Statistical Computing: Coursework B

James Wright (S1604483)

20 March 2019

Preamble

```
# Importing required libraries and setting seed for replication
source("c:\\Users\\wrigh\\OneDrive\\Desktop\\CWB\\CWB2019code.R")
library(microbenchmark)
library(xtable)
library(ggplot2)
set.seed(2019)
```

Question 1

1.1

Below we construct a function that computes the negated log-likelihood corresponding to the given model in the problem sheet:

$$\begin{aligned} l(N,\theta) = &\log\Gamma(y_1+1) + \log\Gamma(y_2+1) \\ &+ \log\Gamma(N-y_1+1) + \log\Gamma(N-y_2+1) - 2\log\Gamma(N+1) \\ &+ 2N\log(1+e^{\theta}) - (y_1+y_2)\theta \end{aligned}$$

This returns a scalar when $N \geq \max(y_1, y_2)$, or $+\infty$ otherwise:

```r
# negloglike: Computes negative log-likelihood
# Input:
# param : vector with 2 values, N and theta
# Y : vector with 2 values, y1 and y2
# Output :
# a scalar equal to negative log-likelihood or infinity

negloglike <- function(param, Y) {
  N <- param[1]
  theta <- param[2]

  # Computing and returning negative log-likelihood if lower bound exceeded
  # zero otherwise
  if (N >= max(Y)) {
    nll <- sum(lgamma((Y + 1))) +
      sum(lgamma(N - Y + 1)) -
      2 * lgamma(N + 1) +
      2 * N * log(1 + exp(theta)) -
      (sum(Y)) * theta
    return(nll)
  } else {
    return(+Inf)
  }
}
```

To make the code more concise, instead of indexing into the elements of $\vec{Y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$, we took the sum of the resulting computed column vectors as this produces the same result formulaically due to element-wise operations in R, and will do so throughout this document where possible.

1.2

We now use the `optim` function with the negative log-likelihood function constructed above to find the maximum likelihood estimates for N and θ in our model, with N the number of people buried and ϕ the probability of finding a femur. We set our value $\vec{Y} = \begin{bmatrix} 256 \\ 237 \end{bmatrix}$ as on the gravesite 256 left and 237 right femurs were found.

We start the optimization for $N \geq 2\max(y_1, y_2)$ to keep this consistent with 3.1 in order to aid the analysis. We choose an initial value for θ that corresponds to $\phi = 0.5$, as with absence of further information a 50% probability for finding a femur is an obvious initial guess.

```
# Calculating maximum likelihood estimates for N and theta
Y <- c(256, 237)
opt <- optim(
  par = c(2 * max(Y), logit(0.5)), # using probability phi=0.5
  fn = negloglike, Y = Y
)

# Storing obtained MLEs for N, theta
estimates <- opt$`par`
N_hat <- estimates[1]
theta_hat <- estimates[2]
```

We obtain the following maximum likelihood estimates for $\hat{N}$ and $\hat{\theta}$:

```
# Converting MLEs into a data.frame to create a table
display_MLEs <- data.frame(
  N_hat = N_hat,
  Theta_hat = theta_hat
)

# Displaying MLEs as table
print(xtable(display_MLEs,
  caption = "Maximum Likelihood Esimates for Parameters N and Theta"
),
add.to.row = list(
  pos = list(-1),
  command = "\\hline $\\hat{N}$ & $\\hat{\\theta}$\\\\"
),
hline.after = c(0, nrow(display_MLEs)),
include.colnames = FALSE,
include.rownames = FALSE, comment = FALSE
)
```

$\hat{N}$	$\hat{\theta}$
388.13	0.55

Table 1: Maximum Likelihood Esimates for Parameters N and Theta

By change of basis we obtain the following maximum likelihood estimate for $\hat{\phi}$:

```
# Converting estimate of theta to estimate of phi
phi <- ilogit(theta_hat)
# Converting the MLE for phi into a data.frame to create a table
display_phi <- data.frame(
  Phi_hat = phi
```

```
)

# Displaying MLEs as table data as table
print(xtable(display_phi,
  caption = "Maximum Likelihood Estimate for Parameter Phi"
),
add.to.row = list(
  pos = list(-1),
  command = "\\hline $\\hat{\\phi}$\\\\"
),
hline.after = c(0, nrow(display_phi)),
include.colnames = FALSE,
include.rownames = FALSE,
comment = FALSE
)
```

$\hat{\phi}$
0.64

Table 2: Maximum Likelihood Estimate for Parameter Phi

1.3

We now compute the Hessian matrix of the negative log-likelihood function at the mode $(\hat{N}, \hat{\theta})$:

```
# Computing Hessian matrix numerically
opthessian <- optimHess(
  par = estimates,
  fn = negloglike,
  Y = Y
)
print(opthessian)
```

```
##              [,1]       [,2]
## [1,] 0.008988309   1.270193
## [2,] 1.270192513 179.898086
```

We wish to construct a 95% confidence interval for N, assuming normality. We already have our point estimate $\hat{N}$ from our prior computations. By recalling standard theory that through taking the inverse of the Hessian matrix H we obtain the variance-covariance matrix corresponding to our problem parameters, we can deduce that the estimated standard error for $\hat{N}$ is given by the square root of the upper-left element of this matrix. We compute this below:

```
# Constructing 95% confidence interval for N
z_0975 <- qnorm(0.975)
inversehessian <- solve(opthessian) # Inverting Hessian matrix
se <- sqrt(diag(inversehessian)) # Standard error of N,theta MLEs
se_N <- se[1] # Extracting standard error for N
interval <- c(N_hat - z_0975 * se_N, N_hat + z_0975 * se_N)
```

```
# Converting confidence interval into a data.frame to create a table
display_interval <- data.frame(
  Lower = interval[1],
  Upper = interval[2]
)

names(display_interval) <- c("2.5%", "97.5%")

print(xtable(display_interval,
  caption = "Confidence Interval for N"
), type = "latex", comment = FALSE, include.rownames = FALSE)
```

2.5%	97.5%
-50.58	826.85

Table 3: Confidence Interval for N

```
# Displaying lower bound of max(y1,y2) as table
lower_bound <- max(Y)
display_lower_bound <- data.frame(
  MAX = max(Y)
)

names(display_lower_bound) <- c("max(Y)")
print(xtable(display_lower_bound,
  digits = 0,
  caption = "Lower Bound for Model"
), type = "latex", comment = FALSE, include.rownames = FALSE)
```

max(Y)
256

Table 4: Lower Bound for Model

Intuitively, we know at least as many people must have died in the war as the larger of the two numbers of right or left femurs found; that is we know with certainty that $N \in [\max(y_1, y_2), \infty)$. Our 95% confidence interval was computed as $(-50.58, 826.85)$. Observe that this confidence interval does not respect the intuitive lower bound for the model, $\max(y_1, y_2)$ which we have computed to be 256 people, as $256 \in (-50.58, 826.85)$ - there exist values in our confidence interval below this value.

Question 2

2.1

Note that $\Psi(x) = \frac{d \log \Gamma(x)}{dx}$ and $\Psi'(x) = \frac{d\Psi(x)}{dx}$. We first compute the derivatives of our negative log-likelihood function. We have

$$\begin{aligned} l(N, \theta) &= \log(\Gamma(y_1 + 1)) + \log(\Gamma(y_2 + 1)) + \log(\Gamma(N - y_1 + 1)) \\ &\quad + \log(\Gamma(N - y_2 + 1)) - 2\log(\Gamma(N + 1)) + 2N \log(1 + e^{\theta}) - (y_1 + y_2)\theta. \end{aligned}$$

Then, by taking partial derivatives with respect to each parameter we obtain,

$$\frac{\partial l}{\partial \theta} = \frac{2Ne^{\theta}}{1 + e^{\theta}} - (y_1 + y_2),$$

and

$$\frac{\partial l}{\partial N} = \Psi(N - y_1 + 1) + \Psi(N - y_2 + 1) - 2\Psi(N + 1) + 2\log(1 + e^{\theta}).$$

Obtaining the second partial derivatives,

$$\begin{aligned} \frac{\partial^2 l}{\partial N^2} &= \Psi'(N - y_1 + 1) + \Psi'(N - y_2 + 1) - 2\Psi'(N + 1), \\ \frac{\partial^2 l}{\partial N \partial \theta} &= \frac{\partial^2 l}{\partial \theta \partial N} = \frac{2e^{\theta}}{1 + e^{\theta}}, \end{aligned}$$

and

$$\frac{\partial^2 l}{\partial \theta^2} = \frac{2N(1 + e^{\theta})e^{\theta} - 2Ne^{\theta}(e^{\theta})}{(1 + e^{\theta})^2} = \frac{2Ne^{\theta}}{(1 + e^{\theta})^2}.$$

2.2

We now compute the (2×2) Hessian matrix for our negative log-likelihood function by hard-coding its respective elements. We obtain the following matrix:

```
# myhessian: Compute hessian matrix analytically
# Input:
# param : vector with 2 values, N and theta
# Y : vector with 2 values, y1 and y2
# Output :
# a hessian matrix
myhessian <- function(param, Y) {
  N <- param[1]
  theta <- param[2]
  second_partial_theta <- 2 * N * exp(theta) / (1 + exp(theta))^(2) # H_22 entry
  second_partial_N <- sum(psigamma(N - Y + 1, 1)) - 2 * psigamma(N + 1, 1) # H_11 entry
  partial_theta_N <- 2 * exp(theta) / (1 + exp(theta)) # H_12,H_21 entries
  hess <- matrix(c(
    second_partial_N, partial_theta_N,
    partial_theta_N, second_partial_theta
  ),
  nrow = 2,
  ncol = 2
  )
  return(hess)
}

myhess <- myhessian(estimates, Y)
print(myhess)
```

```
##             [,1]       [,2]
## [1,] 0.008988314   1.270193
## [2,] 1.270192559 179.898109
```

For each element, we calculate the relative differences of this Hessian matrix when compared to that produced by `optimHess()` by computing

$$\text{Relative Difference} = \frac{(\text{optHessian} - \text{myhess})}{\text{myhess}}$$

.

```
# Calculating relative difference between elements
relative_difference <- (opthessian - myhess) / myhess

# Formatting relative differences to display as table
display_rd <- data.frame(
  second_partial_N = relative_difference[1, 1],
  partial_theta_N = relative_difference[1, 2], # partial_theta_partial_n
  partial_N_theta = relative_difference[2, 1], # partial_n_partial_theta
  second_partial_theta = relative_difference[2, 2]
)

print(xtable(display_rd,
  digits = (-4),
  caption = "Relative Differences of Elements Between Hessian Matrices"
),
```

```
add.to.row = list(
  pos = list(-1),
  command = "\\hline $\\frac{\\partial^2 l}{\\partial N^2}$
& $\\frac{\\partial^2 l}{\\partial N \\partial \\theta}$
& $\\frac{\\partial^2 l}{\\partial \\theta \\partial N}$
& $\\frac{\\partial^2 l}{\\partial \\theta^2}$\\\\"
),
hline.after = c(0, nrow(display_phi)),
comment = FALSE,
include.colnames = FALSE,
include.rownames = FALSE
)
```

$\frac{\partial^2 l}{\partial N^2}$	$\frac{\partial^2 l}{\partial N \partial \theta}$	$\frac{\partial^2 l}{\partial \theta \partial N}$	$\frac{\partial^2 l}{\partial \theta^2}$
-6.1191E-07	-3.6018E-08	-3.6018E-08	-1.3074E-07

Table 5: Relative Differences of Elements Between Hessian Matrices

We see that the Hessian computed using `myhessian` differs very slightly from that computed using `optimHess`, where the elements of `optimHess` are marginally smaller than `myhessian`. This can be seen by observing that the relative differences for both of the $\frac{\partial^2 l}{\partial N \partial \theta} = \frac{\partial^2 l}{\partial \theta \partial N}$ terms are approximately -3.6018×10^{-8} by symmetry. For $\frac{\partial^2 l}{\partial N^2}$ the relative difference is -6.1191×10^{-7}. For $\frac{\partial^2 l}{\partial \theta^2}$ the relative difference is -1.3074×10^{-7}. These very small differences are to be expected, as while `myhessian` is computing the Hessian matrix analytically and therefore only inaccurate through the machines' round-off error for floating point numbers, `optimhess` is computed using the Nelder-Mead numerical method and so is only producing an approximation to the exact Hessian matrix.

2.3

We now compare the absolute difference of the $\frac{\partial^2 l}{\partial N^2}$ terms for the hard-coded Hessian matrix and `optimHess` with the bound for second order difference approximations given by [1]:

$$\left|\frac{\text{comp}\{f(\text{comp}\{N+2h\})\} - 2\text{comp}\{f(\text{comp}\{N+h\})\} + \text{comp}\{f(N)\}}{h^2} - f''(N)\right| \lesssim \frac{\epsilon_0(4L_0 + 2|N|L_1)}{h^2} + \frac{h^2 L_4}{12}.$$

We compute this explicitly below:

```
# Computing absolute difference between Hessians
abs_diff_L2N <- abs(myhess[1, 1] - opthessian[1, 1])

# Computing relevant approximate bounds
L0 <- abs(negloglike(estimates, Y))
L1 <- abs(sum(digamma(N_hat - Y + 1)) - 2 * digamma(N_hat + 1) +
  2 * log(1 + exp(theta_hat)))
L4 <- abs(sum(psigamma(N_hat - Y + 1, 3)) - 2 * psigamma(N_hat + 1, 3))

# Obtaining machine precision and computing difference approximation h=0.0001
h <- 0.0001
epsilon_0 <- .Machine$double.eps
bound <- epsilon_0 * (4 * L0 + 2 * abs(N_hat) * L1) / (h^2) + h^2 * L4 / 12
```

We obtain the following results:

```
# Displaying absolute difference of Hessian evaluations
display_AbsDiff <- data.frame(
  AbsDiff = abs_diff_L2N
)

names(display_AbsDiff) <- c("Absolute Difference")
print(xtable(display_AbsDiff,
  digits = (-4),
  caption = "Absolute Difference Between Hessian Evaluations"
), type = "latex", comment = FALSE, include.rownames = FALSE)
```

Absolute Difference
5.5000E-09

Table 6: Absolute Difference Between Hessian Evaluations

```
# Displaying lower bound of second order difference approximation as table
display_boundSO <- data.frame(
  Bound = bound
)

names(display_boundSO) <- c("Error Bound")
print(xtable(display_boundSO,
  digits = (-4),
  caption = "Bound for Second Partial Derivative with Respect to N"
), type = "latex", comment = FALSE, include.rownames = FALSE)
```

```
# Calculating minimization h and displaying as table
h_min <- (24*.Machine$double.eps*(2*L0+abs(N_hat)*L1)/L4)^(1/4)
display_hmin <- data.frame(
  h_min = h_min
```

Error Bound
6.5206E-07

Table 7: Bound for Second Partial Derivative with Respect to N

```
)
print(xtable(display_hmin,
  digits = (6),
  caption = "Approximate h to Minimise Bound"
),
add.to.row = list(
  pos = list(-1),
  command = "\\hline Minimisation $h$\\\\"
),
hline.after = c(0, nrow(display_hmin)),
comment = FALSE,
include.colnames = FALSE,
include.rownames = FALSE
)
```

Minimisation h
0.015480

Table 8: Approximate h to Minimise Bound

Observe that the absolute difference between the two Hessian evaluations is smaller than that of the second order difference approximation error bound with values of 5.5×10^{-9} and 6.5×10^{-7} respectively. It is worth noting that the second order difference approximation bound is minimised by[1] $h \approx \left(24\epsilon_0 \frac{2L_0+|\hat{N}|L_1}{L_3}\right)^{\left(\frac{1}{4}\right)}$, which we see in Table 8 is approximately the value of $h \approx 0.015$. This means that the resulting value of $\frac{\partial^2 l}{\partial N^2}$ produced my `optimHess` is numerically very similar to the analytic value; but it is in fact possible to improve this difference to a smaller value by taking a larger value of h in `optimHess`.

2.4

We now compute the computational costs of these two functions. In order to reduce the variation in these measurements we use 100,000 tests.

```
# Calculating computational costs of numerical hessians
mb <- microbenchmark::microbenchmark(
  "myhessian" = myhessian(estimates, Y),
  "optimHess" = optimHess(
    par = estimates,
    fn = negloglike, Y = Y
  ), times = 100000
)

display_mb <- summary(mb)
```

```
# Plotting graph of computational costs
autoplot(mb) +
  ggtitle("Figure 1: Computational Costs of optimHess and myHessian") +
  theme(plot.title = element_text(hjust = 0.5))
```


```
# Displaying microbenchmark data as table
print(xtable(display_mb,
  caption = "Computational Costs of the Hessian Calculations [microseconds]"
), type = "latex", comment = FALSE)
```

In Figure 1 we can immediately see that the average computational cost of `myhessian` is much smaller than `optimHess`. We see by comparing mean and median running times that `myhessian` is on average

	expr	min	lq	mean	median	uq	max	neval
1	myhessian	4.40	5.30	6.14	5.70	6.00	3149.60	100000.00
2	optimHess	43.20	49.80	56.48	51.40	53.50	64997.20	100000.00

Table 9: Computational Costs of the Hessian Calculations [microseconds]

approximately 9 and 10 times faster respectively than `optimHess`. By considering the upper and lower quartiles for each function, we also observe that the variability of times is typically wider for `optimHess` than `myhessian`. These results are to be expected as `myhessian` computes the Hessian directly using hard-coded formulae meaning R only needs to perform simple arithmetic which is generally computationally inexpensive, and much more consistent in regards to how many operations are required to approach the answer. Meanwhile, `optimHess` is implicitly performing the Nelder-Mead optimization method which is more computationally expensive for the machine, and the number of operations to approach the answer will depend on the inputs typically causing more variation in running time.

With the above said, it is worth noting that the `myhessian` benchmark does not take into consideration the computational cost for the human when calculating required derivatives to give to the machine - which took far longer than the microseconds difference seen between these algorithms. With that in mind we cannot conclude that using `myhessian` is "better" than `optimHess` in general as this would depend on the context it is being used - the microseconds gained in a single use would not be worth the human labour to implement - while if we were continuously processing large amounts of data this may prove worthwhile over time.

Question 3

3.1

Below we create an optimisation function that computes the required MLEs, for use in our J parametric bootstrap function that starts the optimisation at $N = 2\max\{y_1^{(j)}, y_2^{(j)}\}$, and $\theta = 0$.

```
# opti: Computes MLEs
# Input:
# Y : vector with 2 values, y1 and y2
# Output :
# vector of maximum likelihood estimators
opti <- function(Y) {
  MLE <- optim(
    par = c(2 * max(Y), 0),
    fn = negloglike,
    Y = Y
  )$par
  return(MLE)
}
```

We now create a J-parametric bootstrap function, to produce samples of the parameter estimates seen in question 1.

```
# arch_boot: performs J-parametric bootstrap algorithm
# Input:
# param : vector with 2 values, N and theta
# J: a positive integer
# Output :
# a matrix of size J-by-2 of  bootstrap estimates
arch_boot <- function(param, J) {
  N <- floor(param[1]) # Turning N into an integer
  n <- length(param) # Number of parameters for dimension of matrix
  phi_hat <- ilogit(param[2])
  A <- rbinom(2 * J, N, phi_hat) # Generating 2 pairs of data for left and right femurs
  samples <- matrix(A, J, n) # Storing generated values in J-by-2 matrix
  matrix <- t(apply(samples, 1, opti)) # Compute MLEs for every row in matrix
  return(matrix)
}
```

3.2

Using our J parametric bootstrap function, we produce $J = 10,000$ bootstrap samples of the parameter estimates for N and θ.

```
# Compute bootstrap estimates
J <- 10000
BS_mles <- arch_boot(estimates, J)
```

From this, we calculate estimates of the bias and standard deviation of the estimators of N.

```
# Computing average bias and standard deviation for N estimators
bias_N <- mean(BS_mles[, 1] - N_hat)
sd_N_hat <- sd(BS_mles[, 1] - N_hat)

# Displaying bias and standard deviation for N estimator as table
display_bias_N <- data.frame(
  Bias_N = bias_N,
  SD_N = sd_N_hat
)

names(display_bias_N) <- c("Estimator Bias: N", "Estimator SD: N")

print(xtable(display_bias_N,
  caption = "Estimate of Bias and Standard Deviation for N Estimator"
), type = "latex", comment = FALSE, include.rownames = FALSE)
```

Estimator Bias: N	Estimator SD: N
87.94	1151.35

Table 10: Estimate of Bias and Standard Deviation for N Estimator

Recall that the Maximum Likelihood Estimator for N from 1.2 was $\hat{N} = 388.13$. With this in mind, we see that there is a relatively large bias in the bootstrapped samples of our parameter estimates with a value of 87.94. We also see that there is a standard deviation of 1151.35, which is extremely large when compared to $\hat{N}$. These numbers imply that our estimate procedure is generally both inaccurate and not robust due to high variability; in particular on average these MLEs overestimate the parameter N by around 88 people, and the estimates are dominated by the variability.

We also calculate the bias and standard deviation of the estimators of θ and display them below.

```
# Computing average bias and standard deviation for theta estimators
bias_theta <- mean(BS_mles[, 2] - theta_hat)
sd_theta_hat <- sd(BS_mles[, 2] - theta_hat)

# Displaying bias and standard deviation for theta estimators as table
display_bias_theta <- data.frame(
  Bias_Theta = bias_theta,
  SD_Theta = sd_theta_hat
)

names(display_bias_theta) <- c("Estimator Bias: Theta", "Estimator SD: Theta")

print(xtable(display_bias_theta,
  caption = "Estimate of Bias and Standard Deviation for Theta Estimator"
), type = "latex", comment = FALSE, include.rownames = FALSE)
```

Estimator Bias: Theta	Estimator SD: Theta
2.43	3.84

Table 11: Estimate of Bias and Standard Deviation for Theta Estimator

3.3

We now compute the corresponding bootstrap confidence intervals for N and ϕ.

We assume the the bootstrap principle holds for the $\log(N)$ and $\theta = \text{logit}(\phi)$ scales. That is, the errors of the bootstrapped estimates of some parameter ρ_{true} follow the same distribution as the error for the estimator $\hat{\rho}$, so we have

$$\mathbb{E}(\hat{\rho} - \rho_{\text{true}}) = \mathbb{E}(\hat{\rho}^{(j)} - \hat{\rho})$$

and

$$\text{Var}(\hat{\rho} - \rho_{\text{true}}) = \text{Var}(\hat{\rho}^{(j)} - \hat{\rho}).$$

Then to find a 95% confidence interval for $\log(N)$ we require α, β such that

$$\mathbb{P}(\alpha < \log(\hat{N}^{(j)}) - \log(\hat{N}) < \beta) = 0.95.$$

So by the bootstrap principle,

$$\mathbb{P}(\alpha < \log(\hat{N}) - \log(N_{\text{true}}) < \beta) = 0.95.$$

After rearrangement it follows that to obtain a confidence interval for $\log(N)$

$$\mathbb{P}(\log(\hat{N}) - \beta < \log(N_{\text{true}}) < \log(\hat{N}) - \alpha) = 0.95 \implies CI_{\text{boot}} = (\log(\hat{N}) - \beta, \log(\hat{N}) - \alpha).$$

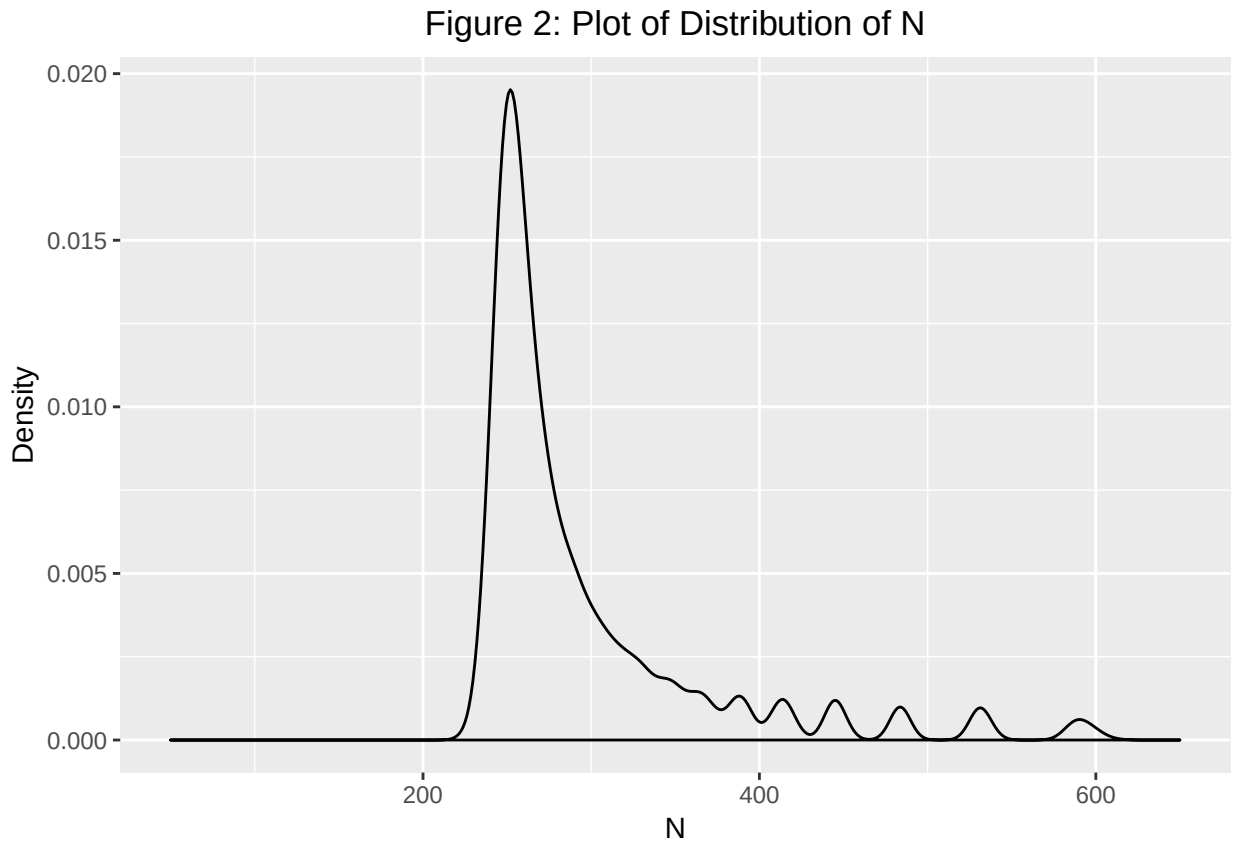
We implement this below, as well as plotting the density function of our estimates. After change of basis we obtain the following confidence interval for N:

```
# Calculating confidence interval for log N assuming bootstrap principle
logdiff_BS_mles <- log(BS_mles[, 1]) - log(N_hat)
quantile_N <- quantile(logdiff_BS_mles, probs = c(0.975, 0.025))

CI_log_N <- log(N_hat) - quantile_N
CI_N <- exp(CI_log_N)

# Creating density plot of MLE N bootstrapped data

data_N_MLE <- data.frame(
  BS_mle_N = BS_mles[, 1]
)
ggplot(data = data_N_MLE, aes(x = BS_mle_N)) +
  geom_density() +
  xlim(50, 650) +
  ggtitle("Figure 2: Plot of Distribution of N") +
  xlab("N") + ylab("Density") + theme(plot.title = element_text(hjust = 0.5))
```




```
# Converting confidence interval for N into a data.frame to create a table
display_CI_N <- data.frame(
  Lower = CI_N[1],
  Upper = CI_N[2]
)

names(display_CI_N) <- c("2.5%", "97.5%")

print(xtable(display_CI_N,
  caption = "Confidence Interval for N"
), type = "latex", comment = FALSE, include.rownames = FALSE)
```

2.5%	97.5%
62.12	630.32

Table 12: Confidence Interval for N

We see that $\hat{N}$ in this case has a narrower confidence interval than that seen in 1.3; with values of $(62.12, 630.32)$ vs $(-50.58, 826.85)$ respectively. This is to be expected, as here we did not need to assume the distribution of N - which as by evidenced Figure 2 is in fact skewed. By the Bootstrap principle, we have that the errors of our bootstrapped estimates have approximately the same distribution as the error of $\hat{N}$. Hence, by using an empirical distribution for the bootstrapped data, we were able to construct a tighter 95% confidence interval than the one constructed assuming asymptotic normality as it will better reflect the true distribution.

By an analogous argument to above, to find a 95% confidence interval for θ we require α, β such that

$$\mathbb{P}(\alpha < \hat{\theta} - \hat{\theta}_{\text{true}} < \beta) = 0.95 \implies CI_{\text{boot}} = (\hat{\theta} - \beta, \hat{\theta} - \alpha).$$

We implement this below, which after a change of basis gives us the following confidence interval for ϕ:

```
# Calculating confidence interval for theta assuming bootstrap principle
diff_theta <- BS_mles[, 2] - theta_hat
quantile_theta <- quantile(diff_theta, probs = c(0.975, 0.025))
CI_theta <- theta_hat - quantile_theta

# Calculating confidence interval for phi
CI_phi <- ilogit(CI_theta)

# Converting confidence interval for phi into a data.frame to create a table
display_CI_phi <- data.frame(
  Lower = CI_phi[1],
  Upper = CI_phi[2]
)

names(display_CI_phi) <- c("2.5%", "97.5%")

print(xtable(display_CI_phi,
  digits = (-6),
  caption = "Confidence Interval for Phi"
), type = "latex", comment = FALSE, include.rownames = FALSE)
```

2.5%	97.5%
4.404987E-07	9.633331E-01

Table 13: Confidence Interval for Phi

Question 4

4.1

We now discuss how the purpose of each step in the procedure `cwb4_scores`. Note that this procedure takes a list of data and a value K that is the desired size of fold used in cross-validation. Consider the following steps:

Step 1

The `cwb4_scores` function first accesses the `cvk_define_splits` function and stores its output in the variable `the_splits`, by inputting the number of rows in the list data and size of fold K. *The purpose of this step is to randomly assign indexes* 1 *to* K *that will define a random splitting into* K *subsets of data* - it does this in the following way:

1. It first checks that K-fold cross-validation makes sense for the data by testing if the number of rows in the data set exceeds the desired number of folds. If not, it stops the procedure gives an error message.
2. If the number of rows exceeds the number of desired folds, it calculates how much data should be allocated to each subset so that they are as close to equal size as possible. This is done using the formula:
$$N_k = \left\lceil \frac{kN}{K} \right\rceil - \left\lceil \frac{(k-1)N}{K} \right\rceil$$
with $k \in \{1, ..., K\}$.
3. It then produces a vector of length N by randomly sampling from a sequence of numbers $1, ..., K$ with replacement, and returns this.

Step 2

The `cwb4_scores` function now accesses the `cvk_do_all` function and stores its output in the variable `scores_cvk`, by inputting the training data column from the data list, the index vector output by `cvk_define_splits` and the function `makeformula(2)` which will generate the linear model formula for longitude, latitude, elevation, and temporal trend - appended with a Fourier series of order 2. *The purpose of this step is to train* K *models, performing* K*-fold cross-validation and return a data frame of Standard Error, Dawid-Sebastiani, and Brier scores on each row corresponding to* K *bootstrapped linear models in order to assess the performance of the models that are trained* - it does this in the following way:

1. It first initialises a data frame of zeros of the maximum length of our index vector to store the Standard Error, Dawid-Sebastiani and Brier scores for us to later loop into.
2. It then loops through values $k \in \{1, ..., K\}$ corresponding to the indexes of our data set. This essentially trains K separate linear models using bootstrapped data sets and then computes SE, DS and Brier scores for each of them. In particular:
   (a) Within this loop, we first split the data into training and validation data based on our created indexes - whereby the data indexed k is allocated to the validation set and the rest of the data is allocated to the training set.
   (b) The set of training data is then used to fit model parameters corresponding to the input linear model formula, and then the validation data is used to make predictions using this model.
   (c) The scores are then computed corresponding to the predictions of this data, an average of them is taken, and they looped into the respective row of the initialised data frame corresponding to the model k.
3. We finish with a data frame with the means of SE, DS and Brier scores for each of the K models, which is returned.

Step 3

The `cwb4_scores` function now trains a model using the training data column of the data set "as a whole", and then makes predictions using the test data column of the data set "as a whole" (i.e. 1-fold cross-validation). It then computes the mean SE, DS and Brier scores corresponding to these predictions and takes the mean - this data frame of values is stored in the variable `scores_test`.

Step 4

The `cwb4_scores` function now creates a list that stores the mean of all of the scores calculated via cross-validation in step 2 as `cvk_mean`, and it stores the standard error of those scores as `cvk_std_error`. It also stores the mean of the scores computed in the 1-fold cross-validated model seen in step 3 as `test`.

4.2

Using these functions, we compute $J = 20$ bootstrap samples of these scores with $K = 10$ folds.

```
# Reading in dataset
data <- read_TMIN_data()

# Creating 20 samples of scores
J <- 20
SE <- c(1:J)
DS <- c(1:J)
Brier <- c(1:J)

# Creating data.frame to store bootstrapped scores (training)
S_boot_train <- data.frame(
  SE = numeric(J),
  DS = numeric(J),
  Brier = numeric(J)
)

# Creating data.frame to store bootstrapped scores (testing)
S_boot_test <- data.frame(
  SE = numeric(J),
  DS = numeric(J),
  Brier = numeric(J)
)

# Creating 20 bootstrap samples and cross-validate K times looping scores into list
K <- 10
for (j in 1:J) {
  data_resample <- data_list_resample(data)
  cwb4 <- cwb4_scores(data_resample, K)
  S_boot_train[j, ] <- cwb4$cvk_mean
  S_boot_test[j, ] <- cwb4$test
}
```

We obtain the following heads of the bootstrapped training and test sample data:

```
# Displaying head of bootstrapped samples as table

display_S_boot_train <- head(S_boot_train)
display_S_boot_test <- head(S_boot_test)

print(xtable(display_S_boot_train,
  digits = (6),
  caption = "Head of Training Bootstrap Score Samples"
), type = "latex", comment = FALSE, include.rownames = TRUE)
```

```
print(xtable(display_S_boot_test,
  digits = (6),
  caption = "Head of Test Bootstrap Score Samples"
), type = "latex", comment = FALSE, include.rownames = TRUE)
```

	SE	DS	Brier
1	10.073090	3.309895	0.117655
2	10.060936	3.308664	0.116998
3	10.041132	3.306710	0.117659
4	9.983383	3.300941	0.116900
5	10.116196	3.314151	0.117251
6	10.030630	3.305689	0.116100

Table 14: Head of Training Bootstrap Score Samples

	SE	DS	Brier
1	9.992624	3.301873	0.117837
2	10.125747	3.315096	0.118093
3	9.975541	3.300151	0.116024
4	10.170013	3.319612	0.116041
5	10.083247	3.310874	0.118655
6	10.036580	3.306230	0.117948

Table 15: Head of Test Bootstrap Score Samples

4.3

Below we compute the sample means and standard deviations for three score types Standard Error, Dawid-Sebastiani, and Brier. We consider each of the training and test data sets separately.

```
# Computing means and standard deviations of scores for training data
train_sample_means_scores <- colMeans(S_boot_train)
train_sample_sd <- apply(S_boot_train, 2, sd)

# Computing means and standard devations of scores for test data
test_sample_means_scores <- colMeans(S_boot_test)
test_sample_sd <- apply(S_boot_test, 2, sd)
```

We obtain the following results:

```
# Putting training data mean and sd scores in data.frame to create table
display_train_scores <- data.frame(
  Mean = train_sample_means_scores,
  SD = train_sample_sd
)

names(display_train_scores) <- c("Mean", "Standard Deviation")

print(xtable(display_train_scores,
  digits = (6),
  caption = "Sample Means and Standard Deviations for Score Types - Training Data"
), type = "latex", comment = FALSE, include.rownames = TRUE)
```

	Mean	Standard Deviation
SE	10.023158	0.051745
DS	3.304903	0.005166
Brier	0.116842	0.000523

Table 16: Sample Means and Standard Deviations for Score Types - Training Data

```r
# Putting test data mean and sd scores in data.frame to create table
display_test_scores <- data.frame(
  Mean = test_sample_means_scores,
  SD = test_sample_sd
)

names(display_test_scores) <- c("Mean", "Standard Deviation")

print(xtable(display_test_scores,
  digits = (6),
  caption = "Sample Means and Standard Deviations for Score Types - Test Data"
), type = "latex", comment = FALSE, include.rownames = TRUE)
```

	Mean	Standard Deviation
SE	10.035137	0.099085
DS	3.306095	0.009881
Brier	0.116549	0.001394

Table 17: Sample Means and Standard Deviations for Score Types - Test Data

4.4

We now estimate the bias $\mathbb{E}[\hat{S_0}^{\text{train}(K)} - S^{\text{test}}]$ and standard deviation of $\hat{S_0}^{\text{train}(K)}$ assuming bootstrap principle holds.

```
# Computing bias and standard deviation of the scores using bootstrap principle
bias_score <- colMeans(S_boot_train - S_boot_test)
standard_devation_score <- apply(S_boot_train - S_boot_test, 2, sd)

# Displaying bias and standard deviations as a table
display_biases <- data.frame(
  score = bias_score,
  sd = standard_devation_score
)

print(xtable(display_biases,
  digits = (6),
  caption = "Bias and Standard Deviation Estimates"
),
add.to.row = list(
  pos = list(-1),
  command = "\\hline Score & Bias & Standard Deviation\\\\"
),
hline.after = c(0, nrow(display_biases)),
include.colnames = FALSE,
comment = FALSE
)
```

Score	Bias	Standard Deviation
SE	-0.011979	0.106992
DS	-0.001192	0.010662
Brier	0.000292	0.001404

Table 18: Bias and Standard Deviation Estimates

By Table 18 we see that the cross-validated score differences are the most biased for the SE score, followed by DS and then Brier, differing by approximately a factor of 10 for each. For SE and DS, the cross-validated the bias is on average below the scores for the test data. For Brier, the scores are biased above that for the test data, however this value is so small it is likely statistically insignificant. We see that the cross-validated scores also have the most variance for SE, followed by DS and then Brier. These results tell us that the cross-validation procedure is both quite accurate and robust.

We would expect that the cross-validated training data scores for the 10-fold cross-validation data set to exceed the test scores, as the individual models are trained on smaller data sets, giving them a worse fit for the data in theory. However, by looking at Tables 16 and 17 we see that there is not a significant difference between the scores in this case - and in this case the cross-validated scores for SE and DS are slightly below the test scores - this is potentially due to the fact that even though each model uses less during the 10-fold cross-validation, due to our use of bootstrap and sheer large amount of data to begin with, we still have enough data training the model for this to have an insignificant effect. By Table 16 and 17 we also see that the test scores have greater standard deviation than the training scores. This is unsurprising as the cross-validated scores from the training data are computed using multiple models which generally should average to a lower standard deviation than that for a single model.

We also observe that the variability has a bigger impact on the score estimates, as the standard deviations around a factor of 10 higher than the biases across all scores. However, standard deviation is still a factor of 1000 smaller than the average estimates seen in Tables 16 and 17, so neither the bias nor standard deviation

dominate the estimates.

References

[1] Lecture 5.
Finn Lindgren. *Statistical Computing - Lecture 5.*
The University of Edinburgh, 2019.