchy MLE

```
get.MC_CI(R=100, name='cauchy',
          plot.histogram=TRUE)
```



Fisher vs Bootstrap vs MC CI:

- Fisher:

$$\hat{m}_{\mathtt{MLE}} : [5.830927, 6.081156]$$

- Bootstrap:

$$\hat{m}_{\mathtt{MLE}} : [5.778, 6.127]$$

- Monte Carlo:

$$\hat{m}_{\mathtt{MLE}} : [5.686, 6.231]$$

Assignment 4 part B

Consider the density $f(x;\theta) = \frac{1-\cos\left(\frac{x-\theta}{2\pi}\right)}{2\pi}$ on $0 \leq x \leq 2\pi$, 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 solutions 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.

Deriving and draw the Log-Likelihood function

$$f(x;\theta) = \frac{1-\cos(x-\theta)}{2\pi},$$

for $0 \leq x \leq 2\pi$ and $-\pi \leq \theta \leq \pi$. The log-likelihood function $\ell(\theta)$ for a set of i.i.d. data points $x_1, x_2, \ldots, x_n$ is the logarithm of the product of their individual probability densities:

$$\ell(\theta) = \log\left(\prod_{i=1}^{n} f(x_i;\theta)\right) = \sum_{i=1}^{n} \log\left(f(x_i;\theta)\right)$$

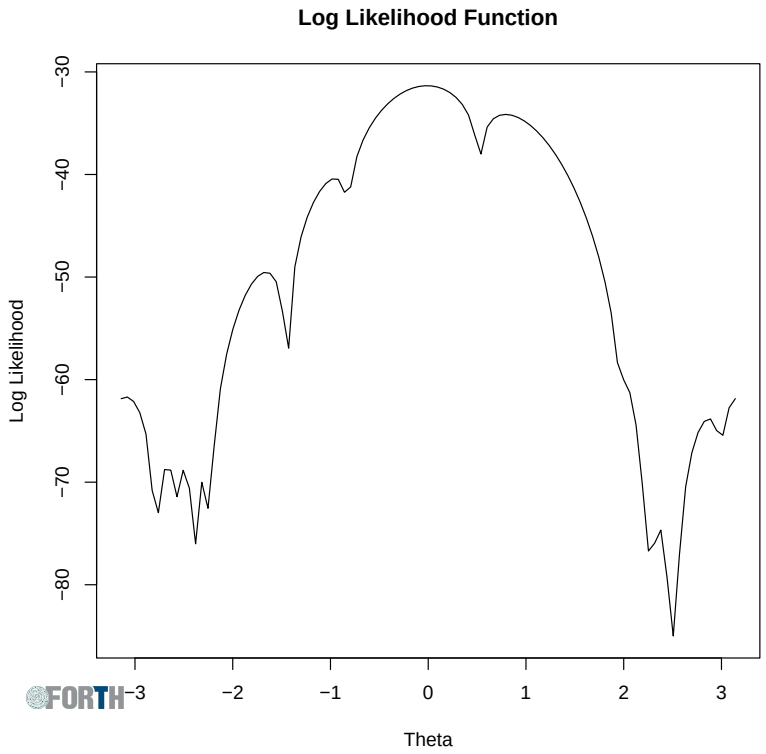
Plugging in the given density function, we get:

$$\ell(\theta) = \sum_{i=1}^{n} \log\left(\frac{1-\cos(x_i-\theta)}{2\pi}\right)$$

We can graph this function using R by evaluating it at a range of θ values between $-\pi$ and π. Let's write the R code to do this:

Log-Likelihood graph between $-\pi$ and π

```
log_likelihood <- function(theta, data) {
  sum(log((1 - cos(data - theta)) / (2 * pi)))
}
data_iid <- 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)
# Range of theta values
theta_values <- seq(-pi, pi, length.out=100)
# Evaluate the log likelihood for each theta
LL_values <- sapply(theta_values, log_likelihood, data=data_iid)
plot(theta_values, LL_values, type='l', xlab="Theta", ylab="Log Likelihood",
     main="Log Likelihood Function")
```



Method of moments

The method of moments is an estimation technique that involves equating the sample moments with the population moments (theoretical moments) and solving for the parameters of the distribution.
Given a probability density function $f(x;\theta)$, the k-th theoretical moment about the origin is given by

$$\mu'_k(\theta) = \mathbb{E}_\theta(X^k) = \int_{-\infty}^{\infty} x^k f(x;\theta)dx.$$

For the method of moments estimator, we start with the first moment (the mean) when estimating a single parameter. For the given density function:
First Moment (Mean):

$$\mathbb{E}[X] = \frac{1}{2\pi}\int_0^{2\pi} x\big(1-\cos(x-\theta)\big)dx = \sin(\theta)+\pi$$

Estimate θ with method of moments

The law of large numbers states that

$$\bar{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i \to \mu \text{ as } n \to \infty.$$

Thus, if the number of observations n is large, the distributional mean, $\mu = \mathbb{E}_\theta$, should be well approximated by the sample mean, i.e.,

$$\bar{X} \approx \mathbb{E}_\theta.$$

This can be turned into an estimator $\hat{\theta}$ by setting

$$\bar{X} = \mathbb{E}_{\hat{\theta}}.$$

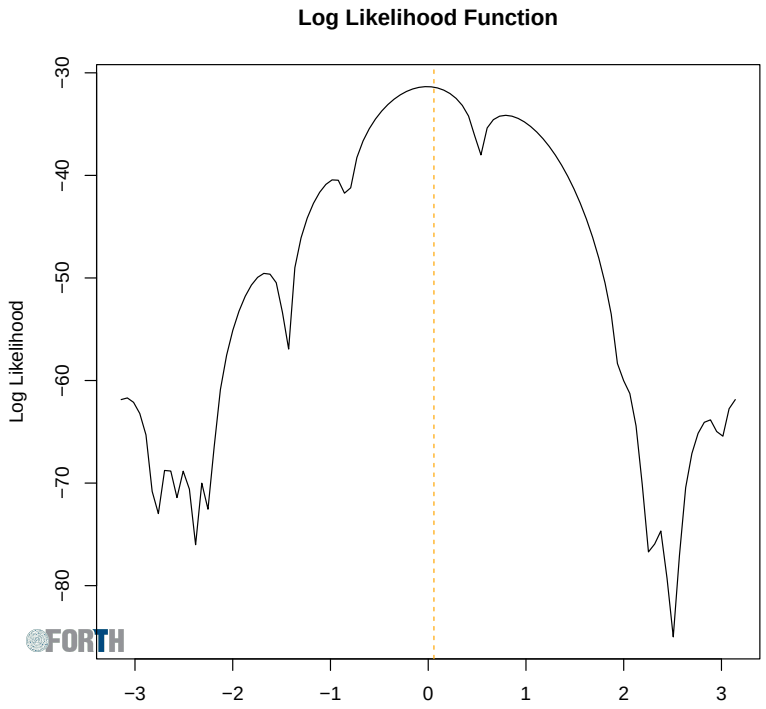
and solving for $\hat{\theta}$.

Estimate θ with method of moments

To find the method of moments (MoM) estimator for θ, we set the theoretical mean equal to the sample mean $\overline{X}$, and solve for θ:

$$\sin(\theta) + \pi = \overline{X} \iff \theta = \sin^{-1}(\overline{X} - \pi) = 0.0584406061404241$$

```
plot(theta_values, LL_values, type='l',
     xlab="Theta", ylab="Log Likelihood",
     main="Log Likelihood Function")
abline(v = theta_MoM, col = 'orange', lty = 2) # Very close to the MLE
```



MLE with Newton-Raphson

The first derivative of the log-likelihood function (score function) with respect to θ is:

$$\frac{d\ell(\theta)}{d\theta} = \sum_{i=1}^{n} \frac{\sin(x_i - \theta)}{1 - \cos(x_i - \theta)}$$

And the second derivative is:

$$\frac{d^2\ell(\theta)}{d\theta^2} = -\sum_{i=1}^{n} \frac{1}{1 - \cos(x_i - \theta)} - \frac{\sin^2(x_i - \theta)}{(1 - \cos(x_i - \theta))^2}$$

These derivatives are used to perform the Newton-Raphson update:

$$\theta_{n+1} = \theta_n - \frac{\frac{d\ell(\theta_n)}{d\theta_n}}{\frac{d^2\ell(\theta_n)}{d\theta_n^2}}$$

We iteratively update θ until convergence, meaning subsequent estimates do not change significantly (within a specified tolerance). In R, we would code it like this:

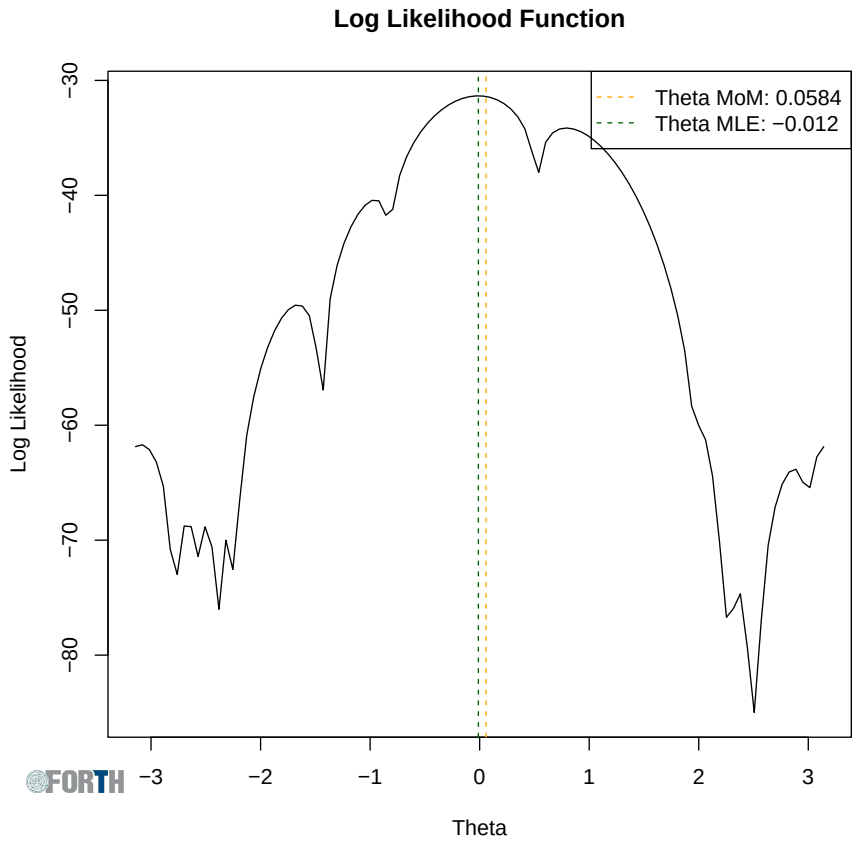
MLE with Newton-Raphson

```
dLL <- function(theta, x) {
  sin_diff <- sin(x - theta)
  cos_diff <- cos(x - theta)
  sum(sin_diff / (1 - cos_diff))
}

d2LL <- function(theta, x) {
  cos_diff <- cos(x - theta)
  sin_diff <- sin(x - theta)
  -sum(-1 / (1 - cos_diff) - (sin_diff^2) / (1 - cos_diff)^2)
}

newton_raphson <- function(data, theta_init, tol=1e-6, max_iter=50) {
  theta <- theta_init
  for (i in 1:max_iter) {
    theta_new <- theta - dLL(theta, data) / d2LL(theta, data)
    # Check for convergence
    if (abs(theta_new - theta) < tol) {
      break
    }
    theta <- theta_new
  }
  return(theta)
}

theta_MLE <- newton_raphson(data=data_iid, theta_init=theta_MoM)
```

MLE with Newton-Raphson

initial guess: Method of moments

```
plot(theta_values, LL_values, type='l', xlab="Theta", ylab="Log Likelihood",
     main="Log Likelihood Function")
abline(v = theta_MoM, col = 'orange', lty = 2)
abline(v = theta_MLE, col = 'darkgreen', lty = 2)
legend('topright', legend = c(paste('Theta MoM:', round(theta_MoM,4)),
                              paste('Theta MLE:', round(theta_MLE,4))),
       col = c('orange', 'darkgreen'), lty = c(2,2), pch = c(NA, NA))
```



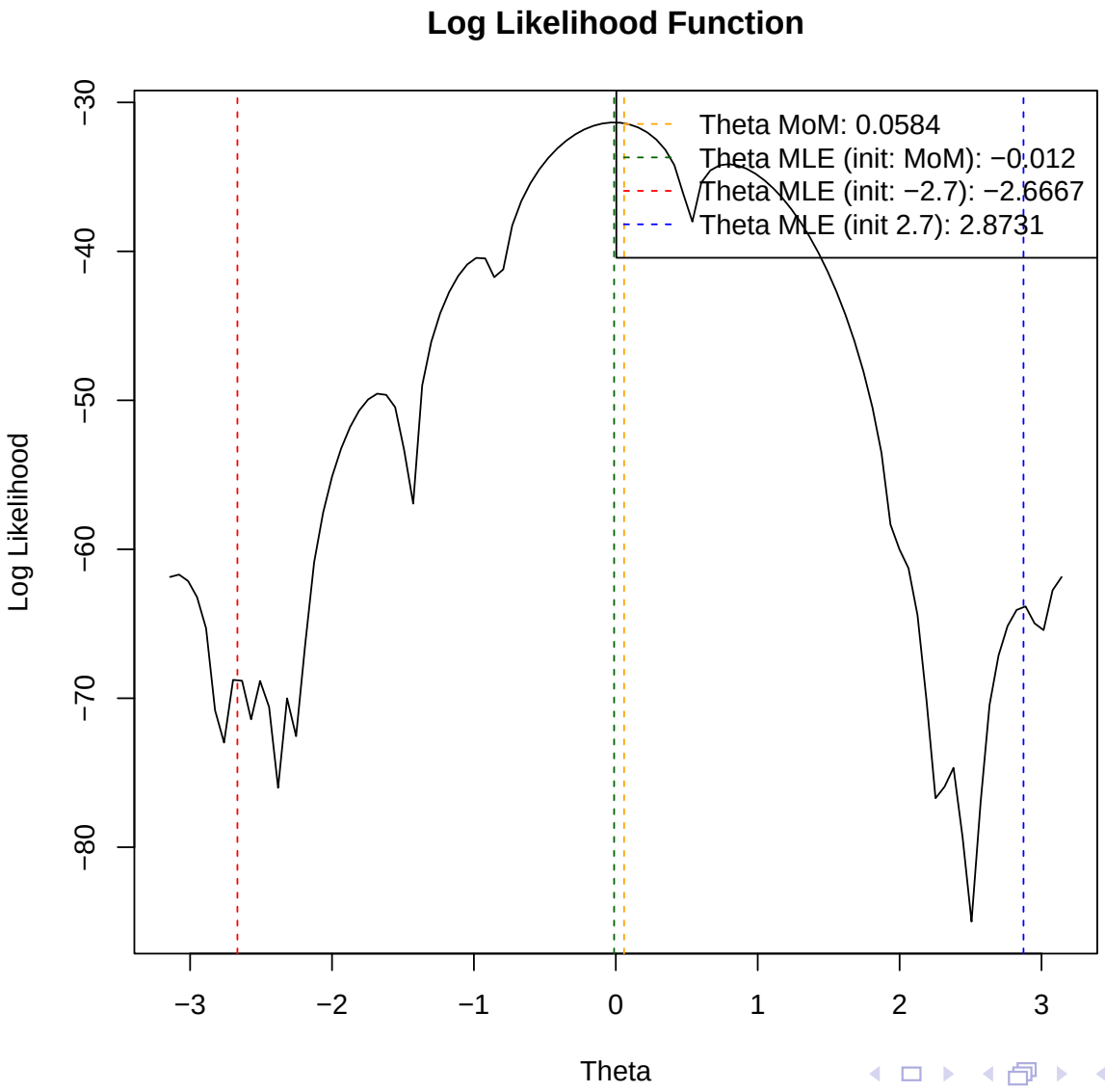
MLE with Newton-Raphson

initial guess: Method of moments; -2.7; 2.7

```
# Start from -2.7 and 2.7
theta_MLE2 <- newton_raphson(data=data_iid, theta_init=-2.7)
theta_MLE3 <- newton_raphson(data=data_iid, theta_init=2.7)

plot(theta_values, LL_values, type='l',
     xlab="Theta", ylab="Log Likelihood",
     main="Log Likelihood Function")
abline(v = theta_MoM, col = 'orange', lty = 2)
abline(v = theta_MLE, col = 'darkgreen', lty = 2)
abline(v = theta_MLE2, col = 'red', lty = 2)
abline(v = theta_MLE3, col = 'blue', lty = 2)
legend('topright', legend = c(paste('Theta MoM:', round(theta_MoM, 4)),
                              paste('Theta MLE (init: MoM):', round(theta_MLE, 4)),
                              paste('Theta MLE (init: -2.7):', round(theta_MLE2, 4)),
                              paste('Theta MLE (init 2.7):', round(theta_MLE3, 4))),
       col = c('orange', 'darkgreen', 'red', 'blue'),
       lty = c(2,2), pch = c(NA, NA))
```

MLE with Newton-Raphson

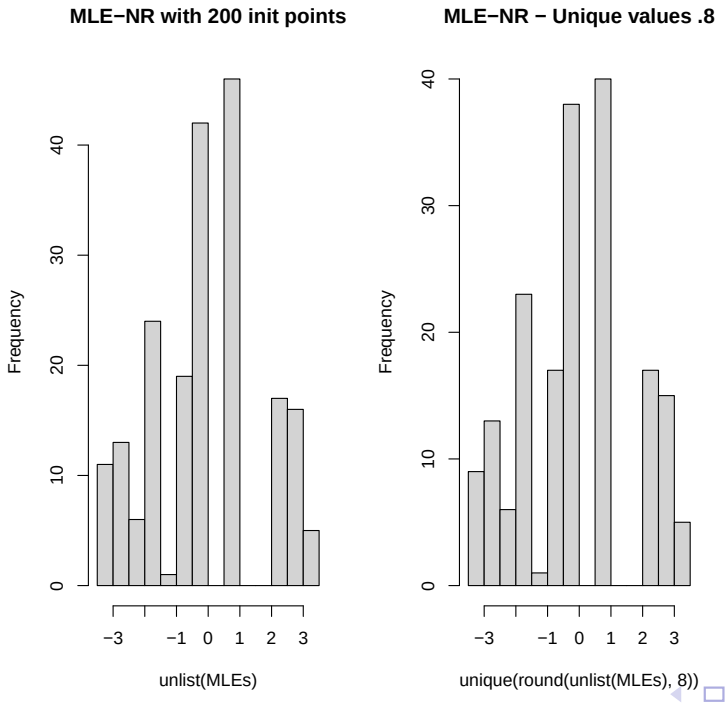
initial guess: Method of moments; -2.7; 2.7



MLE with Newton-Raphson

initial guess: Method of moments; 200 points between $-\pi$, π

```
init_guess <- seq(from = -pi, to = pi, length.out = 200)
MLEs <- list()
i <- 1
for (MLE_guess in init_guess) {
    MLEs[[i]] <- newton_raphson(data=data_iid, theta_init=MLE_guess)
    i = i + 1
}
par(mfrow=c(1,2))
hist(unlist(MLEs), main="MLE-NR with 200 init points")
hist(unique(round(unlist(MLEs), 8)), main="MLE-NR - Unique values .8")
```



MLE with Newton-Raphson

initial guess: Method of moments; 200 points between $-\pi$, π

```
unique(round(unlist(MLEs), 3))
>> -3.093, -2.786, -2.667, -2.508, -2.388, -2.297, -2.232, -1.658, -1.447, -0.953, -0.012,
     0.791, 2.004, 2.236, 2.361, 2.475, 2.514, 2.873, 3.19
```

MLE with Newton-Raphson

initial guess: Method of moments; 200 points between $-\pi$, π

```
unique(round(unlist(MLEs), 3))
>> -3.093, -2.786, -2.667, -2.508, -2.388, -2.297, -2.232, -1.658, -1.447, -0.953, -0.012,
     0.791, 2.004, 2.236, 2.361, 2.475, 2.514, 2.873, 3.19
```

Starting with 0.51 and 0.53 yields significantly different results, as can be observed below.

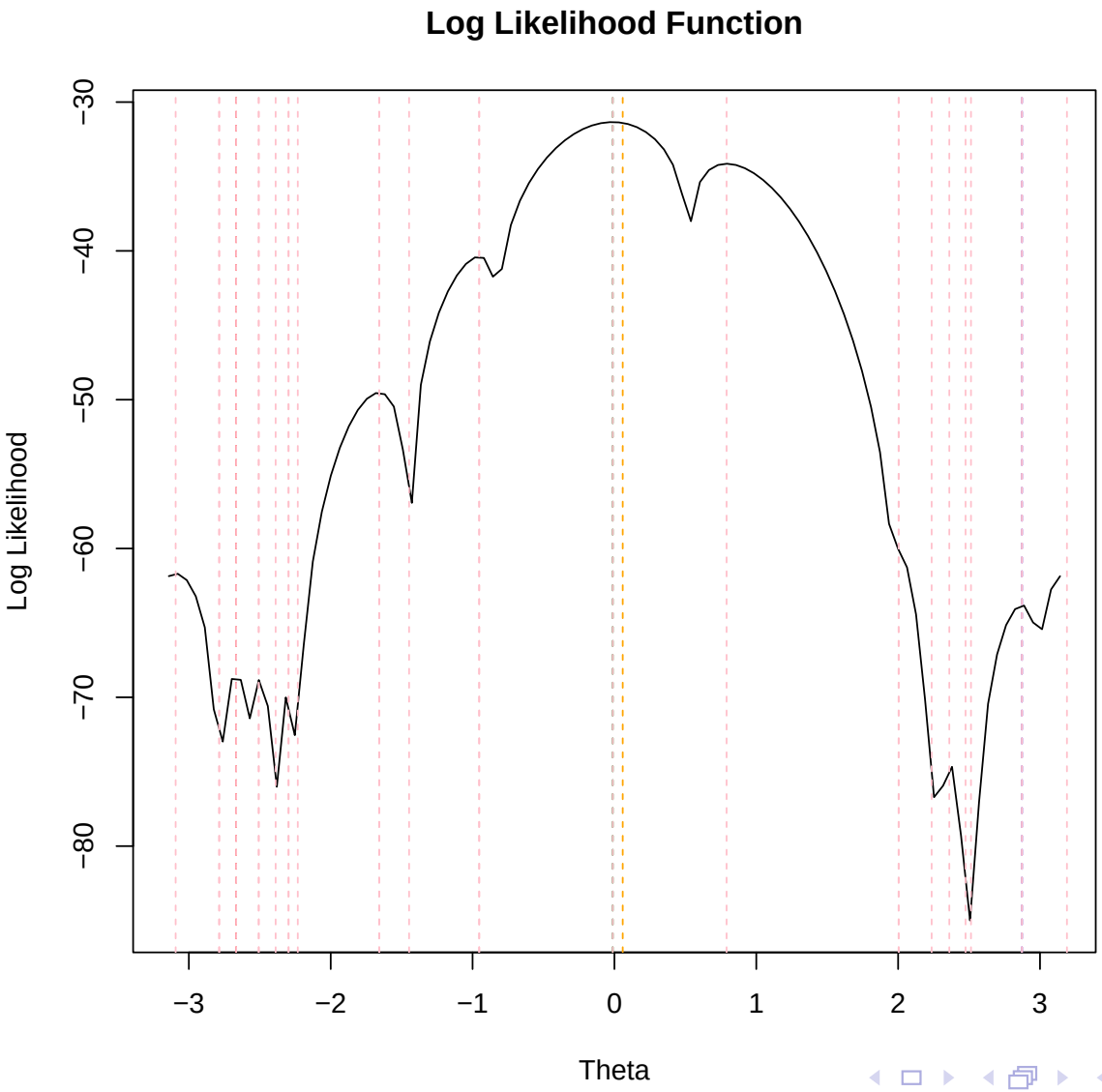
```
newton_raphson(data=data_iid, theta_init=0.51)
newton_raphson(data=data_iid, theta_init=0.53)
>> -0.0119705854515564
>> 0.790599084511473
```

We can examine the various Newton-Raphson solutions for this specific partition of 200 points within the range $-\pi$ to π:

```
plot(theta_values, LL_values, type='l', xlab="Theta",
     ylab="Log Likelihood", main="Log Likelihood Function")
abline(v = theta_MoM, col = 'orange', lty = 2)
abline(v = theta_MLE, col = 'darkgreen', lty = 2)
abline(v = unique(round(unlist(MLEs), 8)), col = 'pink', lty = 2)
```

MLE with Newton-Raphson

initial guess: Method of moments; 200 points between $-\pi$, π



Assignment 4 part C

- 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.
- Are there differences between Newton-Raphson and Fisher-Scoring in this case?
- Compare your results versus the ones obtained from the glm function for the following data:

```
counts <- c(18,17,15,20,10,20,25,13,12)
outcome <- gl(3 ,1 ,9)
treatment <- gl(3 ,3)
print(d.AD <- data.frame(treatment, outcome, counts))
glm.D93 <- glm(counts ~ outcome + treatment, family=poisson())
```

Poisson regression

In **Poisson regression**, we model count data. Given a response variable Y representing count data, and a set of explanatory variables $\mathbf{X}$, the Poisson regression model is:

$$\log(\mu_i) = \mathbf{x}_i^T \boldsymbol{\beta}$$

where:

- μ_i is the expected value of Y_i.
- $\mathbf{x}_i$ is the vector of explanatory variables for the i-th observation.
- $\boldsymbol{\beta}$ is the vector of parameters.

The Poisson distribution of Y_i given μ_i is:

$$P(Y_i = y_i) = \frac{\exp(-\mu_i)\mu_i^{y_i}}{y_i!}$$

The likelihood function for the Poisson regression model, given the data $(y_i, \mathbf{x}_i)$ for $i = 1, \ldots, n$, is the product of the individual probabilities:

$$L(\boldsymbol{\beta}) = \prod_{i=1}^{n} \frac{\exp(-\mu_i)\mu_i^{y_i}}{y_i!}$$

Poisson regression and Fisher scoring

Taking the logarithm, the log-likelihood function $\ell(\boldsymbol{\beta})$ becomes:

$$\ell(\boldsymbol{\beta}) = \sum_{i=1}^{n} \left(-\mu_i + y_i \log(\mu_i) - \log(y_i!)\right) \tag{17}$$

The score function is the gradient (vector of first derivatives) of the log-likelihood with respect to $\boldsymbol{\beta}$. For Poisson regression, it is:

$$U(\boldsymbol{\beta}) = \frac{\partial \ell(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = \mathbf{X}^T(\mathbf{y} - \boldsymbol{\mu}) \tag{18}$$

Let's prove (18). Recall that in (17), $\mu_i = \exp(\mathbf{x}_i^T \boldsymbol{\beta})$. Thus,

$$\frac{\partial \ell(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = \sum_{i=1}^{n} \left(-\frac{\partial \mu_i}{\partial \boldsymbol{\beta}} + y_i \frac{1}{\mu_i} \frac{\partial \mu_i}{\partial \boldsymbol{\beta}} \right)$$

Since $\mu_i = \exp(\mathbf{x}_i^T \boldsymbol{\beta})$, its derivative with respect to $\boldsymbol{\beta}$ is $\mu_i \mathbf{x}_i$. So, the equation becomes:

$$\frac{\partial \ell(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = \sum_{i=1}^{n} \mathbf{x}_i (y_i - \mu_i) = \mathbf{X}^T(\mathbf{y} - \boldsymbol{\mu})$$

Poisson regression and Fisher scoring

Hessian of Log-Likelihood: The second derivative of the log-likelihood with respect to $\boldsymbol{\beta}$ is:

$$\frac{\partial^2 \ell(\boldsymbol{\beta})}{\partial \boldsymbol{\beta} \partial \boldsymbol{\beta}^T} = -\sum_{i=1}^{n} \mu_i \mathbf{x}_i \mathbf{x}_i^T$$

Expectation of the Hessian: The Fisher Information matrix is the expected value of the negative second derivative (Hessian) of the log-likelihood; the expectation of μ_i is just μ_i itself. Therefore, the Fisher Information matrix is:

$$I(\boldsymbol{\beta}) = \mathbf{X}^T \text{diag}(\boldsymbol{\mu}) \mathbf{X}$$

Here, $\text{diag}(\boldsymbol{\mu})$ is a diagonal matrix with the elements of $\boldsymbol{\mu}$ on its diagonal, and $\mathbf{X}$ is the design matrix.

Fisher scoring and confidence interval

The Fisher Scoring algorithm is an iterative procedure to find the maximum likelihood estimate of $\boldsymbol{\beta}$. At each iteration t, the update rule is:

$$\boldsymbol{\beta}^{(t+1)} = \boldsymbol{\beta}^{(t)} + I(\boldsymbol{\beta}^{(t)})^{-1} U(\boldsymbol{\beta}^{(t)})$$

The standard errors of the estimated parameters are the square roots of the diagonal elements of the inverse Fisher Information matrix. Confidence intervals can then be constructed under the assumption that the estimates are approximately normally distributed.
For a confidence level $(1-\alpha)$, the $100(1-\alpha)\%$ confidence interval for each parameter β_j is:

$$\beta_j \pm z_{\alpha/2} \cdot \mathrm{SE}(\beta_j)$$

where $z_{\alpha/2}$ is the $(1-\alpha/2)$ quantile of the standard normal distribution, and $\mathrm{SE}(\beta_j)$ is the standard error of β_j. Let's implement this in R.

Poisson regression and Fisher scoring in R

```r
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)
X <- model.matrix(counts ~ outcome + treatment, d.AD) # Including intercept
y <- d.AD$counts

poisson_fisher_scoring <- function(X, y, max_iter = 100, tol = 1e-8) {
  # Ensure y is a vector | X: Matrix
  n <- nrow(X)
  p <- ncol(X)
  beta <- rep(0, p) # Initialize coefficients
  converged <- FALSE
  for (i in 1:max_iter) {
    eta <- X %*% beta
    mu <- exp(eta)
    # Score funct. (gradient)
    score <- t(X) %*% (y - mu)
    # Fisher Information matrix
    W <- diag(as.vector(mu)) # W is a diag matrix with elements of mu
    fisher_info <- t(X) %*% W %*% X
    # Parameter update
    beta_new <- beta + solve(fisher_info, score)
    # Check convergence
    if (max(abs(beta_new - beta)) < tol) {
      converged <- TRUE
      break
    }
    beta <- beta_new
  }
```

Poisson regression and Fisher scoring in R

```
  if (!converged) {
    warning("Algorithm did not converge")
  }  # SE and confidence intervals:
  se_beta <- sqrt(diag(solve(fisher_info)))
  lower_ci <- beta - qnorm(0.975) * se_beta
  upper_ci <- beta + qnorm(0.975) * se_beta
  return(list(coefficients = beta, se = se_beta, ci_lower = lower_ci, ci_upper = upper_ci))
}
PSF <- poisson_fisher_scoring(X, y)
print(PSF)
>> $coefficients
(Intercept)  3.044522e+00
outcome2    -4.542553e-01
outcome3    -2.929871e-01
treatment2   2.936263e-17
treatment3  -1.687272e-16
>> $se
(Intercept)    outcome2    outcome3  treatment2  treatment3
  0.1708987   0.2021708   0.1927423   0.2000000   0.2000000
>> $ci_lower
(Intercept)  2.7095672
outcome2    -0.8505027
outcome3    -0.6707552
treatment2  -0.3919928
treatment3  -0.3919928
>> $ci_upper
(Intercept)  3.37947764
outcome2    -0.05800787
outcome3     0.08478093
treatment2   0.39199280
treatment3   0.39199280
```

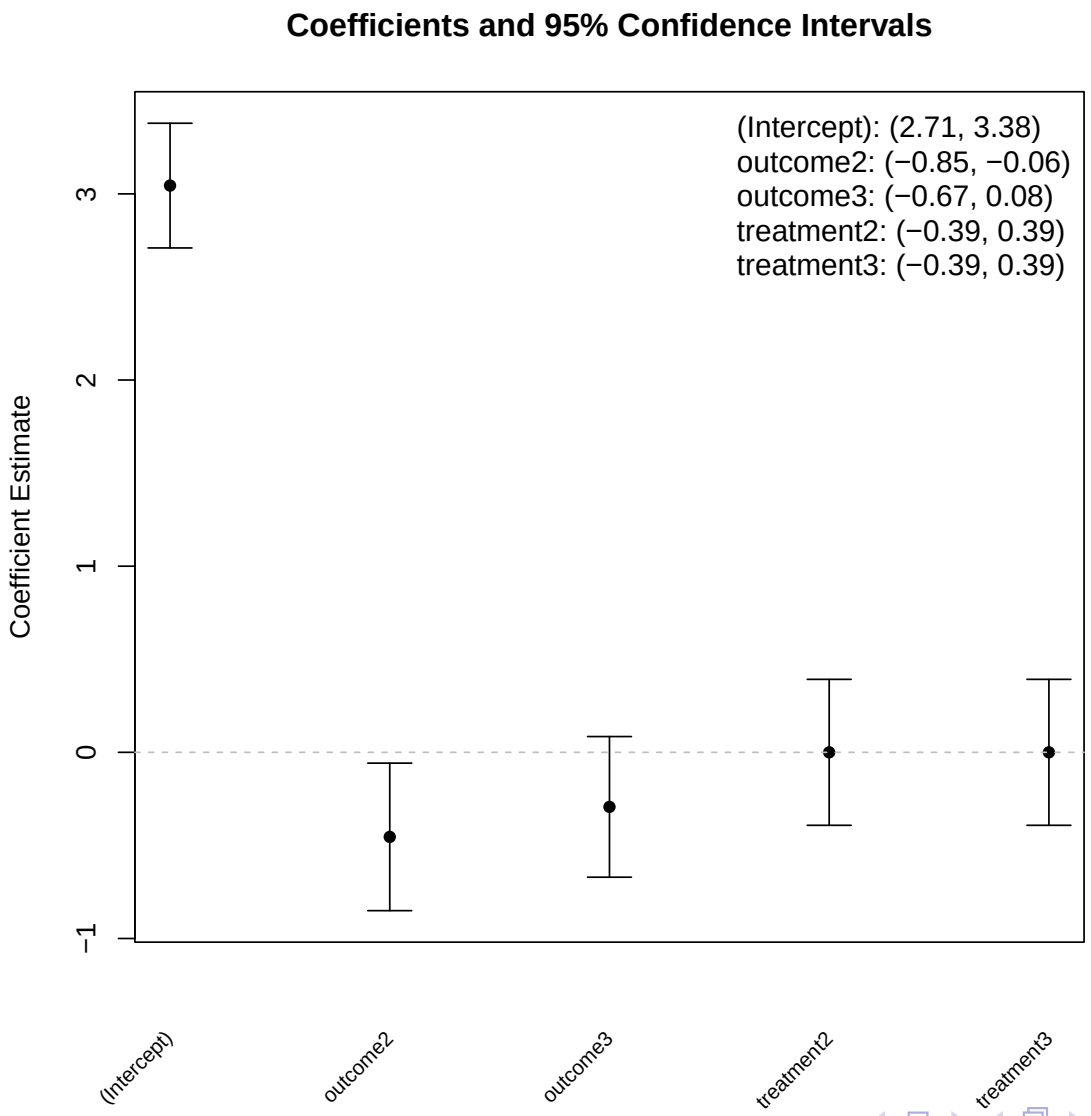
Poisson regression and Fisher scoring in R

95% CI

```r
PSF <- poisson_fisher_scoring(X, y)
# Extracting coefficients and CI
coefficients <- PSF$coefficients
ci_lower <- PSF$ci_lower
ci_upper <- PSF$ci_upper
num_coef <- length(coefficients)
# Set names for coefficients
coef_names <- names(coefficients) <- names(as.data.frame(X))
# Creating a sequence for plotting
coef_seq <- 1:num_coef
plot(coef_seq, coefficients, ylim = range(c(ci_lower, ci_upper)),
     pch = 16, xaxt = 'n', ylab = 'Coefficient Estimate',
     main = 'Coefficients and 95% Confidence Intervals', xlab = "")
# Add CI lines with caps
for(i in coef_seq) {
  lines(x = c(i, i), y = c(ci_lower[i], ci_upper[i]), col = 'black')
  lines(x = c(i - 0.1, i + 0.1), y=c(ci_lower[i], ci_lower[i]),
                                 col='black') # Lower cap
  lines(x = c(i - 0.1, i + 0.1), y=c(ci_upper[i], ci_upper[i]),
                                 col='black') # Upper cap
}
abline(h = 0, col = 'gray', lty = 2)
# Add coefficient names to the x-axis with 45-degree rotation
text(coef_seq, par("usr")[3] - 0.5, labels = names(coefficients),
     srt = 45, adj = 1, xpd = TRUE, cex = 0.8)
# Construct legend text with CI intervals
ci_labels <- paste(coef_names, ": (", sprintf("%.2f", ci_lower),
                   ", ", sprintf("%.2f", ci_upper), ")", sep="")
legend("topright", legend = ci_labels, bty = "n", cex = 1.05)
```

Poisson regression and Fisher scoring in R

95% CI



Comparing with glm(family=poisson())

```
summary(glm.D93)
Call:
glm(formula=counts ~ outcome + treatment, family=poisson())
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
---
```

```
print(confint(glm.D93))
                 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
```

Comparing with glm(family=poisson())

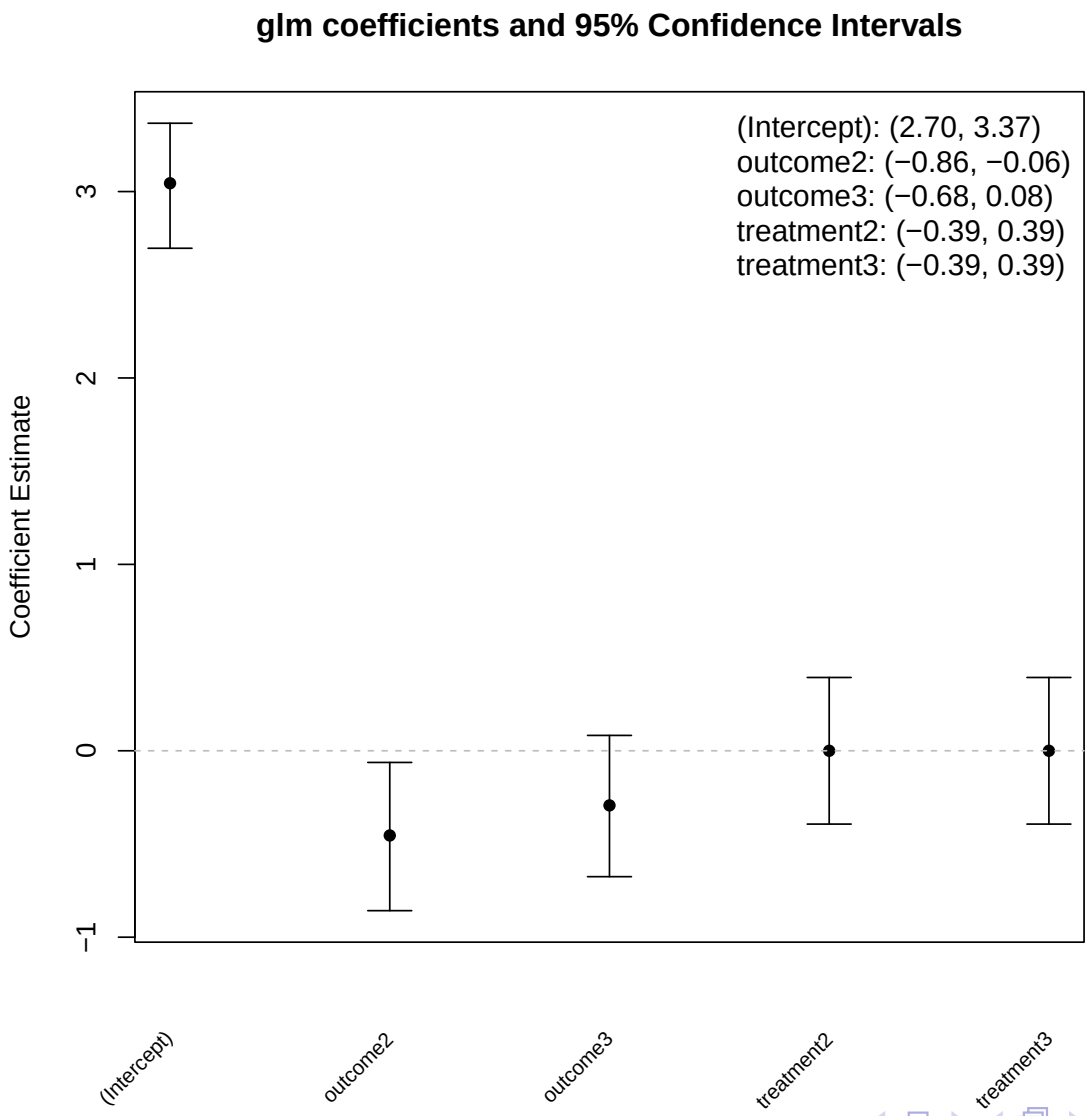
```
summary(glm.D93)
Call:
glm(formula=counts ~ outcome + treatment, family=poisson())
Coefficients:
              Estimate Std. Error z value Pr(>|z|)
(Intercept)  3.045e+00  1.709e-01  17.815   <2e-16 ***
outcome2    -4.543e-01  2.022e-01  -2.247   0.0246 *
outcome3    -2.930e-01  1.927e-01  -1.520   0.1285
treatment2   1.189e-15  2.000e-01   0.000   1.0000
treatment3   8.438e-16  2.000e-01   0.000   1.0000
---
```

```
print(confint(glm.D93))
                 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 closely align with those obtained from our Poisson regression and Fisher scoring analysis.

Comparing with glm(family=poisson())

95% CI



Fisher-scoring vs Newton-Raphson

In the context of Poisson regression:

- **Newton-Raphson**: Would involve calculating the actual Hessian matrix for the log-likelihood function, which is the matrix of second derivatives with respect to the parameters.
- **Fisher Scoring**: Uses the Fisher Information matrix, which in the case of Poisson regression happens to be equal to the expected value of the negative Hessian. This is particularly convenient because the Fisher Information matrix for Poisson regression has a simpler form, as shown earlier.

In summary, for Poisson regression, Newton-Raphson and Fisher Scoring are essentially the same due to the unique properties of the Poisson distribution, where the variance equals the mean, making the Fisher Information matrix identical to the negative Hessian matrix used in Newton-Raphson.

Assignment 4 Part D (bonus)

```
X = read.csv("XMat.csv") # 658, 33
y = read.csv("yvec.csv", header=FALSE) #658, 1
names(X)
>> 'ESS_base', 'BMI', 'COPD', 'HoursPweek', 'Gender', 'Marital', 'factor(Smoke)1', 'factor(
      Smoke)2', 'BECK', 'TRT_bef', 'SEbefPCT', 'TSTbef', 'WASObef', 'N2bef', 'REMlat', 'NREMbef
      ', 'SWSbefpct', 'REM_befpct', 'CentrApnea', 'AHI_bef', 'AHI_REM_bef', 'X.rousalnd', '
      Desathbef', 'SaO2mbef', 'SaO2lbef', 'SaO2minbef', 'SaO285minbe', 'SaO280minbe', 'SBPnight
      _bef', 'DBPnight_be', 'SBPmorn_bef', 'DBPmorn_bef', 'BMIdiff'

length(names(X))
>> 33

library(ggplot2)
library(GGally)

data <- cbind(y, X)

set.seed(42424242)
ggpairs(data[c(1, sample(2:33, 10, replace=F))], # Get randomly 10 variables
  upper = list(continuous = wrap("cor", size = 3.2, color = "darkblue")),
  lower = list(continuous = wrap("smooth", colour = "pink",
                                 fill = "lightblue", alpha = 0.9)),
  diag = list(continuous = wrap("densityDiag", fill = "cornflowerblue",
                                 alpha = 0.85)),
        axisLabels = "show") +
  theme_bw() +
  theme(legend.position = "bottom", axis.text = element_text(size = 12),
        plot.title = element_text(size = 10, hjust = 0.5))
```

Pairplot


V1
AHI_REM_bef
X.rousalnd
SaO2lbef
SEbefPCT
REMlat
SaO2mbef
TRT_bef
WASObef
SBPmorn_bef
SaO2minbef
Corr: 0.000
Corr: 0.034
Corr: −0.013
Corr: 0.112**
Corr: −0.040
Corr: −0.014
Corr: 0.070.
Corr: 0.006
Corr: −0.001
Corr: 0.014
Corr: 0.746***
Corr: −0.691***
Corr: −0.239***
Corr: 0.136***
Corr: −0.668***
Corr: −0.084*
Corr: 0.240***
Corr: 0.183***
Corr: 0.651***
Corr: −0.648***
Corr: −0.311***
Corr: 0.124**
Corr: −0.644***
Corr: −0.061
Corr: 0.341***
Corr: 0.171***
Corr: 0.639***
Corr: 0.156***
Corr: −0.059
Corr: 0.827***
Corr: 0.065.
Corr: −0.181***
Corr: −0.163***
Corr: −0.796***
Corr: −0.357***
Corr: 0.189***
Corr: −0.034
Corr: −0.759***
Corr: −0.104**
Corr: −0.079*
Corr: −0.110**
Corr: 0.205***
Corr: 0.231***
Corr: 0.022
Corr: 0.068.
Corr: 0.107**
Corr: −0.181***
Corr: −0.221***
Corr: −0.809***
Corr: 0.300***
Corr: −0.012
Corr: −0.002
Corr: 0.125**
Corr: 0.142***
Corr: 0.184***

Stepwise VIF-filtering

We first begin with stepwise Variance Inflation Factor (VIF) filtering. The stepwise VIF process begins by calculating VIFs for all predictor variables in a regression model to identify multicollinearity. A threshold is set for VIF, commonly around 15 or 20 (here we use 15), to determine an acceptable level of multicollinearity.

In each step, the variable with the highest VIF, exceeding this threshold, is removed from the model. The VIFs are then recalculated for the remaining variables. This process is repeated iteratively: identifying and removing the highest VIF variable and recalculating VIFs, until all remaining variables have VIFs below the threshold.

Let's build that procedure in R:

Stepwise VIF-filtering in R

```r
library(car)

stepwise_VIF <- function (formula, data, vif_threshold = 15) {
    require(car) # VIF funct.
    continue_selection <- TRUE
    while (continue_selection) {
        model <- lm(formula, data = data)
        vif_values <- vif(model)
        # Ident1fy the predictor with the highest VIF
        worst_predictor <- names(which.max(vif_values))
        max_vif <- max(vif_values)
        cat("Predictor with the highest VIF:", worst_predictor, "VIF:", max_vif, "\n")
        # Check 1f max VIF exceeds threshold
        if(max_vif < vif_threshold) {
            continue_selection <- FALSE
        } else {
            # Update the formula by removing the worst predictor
            formula <- update(formula, as.formula(paste(". ~ . -", worst_predictor)))
            cat("Updated model formula:", deparse(formula), "\n\n") # Updated formula
        }
    }
    final_model <- lm(formula, data=data)
    return (list(formula=formula, model=final_model))
}
step.vif <- stepwise_VIF(as.formula(paste("V1~",  # Full formula
                          paste(names(X), collapse='+'))), data)
```

Stepwise VIF-filtering in R

```
formula.clear <- step.vif$formula

>> Predictor with the highest VIF: SEbefPCT VIF: 29.52243

>> Updated model formula: V1 ~ ESS_base + BMI + COPD + HoursPweek + Gender + Marital +factor.
      Smoke.1 + factor.Smoke.2 + BECK + TRT_bef + TSTbef +  WASObef + N2bef + REMlat + NREMbef
      + SWSbefpct + REM_befpct + CentrApnea + AHI_bef + AHI_REM_bef + X.rousalnd + Desathbef +
      SaO2mbef + SaO2lbef + SaO2minbef + SaO285minbe + SaO280minbe + SBPnight_bef + DBPnight_be
       + SBPmorn_bef + DBPmorn_bef + BMIdiff

>> Predictor with the highest VIF: Desathbef VIF: 18.12557

>> Updated model formula: V1 ~ ESS_base + BMI + COPD + HoursPweek + Gender + Marital + factor.
      Smoke.1 + factor.Smoke.2 + BECK + TRT_bef + TSTbef + WASObef + N2bef + REMlat + NREMbef +
       SWSbefpct + REM_befpct + CentrApnea + AHI_bef + AHI_REM_bef + X.rousalnd + SaO2mbef +
      SaO2lbef + SaO2minbef + SaO285minbe + SaO280minbe + SBPnight_bef + DBPnight_be + SBPmorn_
      bef + DBPmorn_bef + BMIdiff

>> Predictor with the highest VIF: SaO285minbe VIF: 9.080881
```

We observe that two predictors were removed following the stepwise filtering: `SEbefPCT` and `Dasathbef`. Consequently, we reduced p from 33 to 31.

Interaction terms & Feature engineering

```
# Build squared predictors
squared_preds <- function(X, symbol="^2") {
  for (i in 1:ncol(X)) {
    col_name <- names(X)[i]
    X[paste0("I(", col_name, symbol, ')')] <- X[[col_name]]^2
}
  return(X) # Returns first-order terms + squared predictors
}
interactions_xixj <- function(X, symbol=":") {
    X_interaction <- data.frame(matrix(ncol = 0, nrow = nrow(X)))
    # Iterate through each pair of columns (i!=j)
    for (i in 1:ncol(X)) {
        for (j in 1:ncol(X)) {
            if (i != j) {
                # Calculate the interaction term for the current pair of columns
                interaction_term <- X[[i]] * X[[j]]
                # Generate the column name for the interaction term
                interaction_name <- paste0(names(X)[i], symbol, names(X)[j])
                # Add the interaction term to the 'X_interaction' data frame
                X_interaction[[interaction_name]] <- interaction_term
            }
        }
    }
    return(X_interaction) # Returns ALL the X_i : X_j
}
```

Interaction terms & Feature engineering

```
# 2 predictors removed
X.clear <- as.data.frame(model.matrix(formula.clear, data)[,-1])
interactions.2ord <- paste(names(cbind(squared_preds(X.clear),
                                       interactions_xixj(X.clear))),
                           collapse = "+")
formula.2ord <- as.formula(paste("V1~", interactions.2ord)) #V1 refers to the response
```

How many predictors are there in total? Recall that we have $p = 33$, hence:

$$\texttt{Total} = \underbrace{\binom{p}{2}}_{X_i:X_j;\ i\neq j} + \underbrace{p}_{X_i} + \underbrace{p}_{X_i^2} = \underbrace{\binom{31}{2}}_{X_i:X_j;\ i\neq j} + \underbrace{31}_{X_i} + \underbrace{31}_{X_i^2} = 527.$$

```
length(names(model.matrix(formula.2ord, data)[, -1]))  # Check size
>> 527
```

Forward-stepwise feature selection using AIC

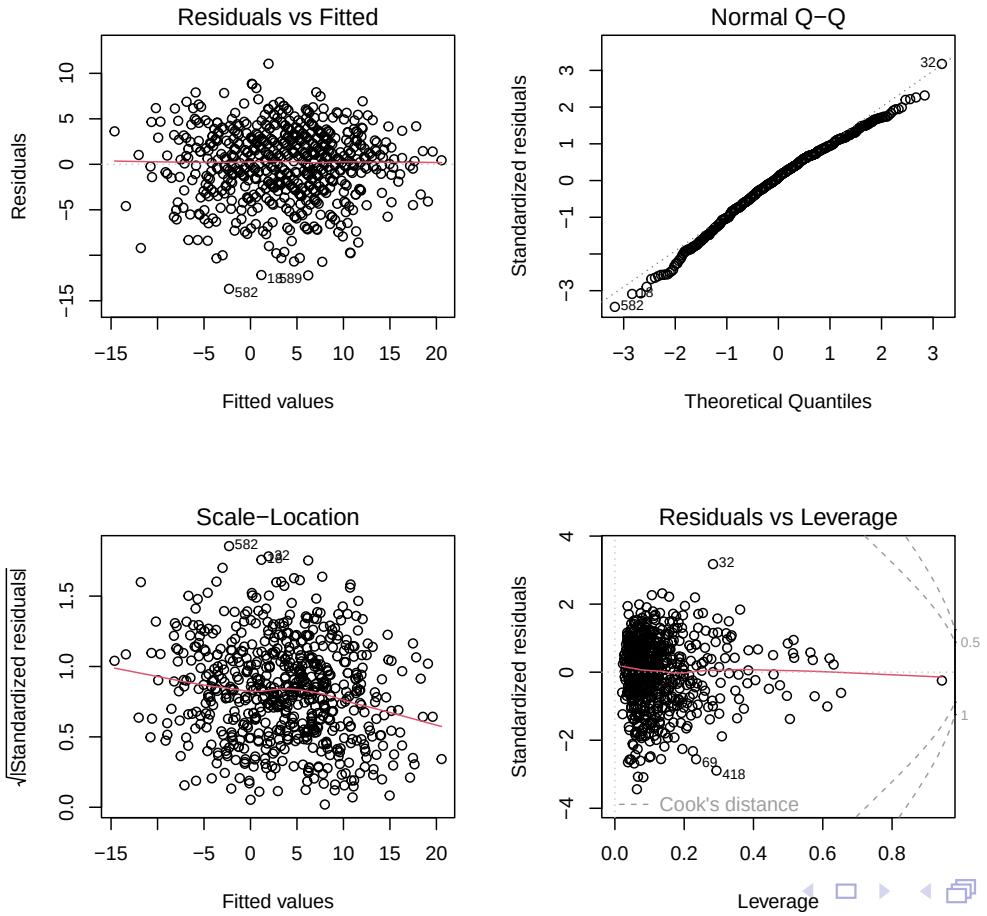
```
library(olsrr)
ols.2o <- lm(formula.2ord, data)
FAIC.selection <- ols_step_forward_aic(ols.2o)
>>
                                    Selection Summary
--------------------------------------------------------------------------------
Variable                          AIC        Sum Sq         RSS        R-Sq      Adj. R-Sq
--------------------------------------------------------------------------------
ESS_base:SaO2lbef               3980.513    18147.748    16181.463    0.52864      0.52792
COPD:Gender                     3963.264    18614.269    15714.942    0.54223      0.54083
HoursPweek:Marital              3953.346    18896.345    15432.866    0.55045      0.54838
factor.Smoke.2:SaO280minbe      3946.442    19103.775    15225.436    0.55649      0.55377
SaO2mbef:SBPnight_bef           3938.117    19340.806    14988.406    0.56339      0.56004
BECK:SaO280minbe                3930.639    19555.166    14774.045    0.56964      0.56567
...                                ...         ...          ...         ...          ...
BMI:SaO280minbe                 3808.936    24523.267     9805.944    0.71436      0.67475
NREMbef:BMIdiff                 3807.359    24576.428     9752.783    0.71590      0.67595
factor.Smoke.1:X.rousalnd       3807.324    24606.546     9722.665    0.71678      0.67639
--------------------------------------------------------------------------------
FAIC.selection$steps # Predictors after forward AIC
>> 82
```

- First-order terms: 3; `TRT_bef`, `SaO280minbe`, `BMI`.
- Second-order terms: 74; `ESS_base:SaO2lbef`, `COPD:Gender`, `HoursPweek:Marital`, ... (71 more...)
- Squared terms: 5; `I(AHI_REM_bef^2)`, `I(DBPnight_be^2)`, `I(CentrApnea^2)`, `I(HoursPweek^2)`, `I(TRT_bef^2)`.

Residual diagnostics

Examining autocorrelation & homoscedasticity

```
formula.faic <- as.formula(paste("V1 ~", paste(FAIC.selection$predictors, collapse='+')))
ols.model <- lm(formula.faic, data)
par(mfrow=c(2,2))
plot(ols.model)
```



Residual diagnostics

Examining **autocorrelation** & homoscedasticity

Recall that we can examine serial correlation (also known as autocorrelation, or time series data) using the Durbin–Watson test.

Durbin–Watson statistic

In statistics, the Durbin–Watson statistic is a test statistic used to detect the presence of autocorrelation. if e_t is the residual given by $e_t = \rho e_{t-1} + v_t$, the DW test statistic is

$$d = \frac{\sum_{t=2}^{T}(e_t - e_{t-1})^2}{\sum_{t=1}^{T} e_t^2},$$

where T is the number of observations. For large T, $d \approx 2(1 - \hat{\rho})$, where $\hat{\rho}$ is the sample autocorrelation of the residuals. (Source)

```
library(lmtest)
dw_test_result <- dwtest(ols.model)
dw_test_result #DW stat. close to 2; p-value relative small => Most likely no autocorr pres.
>>   Durbin-Watson test

data:  ols.model
DW = 1.8932, p-value = 0.08694
alternative hypothesis: true autocorrelation is greater than 0
```

Residual diagnostics

Examining autocorrelation & **homoscedasticity**

```
ols_test_breusch_pagan(ols.model)
>>
 Breusch Pagan Test for
    Heteroskedasticity
 --------------------------------
 Ho: the variance is constant
 Ha: the variance is not constant

              Data
 --------------------------------
 Response : V1
 Variables: fitted values of V1

         Test Summary
 --------------------------------
 DF              =    1
 Chi2            =    6.649429
 Prob > Chi2     =    0.009918761
 #P=0.01>0.005 => can't reject N_0
```

Breusch Pagan Test[ab]

- One of the key assumptions of linear regression is that the residuals are distributed with equal variance at each level of the predictor variable. AKA **homoscedasticity**.
- When this assumption is violated, we say that **heteroscedasticity** is present in the residuals.

[a] $\mathbf{H_0}$: **Homoscedasticity** is present (the residuals are distributed with equal variance)

[b] $\mathbf{H_A}$: **Heteroscedasticity** is present (the residuals are not distributed with equal variance)

Summary of methodological tools & model evaluation

- **Feature engineering**:
  1. **Stepwise-VIF-filtering**.
  2. **Generate second-order terms**.
  3. **Forward-stepwise** `AIC` (82 predictors).
- **Model evaluation**:
  - **Repeated-Nested Cross-Validation + Bootstrap CI for**:
  - **Penalized regression**
    - **Penalized LS Lasso**
    - **Penalized LS Ridge**
    - **Adaptive LAD Lasso** (`rqpen`)
    - **Adaptive Lasso** (`glmnet`)
  - **Linear & non-linear kernels (Radial-Basis, polynomial, sigmoid)**
    - **Support Vector Machine (SVR)**
  - **Ensemble learning**
    - **Random Forest**
    - **XGboost**
    - **Adaboost**
  - **Deep learning**
    - **Multi-layer Perceptron (MLP) Neural Networks**

Define some performance metrics

```
# log acc. ration
LAR <- function(y, forcast) {
  n <- length(y)
  lar <- 0
  n_ <- 0 #counter
  for (i in 1:n) {
    if (y[i] != 0) {
      Q = as.numeric(forcast)[i]/y[i]
        if (Q > 0) {
          lar = lar + abs(log(Q))
          n_ = n_ + 1
        }
    }
  }
  lar <- lar/n_
  return(lar)
}
MAPE <- function(y, forcast) {
  n <- length(y)
  mape_sum <- 0
  n_ <- 0 #counter
  for (i in 1:n) {
    if (y[i] != 0) {
      mape_sum = mape_sum + abs((y[i]-as.numeric(forcast)[i])/y[i])
      n_ = n_ + 1
    }
  }
  mape <- (mape_sum/n_) * 100
  return(mape)
}
```

Define some performance metrics

```
MAE <- function(y, forecast) {
  finite_indices <- is.finite(y) & is.finite(forecast)
  if (any(finite_indices)) {
    mean(abs(y[finite_indices] - forecast[finite_indices]), na.rm = TRUE)
  } else {
    NA  # Failure
  }
}
RMSE <- function(y, forecast) {
  finite_indices <- is.finite(y) & is.finite(forecast)
  if (any(finite_indices)) {
    sqrt(mean((y[finite_indices] - forecast[finite_indices])^2, na.rm = TRUE))
  } else {
    NA  # Or some other default value indicating failure
  }
}
sMdAPE <- function(y, forecast) {
  finite_indices <- is.finite(y) & is.finite(forecast) & (y + forecast != 0)
  if (any(finite_indices)) {
    median(200*abs(y[finite_indices]-forecast[finite_indices])/(y[finite_indices] +
                                                  forecast[finite_indices]),
                                                  na.rm = TRUE)
  } else {
    NA  # Failure
  }
}
```

Cross-validation; generalized function

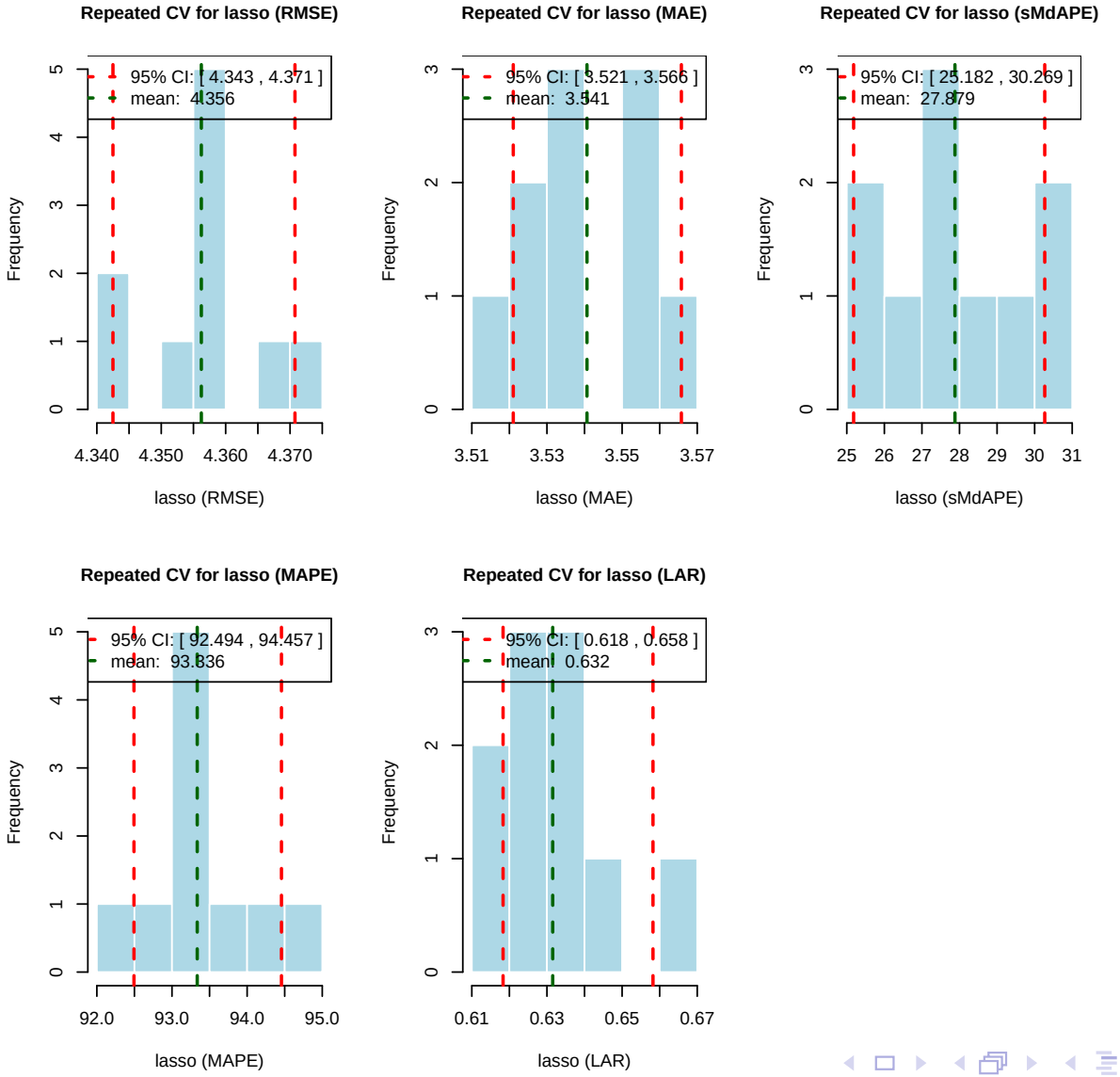
We modify `KfoldCVPerf.general()` from here (*page* 7/88)

```
KfoldCVPerf.general <- function (K=50, data, formula, ..., criterion="AIC") {
    # Loop through the folds
    for (i in 1:K) {
        #Keep the i-th fold as it is (testing)
        val_fold <- folds_list[[i]]
        y_val <- val_fold[[y]]

        ### ... same code ... ###

        } else if (method == "rf") { # Random Forest
            require(randomForest)
            model <- randomForest(formula, data=train_folds, ntree=500)
        } else if (method == "xgboost") {
            require(xgboost)
            # XGBoost model
            model <- xgboost(data = model.matrix(formula, train_folds)[, -1],
                             label = as.vector(y_train),
                             nrounds = 1000,  # Number of boosting rounds
                             eval_metric = "mae",
                             subsample = 0.8,
                             colsample_bytree = 0.8)
        } else {
            stop("Unsupported method or criterion.")
        }
        ### ... same code ... ###
    return (perf_df)
}
```

We also utilize the `RCV.histograms()` and `get.bootstrapCI()` (*page* 15/88 & 28/88).

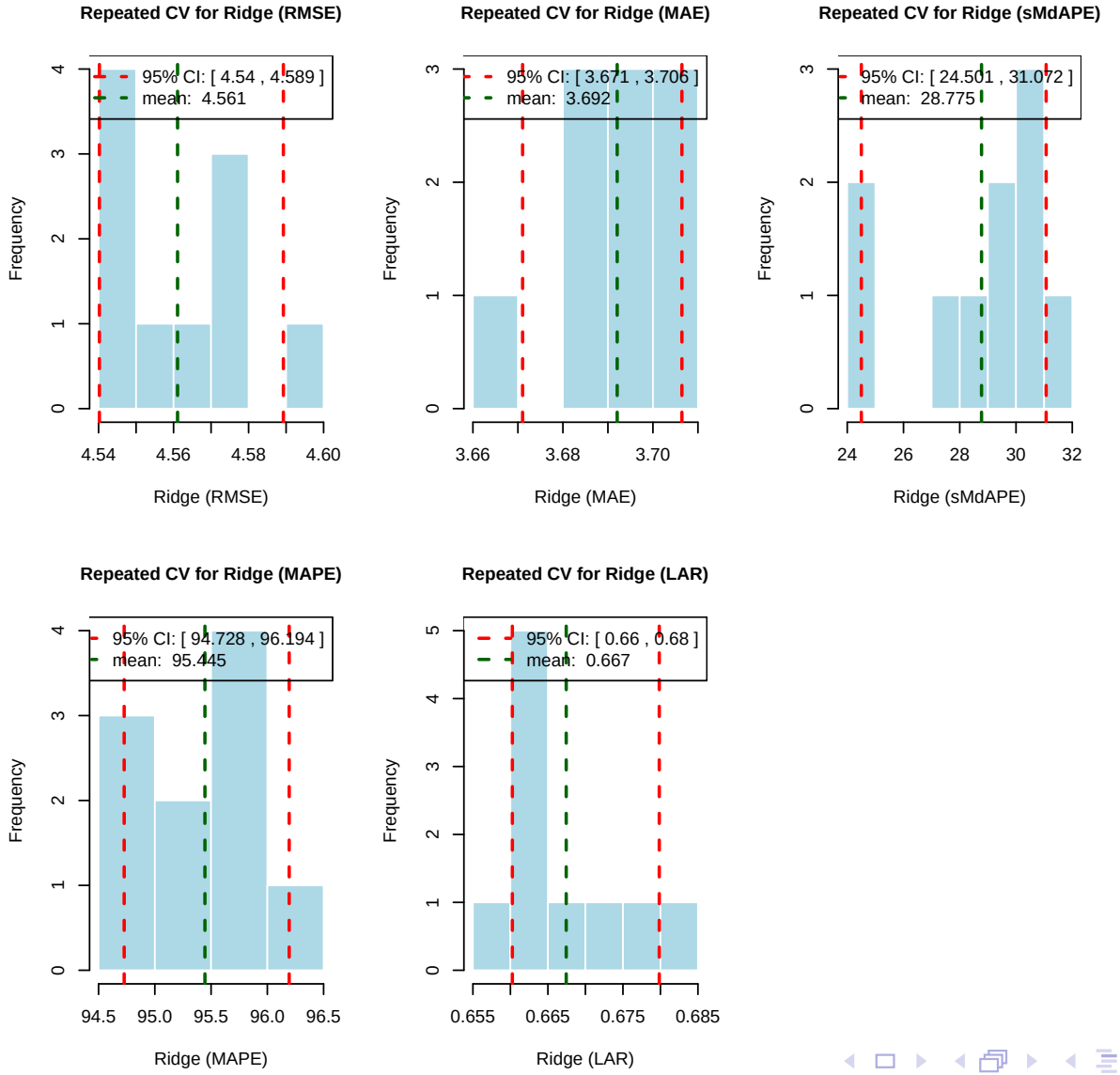
Repeated cross-validation - Penalized LS - Lasso

based on 50-fold cross-validation and 10 repeats



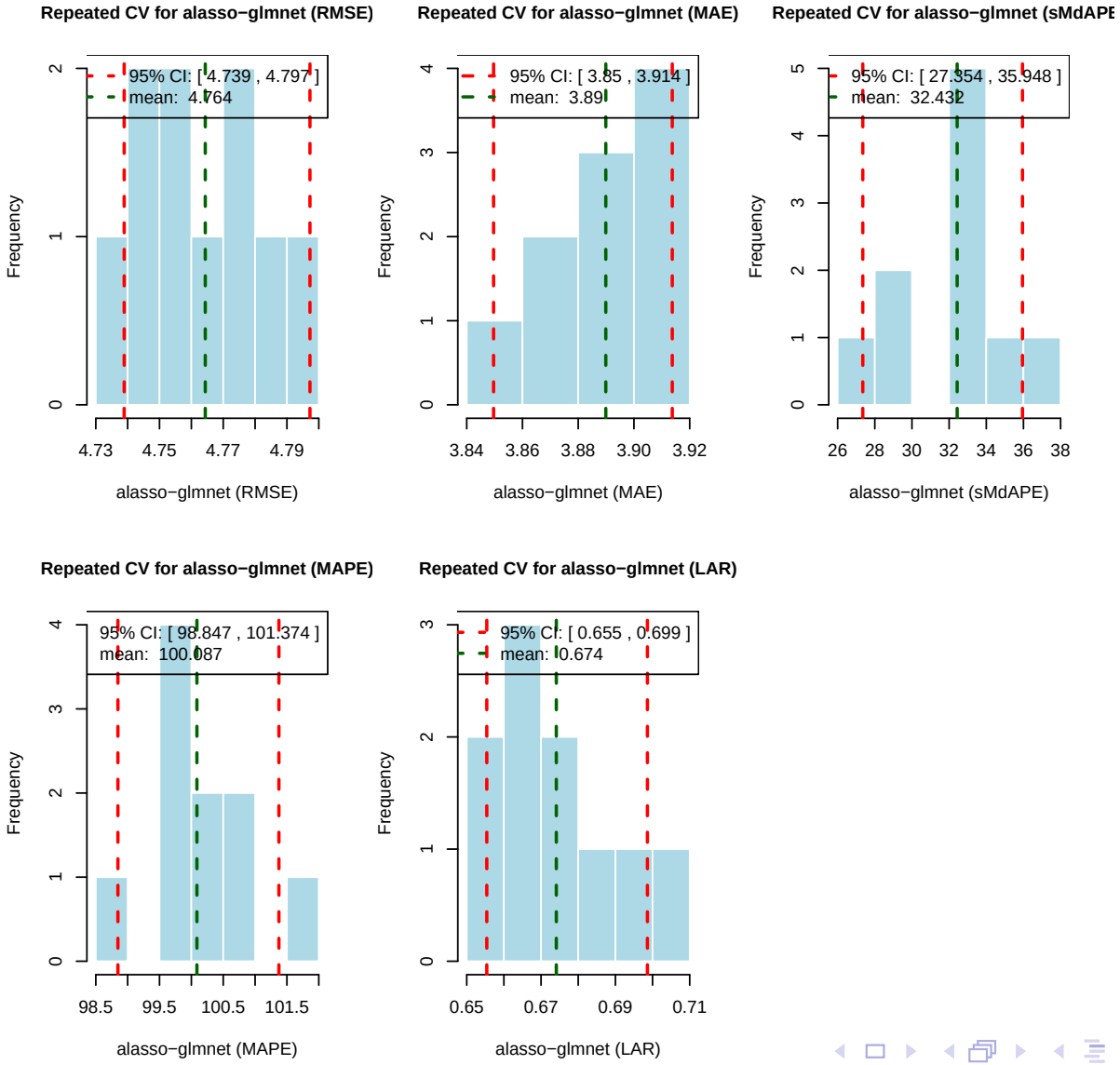
Repeated cross-validation - Penalized LS - Ridge

based on 50-fold cross-validation and 10 repeats



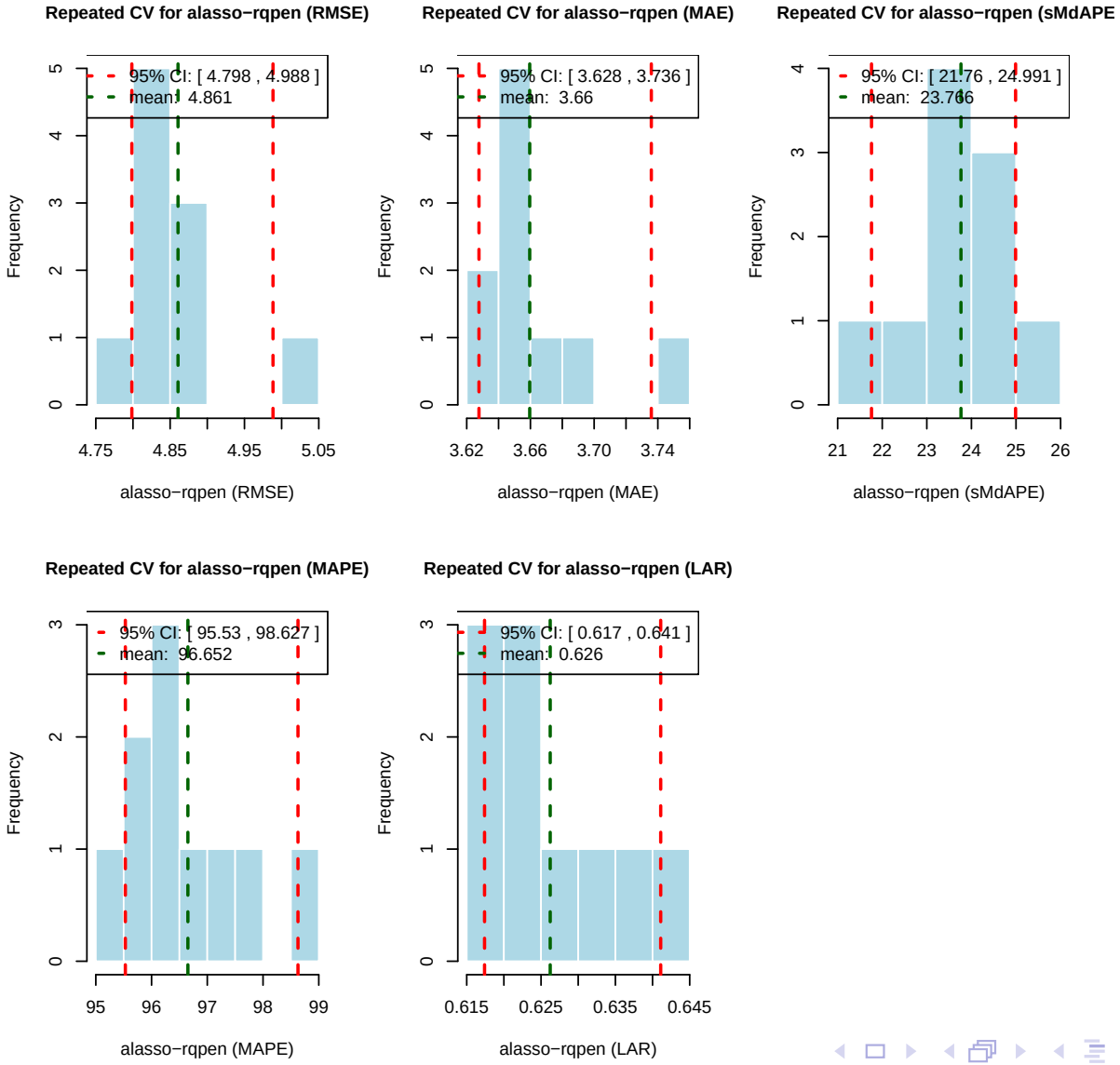
Repeated cross-validation - Penalized LS - Adaptive Lasso

based on 50-fold cross-validation and 10 repeats



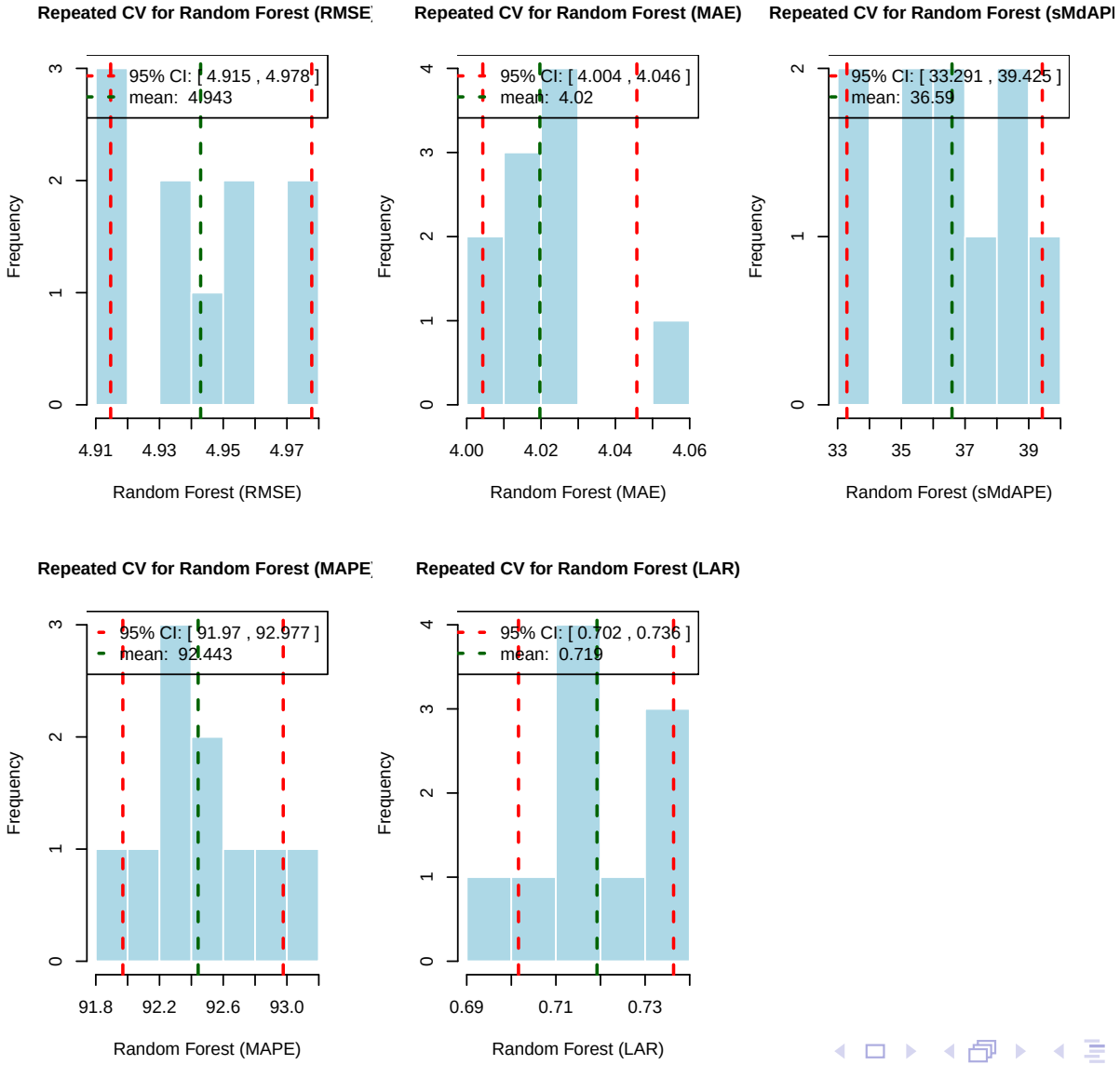
Repeated cross-validation - Penalized adaptive-LAD Lasso

based on 50-fold cross-validation and 10 repeats (≈18½ hours required)



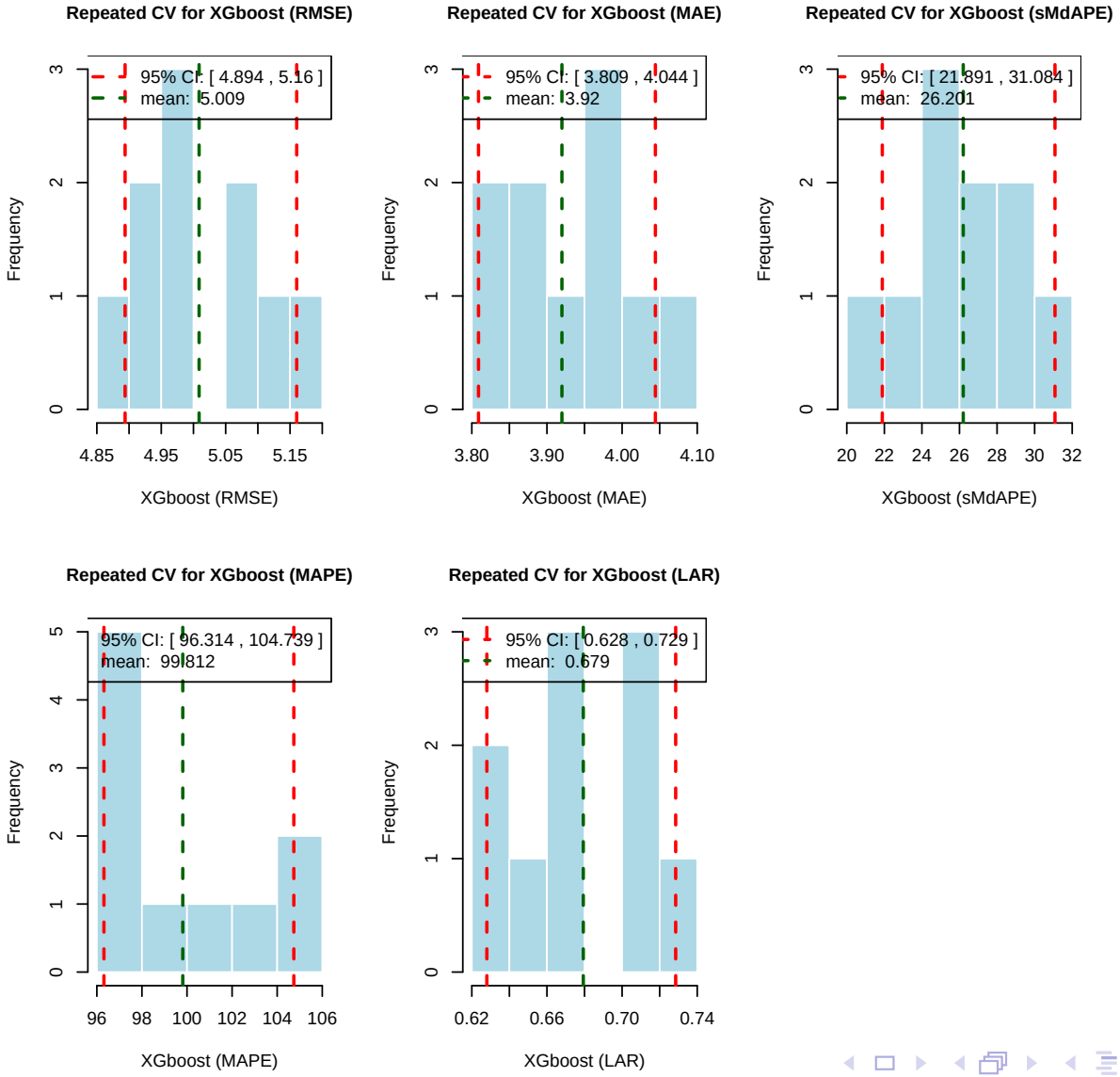
Repeated cross-validation - Random Forest

based on 50-fold cross-validation and 10 repeats



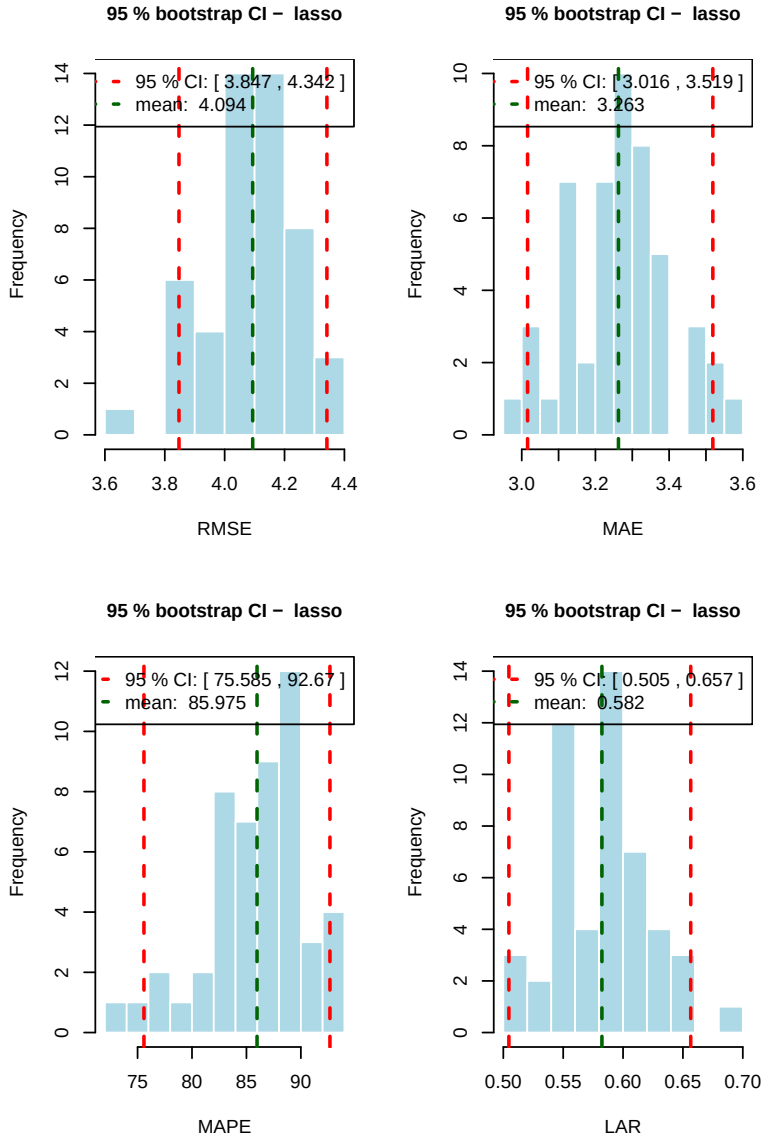
Repeated cross-validation - XGboost

based on 50-fold cross-validation and 10 repeats



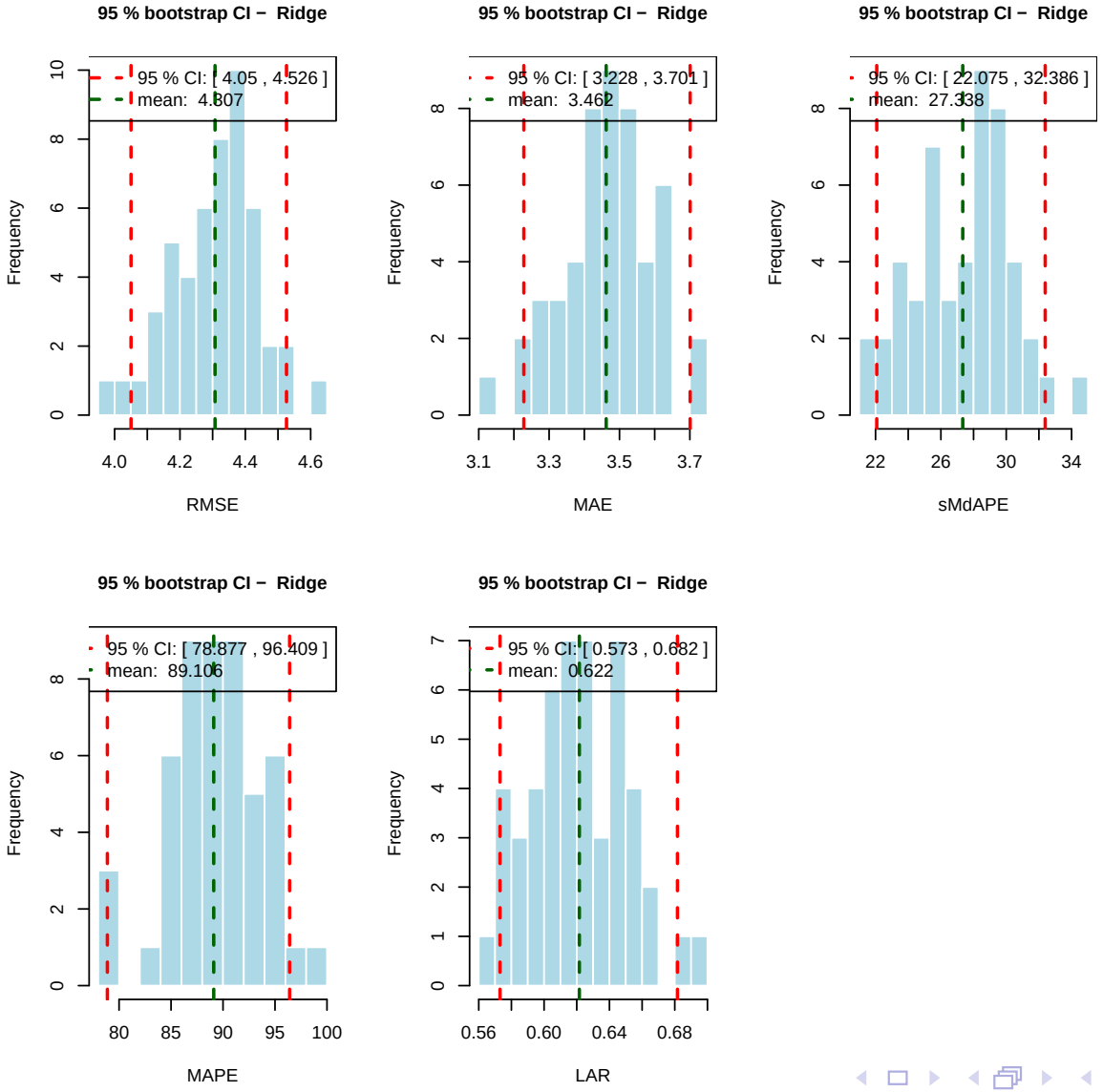
Bootstrap 95% CI - Penalized LS - Lasso

based on 50-fold cross-validation & 50 bootstrap samples



Bootstrap 95% CI - Penalized LS - Ridge

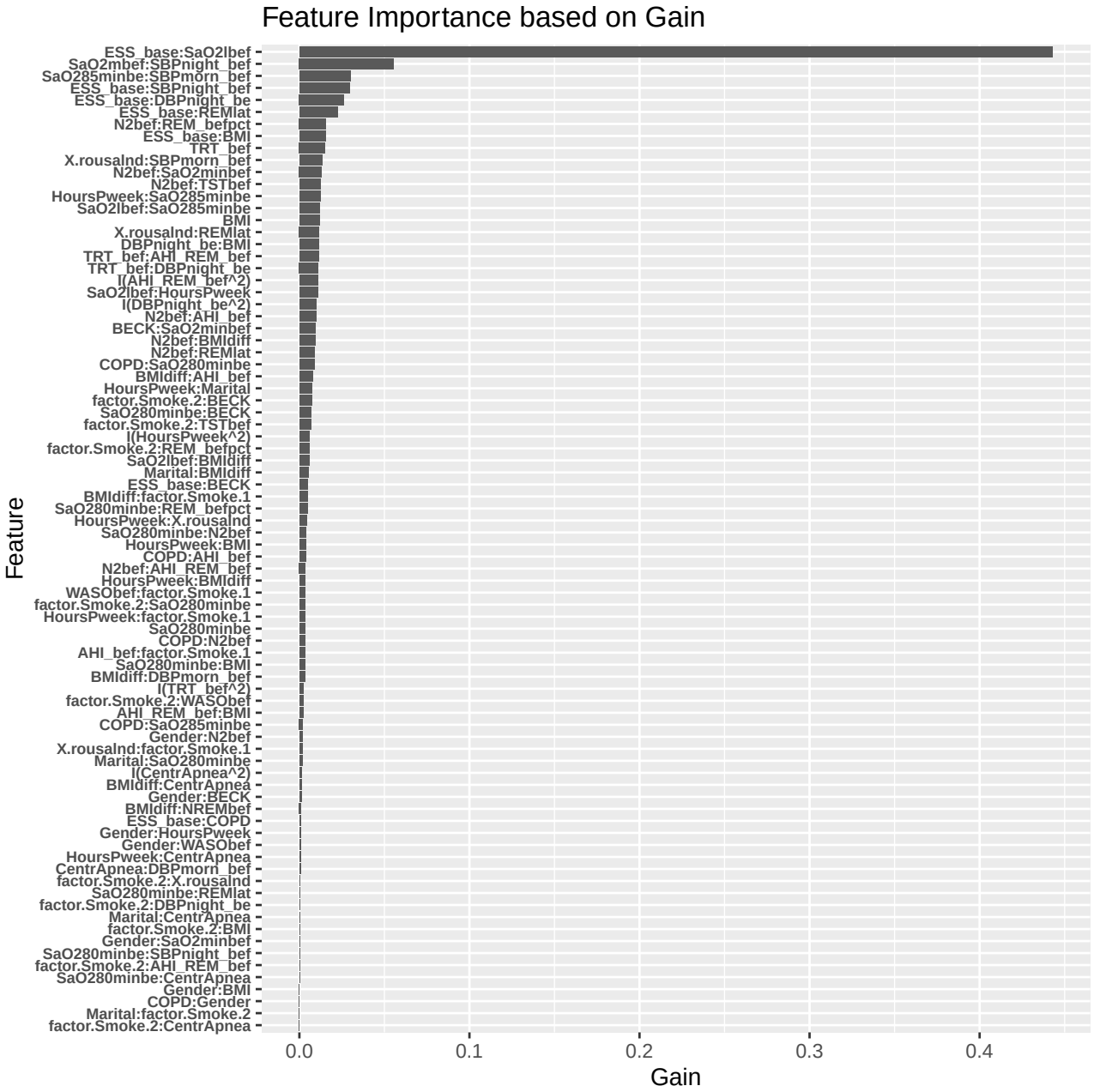
based on 50-fold cross-validation & 50 bootstrap samples



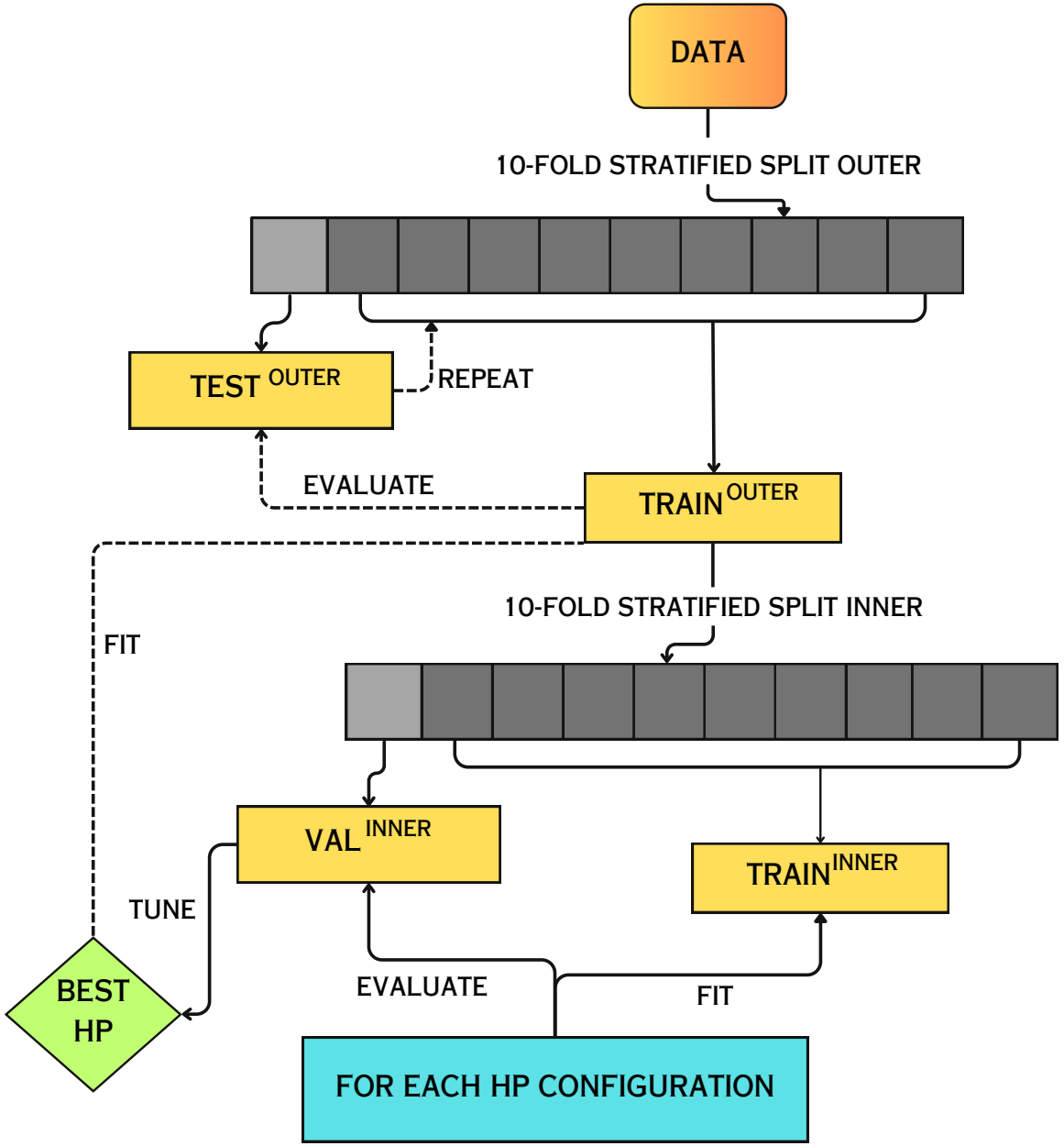
XGboost feature importance based on information gain

```
library(xgboost)
xgb_model <- xgboost(data = model.matrix(formula.faic, data)[,-1],
                     label = as.numeric(unlist(y)),
                     nrounds = 1000,  # Number of boosting rounds
                     eval_metric = "mae",
                     subsample=0.8,
                     colsample_bytree= 0.8)

plot_feature_importance <-  function (model) {
  require(ggplot2)
  require(xgboost)

  importance_matrix <- xgb.importance(model = model)
  importance_matrix <- importance_matrix[order(importance_matrix$Gain, decreasing = TRUE), ]

  #Plot the feature importance bars
  ggplot(importance_matrix, aes(x = reorder(Feature, Gain), y = Gain)) +
    geom_bar(stat = "identity") +
    coord_flip() +
    xlab("Feature") +
    ylab("Gain") +
    ggtitle("Feature Importance based on Gain") +
    theme(axis.text.y = element_text(size = 7.15, face = "bold"))
}


plot_feature_importance(xgb_model)
```

XGboost feature importance based on information gain



Nested cross-validation diagram



Nested cross-validation class in

The class presented here is an adapted version of the nested cross-validation technique originally developed in my B.Sc. thesis. While the thesis focused on its application to binary classification problems, this revised version has been generalized to encompass all types of regression problems, including those involving deep learning. We need to import some necessary libraries:

```
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.pipeline import Pipeline
from sklearn.utils import resample #Bootstrapping
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.model_selection import KFold, GridSearchCV
from sklearn.metrics import make_scorer, mean_absolute_error, mean_squared_error
from numpy import sqrt
from collections import defaultdict
import time

data = pd.read_csv("interactions.csv") # Data with the forward.AIC interactions from R
y = data[["V1"]]
X = data.iloc[:, 1:]
```

Nested cross-validation class in

```python
class NestedCV:

    def __init__(self, innercv: int = 10, outercv: int = 10):
        self.innercv = innercv
        self.outercv = outercv
        self.best_hp_list = []

    def __repr__(self):
        return f"NestedCV(inner loops: {self.innercv}, outer loops: {self.outercv})"

    def fit(self, X: pd.DataFrame, y: pd.DataFrame, pipeline: Pipeline,
            grid_param: dict, trace: bool = True, njobs: bool = False):

        X = np.array(X)
        y = np.ravel(y)

        scoring_metrics = {
            'MAE': make_scorer(mean_absolute_error, greater_is_better=False),
            'MSE': make_scorer(mean_squared_error, greater_is_better=False)
        }
        self.nested_cv_scores = defaultdict(list)

        inner_cv = KFold(n_splits=self.innercv, shuffle=True, random_state=5666)
        outer_cv = KFold(n_splits=self.outercv, shuffle=True, random_state=5666)

        for i, (train_idx, test_idx) in enumerate(outer_cv.split(X, y)):
            outer_start_time = time.time()
            X_train, X_test = X[train_idx], X[test_idx]
            y_train, y_test = y[train_idx], y[test_idx]
```

Nested cross-validation class in

```
grid_search_params = {'cv': inner_cv,
                      'scoring': scoring_metrics,
                      'refit': 'MSE',
                      'return_train_score': True,
                      'n_jobs': -1 if njobs else None}
inner_cv_search = GridSearchCV(estimator=pipeline, param_grid=grid_param, **grid_search_params)
inner_cv_search.fit(X_train, y_train)
self.best_params = inner_cv_search.best_params_
self.best_hp_list.append(self.best_params)

y_pred = inner_cv_search.predict(X_test)
for metric in scoring_metrics.keys():
    if metric == 'MSE':
        score = mean_squared_error(y_test, y_pred)
        metric_name = 'RMSE'
        score = sqrt(score)  # Convert MSE to RMSE
    else:
        score = mean_absolute_error(y_test, y_pred)
        metric_name = metric

    self.nested_cv_scores[metric_name].append(score)

    if trace:
        print(f"Outer fold {i+1} {metric_name}: {score:.3f}")

if trace:
    print(f"Best hyperparameters: {self.best_params}")
    percentage_done = (i + 1) / outer_cv.n_splits * 100
    print(f"{percentage_done:.2f}% of the procedure is complete\n")

outer_end_time = time.time()
outer_time_elapsed = outer_end_time - outer_start_time
```

Nested cross-validation class in

```
        if trace:
            mins, secs = divmod(outer_time_elapsed, 60)
            outer_time = f"{int(mins)} min and {secs:.2f} sec"
            print(f"Time taken for outer-fold-{i+1}: {outer_time}\n")

    self.mean_outer_cv_scores = {metric: np.mean(scores) for metric, scores \
                                                                in self.nested_cv_scores.items()
                                                                }

    last_step_name = list(pipeline.named_steps.keys())[-1]
    self.model_name = pipeline.named_steps[last_step_name].__class__.__name__
    self.pipe = pipeline
    self.params = grid_param

def best_hp(self):
    best_hp_df = pd.DataFrame(self.best_hp_list)
    best_hp_df.index = [f"Outer Fold {i+1}" for i in range(self.outercv)]
    return best_hp_df

def performance(self):
    mean_scores_df = pd.DataFrame(self.mean_outer_cv_scores,
                                  index=[f'\'{self.model_name}\' NestedCV Performance']).T
    return mean_scores_df
```

Nested CV: Support Vector Machine (**SVR**) (≈1h5min required)

```
from sklearn.svm import SVR
# SVR pipeline
SVR_pipe = Pipeline([
    ('Scaler', StandardScaler()),
    ('regressor', SVR(gamma='scale'))
])
# HP to tune
SVR_grid_param = {
    'regressor__C': [1, 10, 50, 100],
    'regressor__kernel': ['rbf', 'linear', 'poly'],
    'regressor__epsilon': [0.1, 1],
    #'regressor__gamma': ['scale', 'auto']
}
NestedCV_SVR = NestedCV(innercv=10, outercv=50)
NestedCV_SVR.fit(X, y, SVR_pipe, SVR_grid_param, njobs=True) # njobs=True => parallel prog.
NestedCV_SVR.best_hp()
>>                          C                epsilon          Kernel
>> Outer Fold 1             10                 1.0            linear
>> Outer Fold 2             10                 1.0            linear
>> Outer Fold 3             10                 1.0            linear
>> Outer Fold 5            100                 1.0            linear
>> ...                     ...                 ...             ...
>> Outer Fold 49            10                 1.0            linear
>> Outer Fold 50            50                 1.0            linear
NestedCV_SVR.performance()
```

	'SVR' NestedCV Performance
MAE	3.529010
RMSE	4.464627

Nested CV: AdaBoost (≈1h required)

```
from sklearn.ensemble import AdaBoostRegressor

AB_pipe = Pipeline([
    ('Scaler', StandardScaler()),
    ('regressor', AdaBoostRegressor(random_state=42))
])
param_grid = {
    'regressor__n_estimators': [100, 1000],
    'regressor__learning_rate': [0.01, 0.5],
    'regressor__loss': ['linear', 'square', 'exponential']
}

NestedCV_AB = NestedCV(innercv=10, outercv=50)
NestedCV_AB.fit(X, y, AB_pipe, param_grid, njobs=True)

NestedCV_AB.performance()
```

'AdaBoostRegressor' NestedCV Performance	
MAE	4.160510
RMSE	4.947018

Nested CV: Multi-layer Perceptron (MLP) Neural Networks

```
from sklearn.neural_network import MLPRegressor

# Creating a neural network pipeline
NN_pipe = Pipeline([
    ('Scaler', StandardScaler()),
    ('regressor', MLPRegressor(random_state=42, max_iter=5000))
])

NN_grid_param = {
    'regressor__hidden_layer_sizes': [(40,), (50,), (50, 25), (40, 20), (100, 50)],
    'regressor__activation': ['tanh', 'relu'],
    'regressor__solver': ['sgd', 'adam'],
    'regressor__alpha': [0.001, 0.01]
}

NestedCV_NN = NestedCV(innercv=10, outercv=10)
NestedCV_NN.fit(X, y, NN_pipe, NN_grid_param, njobs=True)

NestedCV_NN.performance()
```

'MLPRegressor' NestedCV Performance	
MAE	4.515212
RMSE	5.685101

Conclusions

We can observe that our top three models, in terms of MAE, are the following:

1. **Penalized linear regression with Lasso:**
   - **Repeated** 50 fold CV 95% CI: (3.52, 3.56) with average MAE=3.540.
   - **Bootstrap** 50 samples 95% CI: (3.01, 3.51) with average MAE=3.263.
2. **Linear based kernel SVM:**
   - **Nested CV MAE performance**: 3.529.
3. **Penalized adaptive LAD Lasso:**
   - **Repeated** 50 fold CV 95% CI: (3.62, 3.73) with average MAE=3.66.

Further investigation: Bootstrapping `StepAIC()`

We can employ the `` `BootStepAIC` `` package to use bootstrap and identify the most frequently occurring predictors. Some theory behind the package:

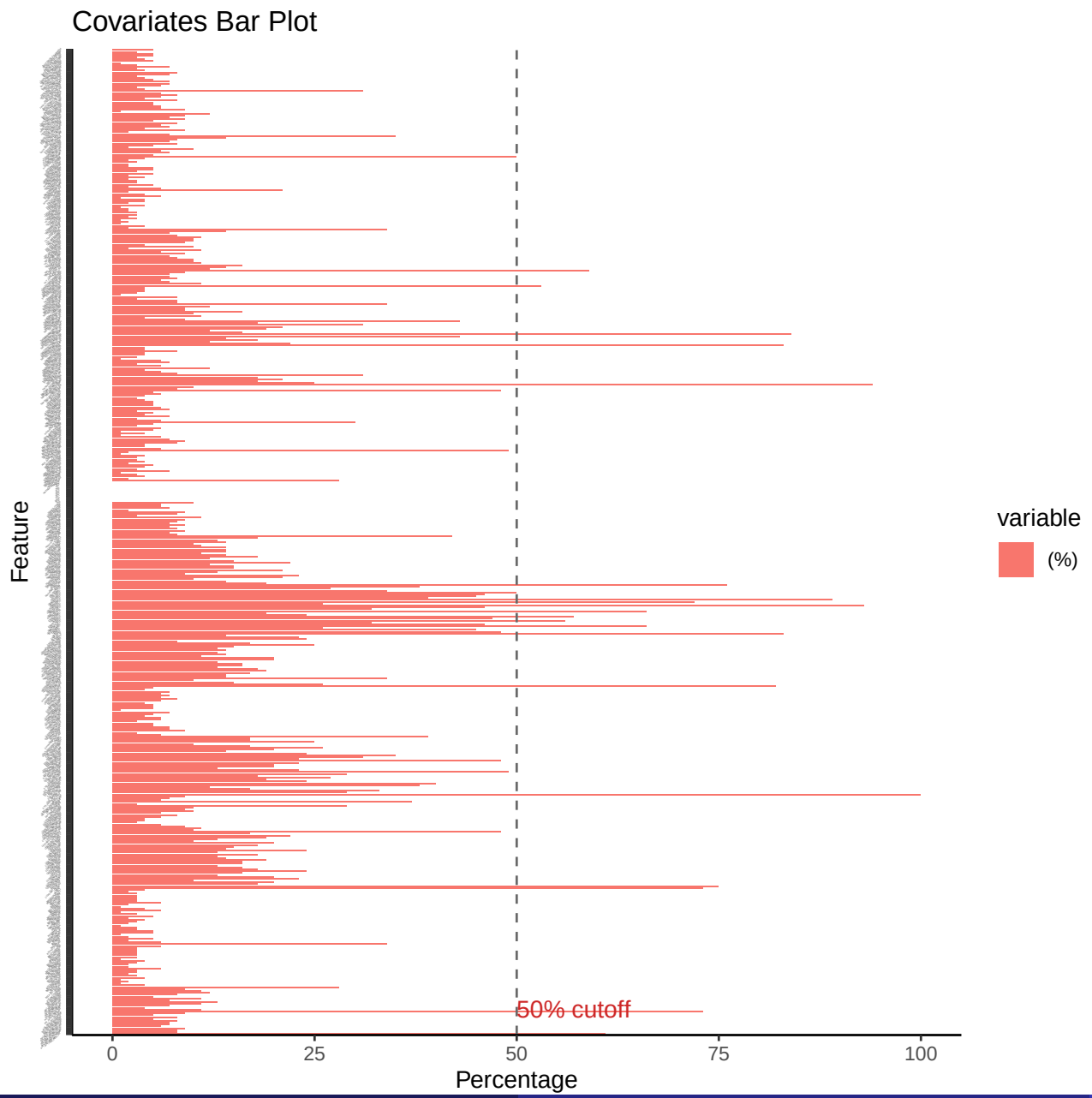
- `$`**Covariates:** A matrix indicating the percentage of times each variable was selected across the bootstrap samples.
- `$`**Significance:** Shows the percentage of times the regression coefficient of each variable was statistically significant at the specified alpha level.
- `$`**Sign:** Indicates the percentage of times the sign of the regression coefficient for each variable was positive or negative.

Further investigation: Bootstrapping StepAIC()

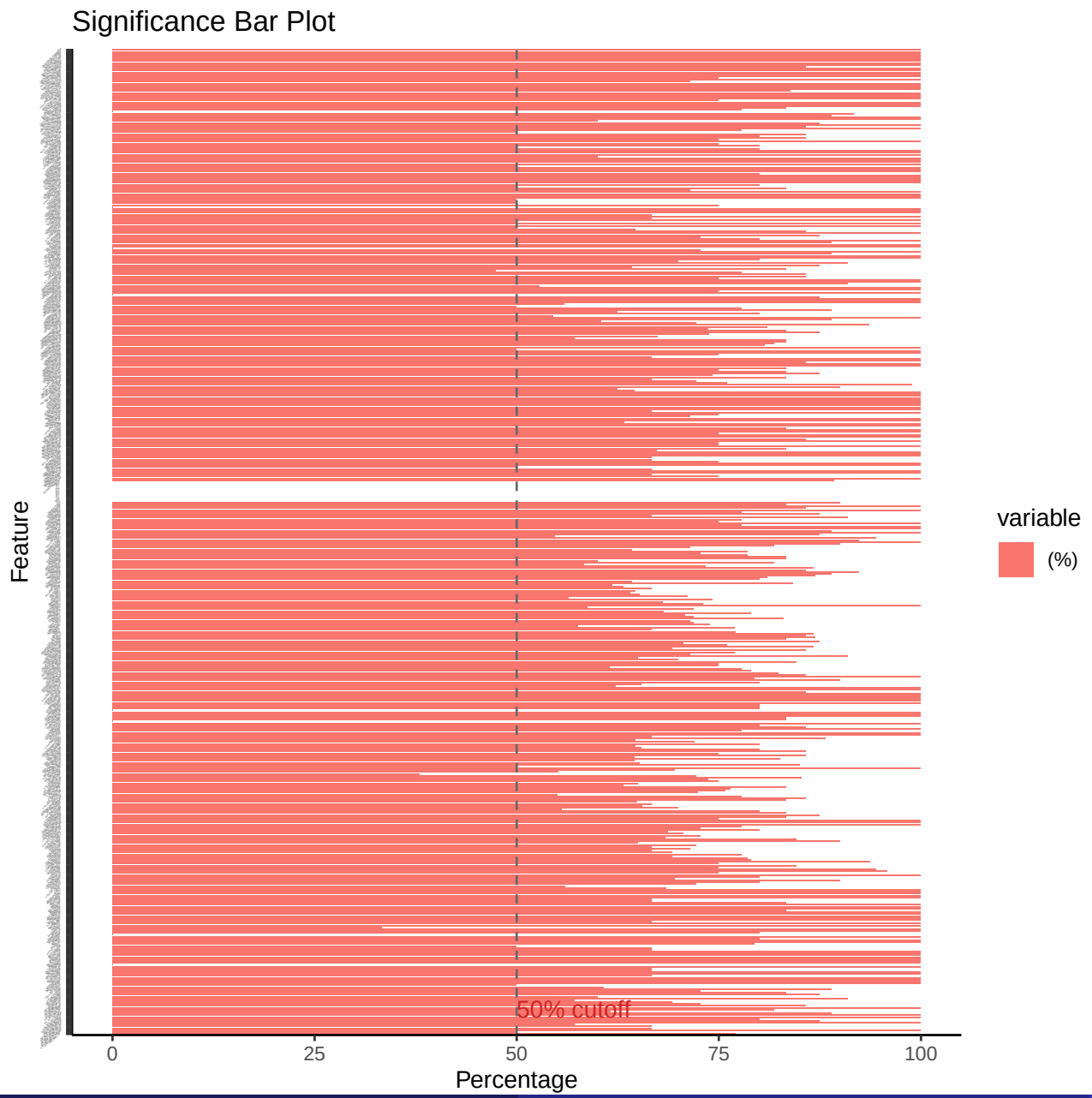
(≈4½ hours required)

```
library(bootStepAIC)
init.model <- lm(V1~1, data) # null model
full.model <- lm(formula.2ord, data) # full model
boot.faic <- boot.stepAIC(init.model, data=data,
                          scope=list(lower=init.model,
                                     upper=full.model),
                          direction="forward", B=100) # Bootstrap samples
sec_ord.features <- names(as.data.frame(model.matrix(formula.2ord, data)[,-1]))
plot_boot_stepAIC_subset <- function(boot_stepAIC_result, component, subset_features) {
    if (!component %in% c("Covariates", "Sign1ficance", "Sign")) {
        stop("Invalid component. Choose from 'Covariates', 'Sign1ficance', or 'Sign'.")
    }
    data_to_plot <- as.data.frame(boot_stepAIC_result[[component]])
    data_to_plot <- data_to_plot[subset_features, , drop = FALSE]
    data_to_plot$Feature <- rownames(data_to_plot)
    data_long <- reshape2::melt(data_to_plot, id.vars = "Feature")

    p <- ggplot(data_long, aes(x = Feature, y = value, fill = variable)) +
        geom_bar(stat = "identity", position = position_dodge()) +
        coord_flip() +
        labs(x = "Feature", y = "Percentage", title = paste(component, "Bar Plot")) +
        theme_classic() +
        theme(axis.text.y = element_text(size = 2.2, angle = 45, hjust = 0.9)) +
        geom_hline(yintercept = 50, linetype = "dashed", color = "dimgrey") +
        annotate("text", x = 0.5, y = 50, label = "50% cutoff",
                 hjust = 0, color = "firebrick3", angle = 0, vjust = -1)

    return(p)
}
```

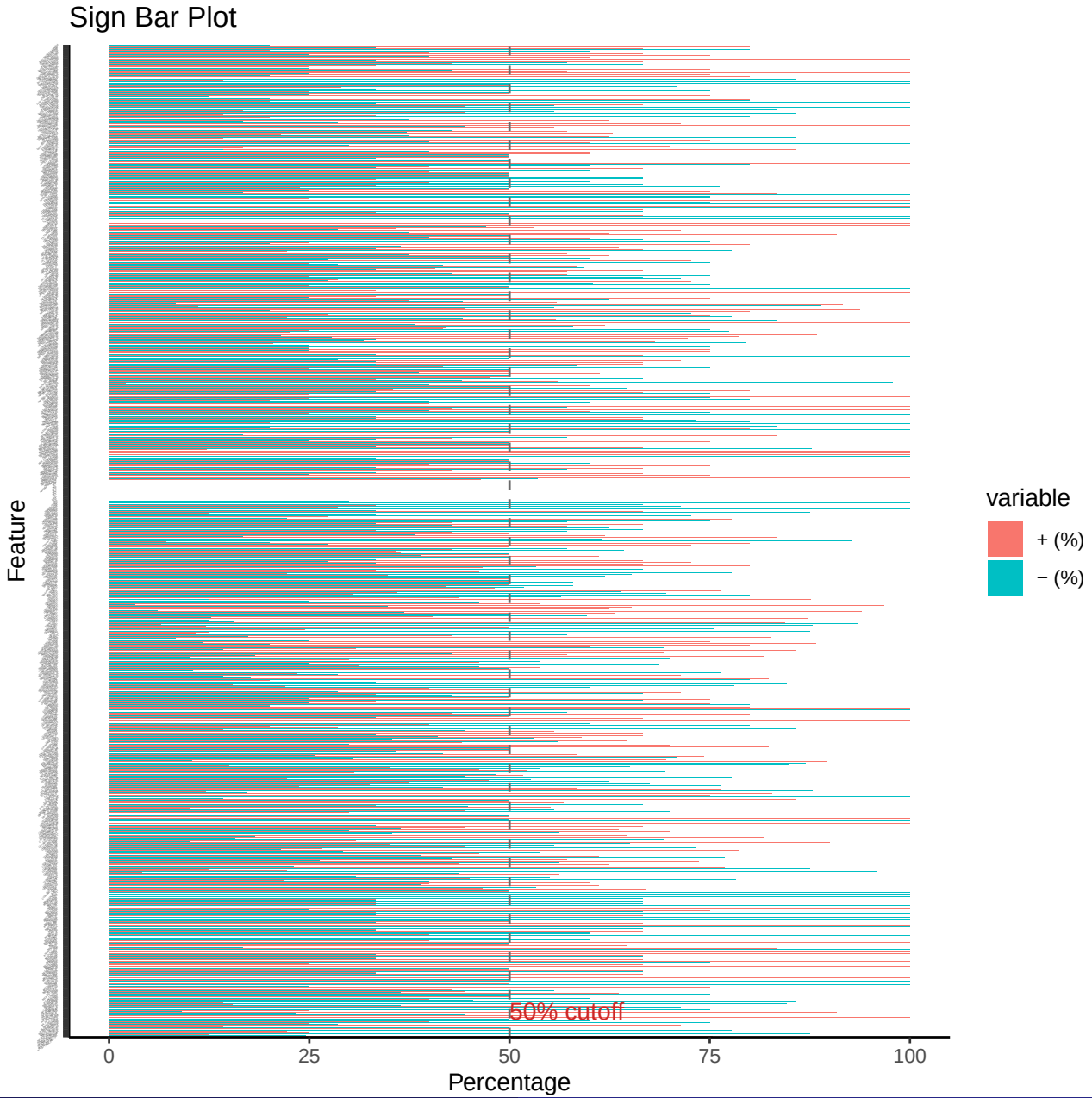
Bootstrapping `StepAIC()`: $Covariates



Bootstrapping `StepAIC()`: $Significance

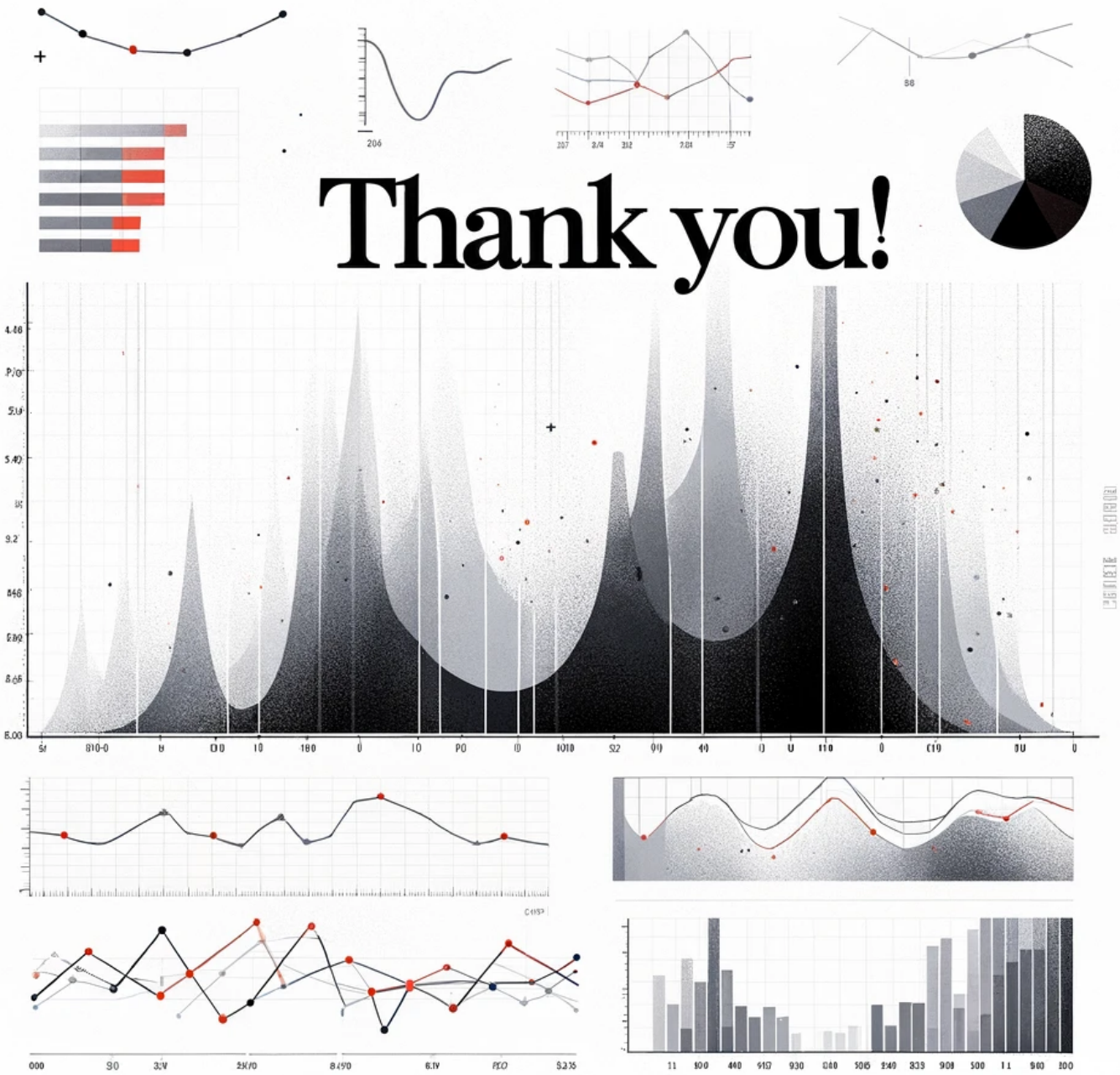


Bootstrapping StepAIC(): $Sign



Most frequent predictors based on 100 bootstrap stepAIC

- **Predictors selected > 50%:** ESS_base, SBPnight_bef, I(SBPnight_bef^2), I(SaO285minbe^2), SaO285minbe, I(AHI_REM_bef^2), SaO280minbe, HoursPweek, Marital, COPD, AHI_bef, BMIdiff, I(SaO280minbe^2), I(BMIdiff^2), I(REM_befpct^2), AHI_REM_bef, TRT_bef, I(HoursPweek^2), I(DBPnight_be^2), SaO2minbef. (20 in total)
- **Predictors selected > 60%:** ESS_base, SBPnight_bef, I(SBPnight_bef^2), I(SaO285minbe^2), SaO285minbe, I(AHI_REM_bef^2), SaO280minbe, HoursPweek, Marital, COPD, AHI_bef, BMIdiff, I(SaO280minbe^2), I(BMIdiff^2), I(REM_befpct^2), AHI_REM_bef. (16 in total)
- **Predictors selected > 70%:** ESS_base, SBPnight_bef, I(SBPnight_bef^2), I(SaO285minbe^2), SaO285minbe, I(AHI_REM_bef^2), SaO280minbe, HoursPweek, Marital, COPD, AHI_bef, BMIdiff, I(SaO280minbe^2). (13 in total)
- **Predictors selected > 80%:** ESS_base, SBPnight_bef, I(SBPnight_bef^2), I(SaO285minbe^2), SaO285minbe, I(AHI_REM_bef^2), SaO280minbe, HoursPweek. (8 in total)


Thank you!