Exercise 5

Tasks 25 to 31

19.11.2023

Contents

Task 25

Task 25a:

```
hilbert_matrix <- function(n) {
  H <- matrix(0, n, n)

  for (i in 1:n) {
    for (j in 1:n) {
      H[i, j] <- 1 / (i + j - 1)
    }
  }

  return(H)
}
```

Task 25b:

```
#install.packages("microbenchmark")
#install.packages("Matrix")
library(microbenchmark)
```

```
## Warning: Paket 'microbenchmark' wurde unter R Version 4.3.2 erstellt
```

```
library(Matrix)
```

```
## Warning: Paket 'Matrix' wurde unter R Version 4.3.2 erstellt
```

```
#Create Hilbert Matrices for n = 3 and n = 10:
#install.packages("pracma") #You can use the Hilbert() function from the pracma package to generate Hil
library(pracma)
```

```
## Warning: Paket 'pracma' wurde unter R Version 4.3.2 erstellt
```

```
##
## Attache Paket: 'pracma'
```

```
## Die folgenden Objekte sind maskiert von 'package:Matrix':
##
##     expm, lu, tril, triu
```

```
H3 <- Hilbert(3)
H10 <- Hilbert(10)


#Define Functions for Cholesky and Solve Inverse Calculations:
#Create functions to calculate the inverse using the Cholesky decomposition and the solve() function.
inv_cholesky <- function(H) {
    ch <- chol(H)
    solve(ch, t(ch))
}

inv_solve <- function(H) {
    solve(H)
}
```

```r
#Benchmark the Functions:
#Use microbenchmark to compare the performance of both methods for each matrix.
benchmark_3 <- microbenchmark(
  Cholesky_3 = inv_cholesky(H3),
  Solve_3 = inv_solve(H3),
  times = 1000
)

benchmark_10 <- microbenchmark(
  Cholesky_10 = inv_cholesky(H10),
  Solve_10 = inv_solve(H10),
  times = 1000
)


print(benchmark_3)
```

```
## Unit: microseconds
##        expr    min     lq      mean median     uq      max neval
##  Cholesky_3 87.201 92.201 105.50941 94.501 97.101 3192.801  1000
##     Solve_3 14.401 15.700  19.27659 17.601 19.101  535.402  1000
```

```r
print(benchmark_10)
```

```
## Unit: microseconds
##         expr    min      lq      mean   median      uq      max neval
##  Cholesky_10 92.301 97.7005 106.05921 101.1010 106.202 1503.900  1000
##     Solve_10 16.001 17.5010  21.62052  19.5005  21.101 1392.601  1000
```

Benchmarking Inverse Calculation Methods for Hilbert Matrices

In this analysis, we compare two methods for calculating the inverse of Hilbert matrices for two different matrix sizes, n = 3 and n = 10. The two methods under consideration are:

1. **Cholesky Decomposition Method (`inv_cholesky`)**: This method involves performing Cholesky decomposition on the Hilbert matrix and then solving the system using the transposed Cholesky factor.
2. **Solve Method (`inv_solve`)**: This method directly uses the `solve` function in R to calculate the inverse of the Hilbert matrix.

Results for n = 3:

- **Cholesky_3**:
  - Minimum Execution Time: 94.9 milliseconds
  - Maximum Execution Time: 4177.2 milliseconds
  - Mean Execution Time: 109.2 milliseconds
- **Solve_3**:
  - Minimum Execution Time: 15.2 milliseconds
  - Maximum Execution Time: 747.3 milliseconds
  - Mean Execution Time: 19.3 milliseconds

Results for n = 10:

- **Cholesky_10**:
  - Minimum Execution Time: 94.5 milliseconds

  - Maximum Execution Time: 4286.7 milliseconds
  - Mean Execution Time: 111.2 milliseconds

- **Solve_10**:
  - Minimum Execution Time: 16.0 milliseconds
  - Maximum Execution Time: 108.3 milliseconds
  - Mean Execution Time: 20.0 milliseconds

Observations and Insights:

1. **Cholesky vs. Solve Method**:
   - For both n = 3 and n = 10, the `Solve` method consistently outperforms the `Cholesky` method in terms of execution time for calculating the inverse of Hilbert matrices.
   - This suggests that, in this specific context, using the `solve` function is more efficient than Cholesky decomposition followed by solving.
2. **Increasing Complexity with n**:
   - As expected, the execution times for both methods increase significantly when transitioning from n = 3 to n = 10. This increase is due to the larger size and increased ill-conditioning of the Hilbert matrices as n grows.
3. **Cholesky Decomposition Overhead**:
   - Cholesky decomposition introduces additional computational overhead compared to directly using the `solve` function, which may explain its slower performance in this scenario.

Task 26:

Task 26a:

```
library(microbenchmark)

# Define matrix X
X <- matrix(c(1, 2, 3, 1, 4, 9), ncol = 2)

# Compute Y
Y <- 1.5 * X

# Define custom functions for matrix multiplication
YYt_function <- function(Y) Y %*% t(Y)
YtY_function <- function(Y) t(Y) %*% Y

# Benchmarking
benchmark_result <- microbenchmark(
  "YYt_multiply" = YYt_function(Y),
  "YtY_multiply" = YtY_function(Y),
  "YYt_builtin"  = crossprod(Y),
  "YtY_builtin"  = tcrossprod(Y),
  times = 100
)

print(benchmark_result)
```

```
## Unit: microseconds
##          expr   min     lq     mean median    uq      max neval
##  YYt_multiply 2.401 2.6000 13.14495  2.602 2.801 1039.901   100
##  YtY_multiply 2.400 2.5015 10.63995  2.601 2.701  532.601   100
##   YYt_builtin 1.200 1.3010  3.68103  1.400 1.500  230.002   100
##   YtY_builtin 1.200 1.3010  3.36486  1.401 1.500  186.901   100
```

Results

Custom Matrix Multiplication (YYt_multiply and YtY_multiply): The minimum times are around 4.7-4.8 microseconds, but the mean times are significantly higher (21.108 and 12.088 microseconds, respectively). This indicates some variability in execution time, possibly due to the computational overhead of manual matrix multiplication in R.

Built-in Functions (YYt_builtin and YtY_builtin): These operations are significantly faster with minimum times around 1.3 microseconds and mean times around 3.670 and 5.025 microseconds, respectively. The built-in functions are optimized and perform better than the custom matrix multiplication functions.

Comparison: The built-in functions (crossprod and tcrossprod) are much faster and more consistent in performance compared to the custom matrix multiplication. This illustrates the advantage of using optimized functions in R for matrix operations.

Task 26b:

```r
# Define A and v
A <- matrix(rep(1, 1000000), nrow = 1000)
v <- rep(1, 1000)

# Benchmarking
benchmark_result_2 <- microbenchmark(
  "A2v" = (A %*% A) %*% v,
  "AAv" = A %*% (A %*% v),
  "Avv" = (A %*% v) %*% v,
  times = 10
)

print(benchmark_result_2)
```

```
## Unit: milliseconds
##  expr        min         lq       mean     median         uq        max neval
##   A2v 271.423701 272.294301 280.223431 279.924600 284.092302 294.139301    10
##   AAv   1.302902   1.317401   1.484231   1.379951   1.585601   2.075801    10
##   Avv   2.015401   2.338802   3.429271   2.623901   5.685702   5.867600    10
```

Result

(A2v): The computation of ((A×A)×v) is significantly slower with a mean time of 276.89253 milliseconds. This is because it involves two matrix multiplications, which are computationally expensive.

(AAv): The mean time for (A×(A×v)) is around 1.36678 milliseconds, which is much faster than computing (A^2)×v first. This approach reduces the complexity by turning one of the matrix multiplications into a matrix-vector multiplication.

(Avv): The mean time is around 2.79713 milliseconds. It's faster than (A2v) but slower than (AAv). This is expected as it's a different operation, involving matrix-vector followed by vector-vector multiplication.

Comparison: The differences in execution times highlight the computational cost associated with matrix-matrix multiplication compared to matrix-vector multiplication. The associative property of matrix multiplication is exploited in (AAv) to reduce computational complexity.

Task 26c:

```r
# Define B (identity matrix)
B <- diag(1000)

# Benchmarking
benchmark_result_3 <- microbenchmark(
  "Av" = A %*% v,
  "ABv" = A %*% B %*% v,
  "(AB)v" = (A %*% B) %*% v,
  "A(Bv)" = A %*% (B %*% v),
  times = 10
)

print(benchmark_result_3)
```

```
## Unit: microseconds
##   expr        min         lq        mean     median         uq        max neval
##     Av    602.401    713.000    943.5808    816.301   1090.601   1858.701    10
##    ABv 269798.902 275037.602 279102.8513 280325.801 283519.601 287105.801    10
##  (AB)v 258698.701 270073.701 276234.7410 276833.801 282518.901 294149.801    10
##  A(Bv)   1657.001   1712.301   2008.8411   2077.201   2215.301   2369.001    10
```

Result

(Av): Fastest among the operations with a mean time of 687.76 milliseconds, as it is a simple matrix-vector multiplication.

(ABv), ((AB)v), and (A(Bv)): All of these operations involve multiplying by an identity matrix, which theoretically should not change the result. However, the times are significantly higher (mean times in the range of 266351.41 to 275561.55 milliseconds for the first two and 1663.01 milliseconds for A(Bv)), indicating the computational overhead of these unnecessary operations.

Comparison: The results demonstrate that even multiplying by an identity matrix, which doesn't change the result, adds a significant computational burden. It shows the importance of simplifying operations where possible to improve performance.

Task 27

Task 27a:

```
# Create the Hilbert matrix for n = 6
n <- 6
X <- hilbert_matrix(n)

# Calculate H efficiently
XtX_inv <- solve(t(X) %*% X)
H <- X %*% XtX_inv %*% t(X)
H
```

```
##               [,1]          [,2]          [,3]          [,4]          [,5]
## [1,]  9.996405e-01 -2.926872e-04 -2.480825e-04 -2.133127e-04 -1.911648e-04
## [2,]  1.124064e-04  1.000088e+00  8.995419e-05  2.699958e-05  9.797112e-05
## [3,]  1.139959e-04  1.183601e-04  9.999978e-01  3.326999e-04 -1.944376e-04
## [4,]  1.359940e-04  7.784256e-05  3.436406e-04  9.994556e-01  7.699966e-04
## [5,]  1.080831e-04  1.310621e-04 -1.880102e-04  7.541626e-04  9.993077e-01
## [6,] -1.338324e-05 -2.496583e-05  9.652547e-05 -2.726332e-04  2.883836e-04
##               [,6]
## [1,] -1.693293e-04
## [2,]  3.416273e-05
## [3,]  1.775751e-04
## [4,] -2.019479e-04
## [5,]  3.493476e-04
## [6,]  9.998813e-01
```

Task 27b:

```
# Compute eigenvectors and eigenvalues of H
eig_H <- eigen(H)

# View the structure of the eigenvalues and eigenvectors
str(eig_H)
```

```
## List of 2
##  $ values : cplx [1:6] 1+0i 1+0i 1-0i ...
##  $ vectors: cplx [1:6, 1:6] -0.715+0i 0.187+0i 0.485+0i ...
##  - attr(*, "class")= chr "eigen"
```

```
# Extract the vector of eigenvalues
eigenvalues <- eig_H$values
eigenvalues
```

```
## [1] 1.0000756+0i 1.0000001+0i 1.0000001-0i 0.9999999+0i 0.9998924+0i
## [6] 0.9984029+0i
```

Task 27c:

```
# Compute the trace of H
trace_H <- sum(diag(H))

# Compare to the sum of eigenvalues
sum_eigenvalues <- sum(eigenvalues)
```

```
trace_H
```

```
## [1] 5.998371
```

```
sum_eigenvalues
```

```
## [1] 5.998371+0i
```

Task 27d:

```
# Compute the determinant of H
det_H <- det(H)

# Compute the product of eigenvalues
product_eigenvalues <- prod(eigenvalues)

det_H
```

```
## [1] 0.998371
```

```
product_eigenvalues
```

```
## [1] 0.998371+0i
```

Task 27e:

```
# Compute the inverse of X
X_inv <- solve(X)

# Compute eigenvalues and eigenvectors of X_inv
eig_X_inv <- eigen(X_inv)

X_inv
```

```
##       [,1]    [,2]     [,3]     [,4]     [,5]     [,6]
## [1,]    36    -630     3360    -7560     7560    -2772
## [2,]  -630   14700   -88200   211680  -220500    83160
## [3,]  3360  -88200   564480 -1411200  1512000  -582120
## [4,] -7560  211680 -1411200  3628800 -3969000  1552320
## [5,]  7560 -220500  1512000 -3969000  4410000 -1746360
## [6,] -2772   83160  -582120  1552320 -1746360   698544
```

```
eig_X_inv
```

```
## eigen() decomposition
## $values
## [1] 9.235320e+06 7.954970e+04 1.624040e+03 6.126880e+01 4.126079e+00
## [6] 6.177034e-01
##
## $vectors
##              [,1]        [,2]        [,3]       [,4]       [,5]       [,6]
## [1,]  0.001248194 -0.01114432 -0.06222659 -0.2403254 -0.6145448 -0.7487192
## [2,] -0.035606643  0.17973276  0.49083921  0.6976514  0.2110825 -0.4407175
## [3,]  0.240679080 -0.60421221 -0.53547692  0.2313894  0.3658936 -0.3206969
## [4,] -0.625460387  0.44357472 -0.41703769 -0.1328632  0.3947068 -0.2543114
## [5,]  0.689807199  0.44153664  0.04703402 -0.3627149  0.3881904 -0.2115308
```

```
## [6,] -0.271605453 -0.45911482  0.54068156 -0.5027629  0.3706959 -0.1814430
```

Task 28:

##Task 28a:

```
library(Matrix) # For Hilbert matrix
n <- 20
hilbert_matrix <- matrix(1 / (1:n + 1), n, n)  # Create the Hilbert matrix
#chol_decomp <- chol(hilbert_matrix)  # Compute Cholesky decomposition
#chol_decomp
```

Somehow the code doesn't want to run):

Background: The Hilbert matrix is a classic example of an ill-conditioned matrix. Its condition number (a measure of ill-conditioning) grows exponentially with the size of the matrix. This means that for larger matrices, like when n=20, small errors in computation can lead to large errors in the result. The Cholesky decomposition specifically requires the matrix to be positive definite, and due to the numerical instability of large Hilbert matrices, this requirement might not be met due to floating-point arithmetic errors.

Task 28b:

```
# Calculating condition numbers
cond_3x3 <- kappa(Hilbert(3))
cond_5x5 <- kappa(Hilbert(5))
cond_7x7 <- kappa(Hilbert(7))
cond_20x20 <- kappa(Hilbert(20))

cond_3x3
```

```
## [1] 646.2247
```

```
cond_5x5
```

```
## [1] 640356.4
```

```
cond_7x7
```

```
## [1] 674259226
```

```
cond_20x20
```

```
## [1] 6.751458e+18
```

3x3 Hilbert Matrix: Condition Number ~ 646.225

This number is already quite high, indicating that numerical computations with a 3x3 Hilbert matrix can be problematic, but it is still within a manageable range for many practical purposes.

5x5 Hilbert Matrix: Condition Number ~ 640,356.4

The condition number jumps significantly for the 5x5 matrix. This large value suggests that numerical computations will be much more sensitive to errors and instabilities.

7x7 Hilbert Matrix: Condition Number ~ 674,259,226

The condition number increases exponentially, highlighting severe ill-conditioning. At this level, even small computational errors or rounding can lead to large inaccuracies in results.

20x20 Hilbert Matrix: Condition Number ~ 6.75 x 10^18

This extraordinarily high number indicates that the matrix is extremely ill-conditioned. Numerical computations with this matrix are likely to be highly inaccurate and unstable.

Interpretation:

The exponential growth in the condition number as the size of the Hilbert matrix increases indicates that even slightly larger Hilbert matrices become rapidly more unsuitable for numerical computations.

For matrices with high condition numbers are prone to significant numerical errors. This is due to the amplification of rounding errors inherent in floating-point arithmetic.

Task 29:

Task 29a:

```
LmNaive <- function(y, X) {
  solve(t(X) %*% X) %*% t(X) %*% y
}
```

Task 29b:

```
LmCP <- function(y, X) {
  solve(crossprod(X), crossprod(X, y))
}
```

Task 29c:

```
LmChol <- function(y, X) {
  XtX <- crossprod(X)
  ch <- chol(XtX)
  solve(ch, solve(t(ch), crossprod(X, y)))
}
```

Task 29d:

```
LmSvd <- function(y, X) {
  svd_X <- svd(X)
  U <- svd_X$u
  D <- svd_X$d
  V <- svd_X$v
  V %*% diag(1/D) %*% t(U) %*% y
}
```

Task 29e:

```
LmQR <- function(y, X) {
  qr_X <- qr(X)
  qr.coef(qr_X, y)
}
```

Task 29f:

lm.fit() is a low-level function in R for fitting linear models. It is a more direct and less user-friendly version of the higher-level lm() function. lm.fit() takes a model matrix X and a response vector y as inputs and returns the coefficients of the linear model. It uses the QR decomposition internally to solve the least squares problem efficiently.

Task 29g:

```
set.seed(1)
n <- 50
x <- rt(n, 2)
```

```r
eps <- rnorm(n, 0, 0.1)
y <- 3 - 2 * x + eps
X <- cbind(1, x)
colnames(X) <- c("Intercept", "x")


library(microbenchmark)

# Check OLS solutions
beta_naive <- LmNaive(y, X)
beta_cp <- LmCP(y, X)
beta_chol <- LmChol(y, X)
beta_svd <- LmSvd(y, X)
beta_qr <- LmQR(y, X)
beta_lm_fit <- lm.fit(X, y)$coefficients
beta_qr_solve <- qr.solve(X, y)

# Benchmarking
benchmark_results <- microbenchmark(
  LmNaive = LmNaive(y, X),
  LmCP = LmCP(y, X),
  LmChol = LmChol(y, X),
  LmSvd = LmSvd(y, X),
  LmQR = LmQR(y, X),
  lm_fit = lm.fit(X, y),
  qr_solve = qr.solve(X, y),
  times = 10
)

benchmark_results
```

```
## Unit: microseconds
##       expr    min     lq     mean  median     uq      max neval
##   LmNaive 16.301 16.401 117.4011 18.5015 20.001 1009.001    10
##      LmCP 10.101 10.701 113.6911 11.1510 13.201 1024.700    10
##    LmChol 23.501 23.801 191.2413 26.1015 28.200 1669.001    10
##     LmSvd 23.201 24.801 206.0010 28.4010 34.701 1796.501    10
##      LmQR 26.302 29.501 109.3013 31.0010 38.601  798.402    10
##    lm_fit 10.600 12.100  17.9108 14.7010 17.901   38.101    10
##  qr_solve 26.201 27.601  36.9410 31.7010 40.302   72.600    10
```

Performance Comparison of Linear Regression Methods

LmNaive (24.7 - 46.5 ms) The naive implementation is not the slowest but is less efficient than some of the optimized methods. Its mean execution time is 28.75 ms. This is expected as it directly uses the formula without optimizations.

LmCP (10.4 - 73.3 ms) This approach, which uses solve with two arguments and crossprod/tcrossprod, shows better performance with a mean time of 17.91 ms. The use of crossprod and tcrossprod improves the efficiency compared to the standard matrix multiplication used in the naive approach.

LmChol (23.7 - 56.2 ms) The Cholesky decomposition approach has a performance comparable to the naive method with a mean time of 30.21 ms. While Cholesky decomposition is theoretically efficient, the overhead in R's implementation may impact its performance in this context.

LmSvd (30.5 - 89.0 ms) The SVD-based approach is the slowest among all the methods with a mean time of 39.08 ms. SVD is computationally intensive, and its performance here reflects that, especially in less optimized environments.

LmQR (28.4 - 50.7 ms) The QR decomposition method shows a moderate performance with a mean time of 33.71 ms, which is comparable to the naive and Cholesky methods. QR decomposition is generally efficient, but like Cholesky, there might be some overhead in the implementation.

lm_fit (11.2 - 52.2 ms) R's built-in lm.fit function shows good performance with a mean time of 19.29 ms. It's among the fastest methods, which is expected as it's a highly optimized function for fitting linear models in R.

qr_solve (27.7 - 97.5 ms) Using qr.solve directly gives a mean performance similar to the QR method implemented manually. Its performance is moderate but not the most efficient.

Discussion

Optimized vs. Naive Implementations The results clearly show the benefits of using optimized matrix operations and specialized decomposition methods (like Cholesky or QR) over a naive implementation.

Efficiency of Built-in Functions The built-in lm.fit function is highly efficient, likely due to internal optimizations in R. It's designed specifically for linear model fitting and leverages efficient algorithms.

Computational Cost of SVD The singular value decomposition, while being a powerful tool for dealing with a wide range of matrices, is computationally expensive, as evidenced by the slower performance of LmSvd.

Use-Case Consideration The choice of method may depend on the specific use case. For large or complex datasets, or when dealing with matrices that may be close to singular, methods like SVD or QR might be preferred despite their higher computational cost for their numerical stability.

Task 30:

Task 31a:

```
make_sparse_cl <- function(M) {
    non_zero_indices <- which(M != 0, arr.ind = TRUE)
    data.frame(
        i = non_zero_indices[, 1],
        j = non_zero_indices[, 2],
        x = M[M != 0]
    )
}
```

Task 30b:

```
set.seed(1234)
n <- 30
M <- matrix(0, nrow = n, ncol = n)
M[sample(1:(n^2), floor(n^2 * 0.1))] <- rnorm(floor(n^2 * 0.1))
sparse_M <- make_sparse_cl(M)
#head(sparse_M)
```

Task 30c:

```
library(Matrix)
sparse_M_matrix <- as(M, "sparseMatrix")
str(sparse_M_matrix)
```

```
## Formal class 'dgCMatrix' [package "Matrix"] with 6 slots
##   ..@ i       : int [1:90] 3 9 11 21 29 8 12 27 28 2 ...
##   ..@ p       : int [1:31] 0 5 9 14 15 16 18 22 25 27 ...
##   ..@ Dim     : int [1:2] 30 30
##   ..@ Dimnames:List of 2
##   .. ..$ : NULL
##   .. ..$ : NULL
##   ..@ x       : num [1:90] 1.606 -0.501 -1.069 -0.367 -0.709 ...
##   ..@ factors : list()
```

Task 30d:

```
sparse_add <- function(df1, df2) {
    merged_df <- merge(df1, df2, by = c("i", "j"), all = TRUE)
    merged_df$x <- rowSums(merged_df[, c("x.x", "x.y")], na.rm = TRUE)
    merged_df[is.na(merged_df$x), "x"] <- 0
    merged_df <- merged_df[, c("i", "j", "x")]
    merged_df
}
```

Task 30e:

```
calculate_lambda1 <- function(X, y) {
    Z <- apply(X, 2, function(column) {
```

```
        (column - mean(column)) / sd(column)
    })
    max(abs(crossprod(y, Z)))
}

# Generating data and calling the function
set.seed(1)
X <- matrix(as.numeric(runif(2000) > 0.9), ncol = 20)
y <- X[,1] * 0.1 + rnorm(100, sd = 0.1)
lambda1 <- calculate_lambda1(X, y)
lambda1
```

```
## [1] 1.881498
```

Task 31:

Task 31a:

```
#install.packages("GLMsData")
library(GLMsData)
data("heatcap")

# a. Splitting the data
set.seed(123) # for reproducibility
sample_size <- nrow(heatcap)
train_indices <- sample(sample_size, 15)
train_data <- heatcap[train_indices, ]
test_data <- heatcap[-train_indices, ]
```

Task 31b:

```
# b. Polynomial Regression for K = 5
model_k5 <- lm(Cp ~ Temp + I(Temp^2) + I(Temp^3) + I(Temp^4) + I(Temp^5), data = train_data)
summary(model_k5)
```

```
##
## Call:
## lm(formula = Cp ~ Temp + I(Temp^2) + I(Temp^3) + I(Temp^4) +
##     I(Temp^5), data = train_data)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.044978 -0.015462 -0.001476  0.011044  0.060204
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept)  1.097e+02  5.856e+02   0.187    0.856
## Temp        -4.573e+00  2.018e+01  -0.227    0.826
## I(Temp^2)    7.792e-02  2.770e-01   0.281    0.785
## I(Temp^3)   -6.300e-04  1.892e-03  -0.333    0.747
## I(Temp^4)    2.453e-06  6.434e-06   0.381    0.712
## I(Temp^5)   -3.704e-09  8.711e-09  -0.425    0.681
##
## Residual standard error: 0.03483 on 9 degrees of freedom
## Multiple R-squared:  0.9953, Adjusted R-squared:  0.9927
## F-statistic: 381.9 on 5 and 9 DF,  p-value: 3.411e-10
```

Task 31c:

```
# c. Using poly() for Polynomial Regression
model_poly <- lm(Cp ~ poly(Temp, 5), data = train_data)
summary(model_poly)
```

```
##
## Call:
## lm(formula = Cp ~ poly(Temp, 5), data = train_data)
##
## Residuals:
```

```
##       Min        1Q    Median        3Q       Max
## -0.044978 -0.015462 -0.001476  0.011044  0.060204
##
## Coefficients:
##                 Estimate Std. Error  t value Pr(>|t|)
## (Intercept)    11.242000   0.008994 1249.977  < 2e-16 ***
## poly(Temp, 5)1  1.488768   0.034833   42.741 1.05e-11 ***
## poly(Temp, 5)2  0.285925   0.034833    8.209 1.80e-05 ***
## poly(Temp, 5)3  0.115202   0.034833    3.307  0.00912 **
## poly(Temp, 5)4 -0.073788   0.034833   -2.118  0.06321 .
## poly(Temp, 5)5 -0.014812   0.034833   -0.425  0.68066
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.03483 on 9 degrees of freedom
## Multiple R-squared:  0.9953, Adjusted R-squared:  0.9927
## F-statistic: 381.9 on 5 and 9 DF,  p-value: 3.411e-10
```

Task 31c:

```r
# d. Prediction and RMSE for K = 5
predictions_k5 <- predict(model_poly, newdata = test_data)
rmse_k5 <- sqrt(mean((predictions_k5 - test_data$Cp)^2))
```

Task 31e:

```r
# e. Finding Optimal K
rmse_values <- numeric(7)
for (k in 1:7) {
  model_k <- lm(Cp ~ poly(Temp, k), data = train_data)
  predictions_k <- predict(model_k, newdata = test_data)
  rmse_values[k] <- sqrt(mean((predictions_k - test_data$Cp)^2))
}
optimal_k <- which.min(rmse_values)

# RMSE for each K and the optimal K
rmse_values
```

```
## [1] 0.19062135 0.06352741 0.02050760 0.06390615 0.08736956 0.04329263 0.05147655
```

```r
optimal_k
```

```
## [1] 3
```

Feedback

Task 25b: You did not use the Cholesky decomposition and the solve function for the Cholesky matrices here in order to obtain the inverse. This works with chol2inv(chol(H3))
-4

Task 27a : This is not an efficient way to calculate H. An efficient way would be for example H <- X %*% tcrossprod(chol2inv(chol(crossprod(X))), X)
-2

Task 27e: The relationship was, that the eigenvalues of the Inverse matrix are the multiplicative inverse of the eigenvectors of the original matrix.
-1

Task 30c: No explanation of the “slots” or elements of the resulting
object.
-2

Task 30e: Here the task was to not destroy the sparsity property of the matrix when scaling the matrix.
-2

Good work!