Métodos Computacionais para Estatística e Otimização

Lista 5 - Luiz Henrique Barretta Francisco

1. A decomposição de Cholesky de uma matriz simétrica e definida positiva é muito popular em estatística. Para você ter uma ideia do algoritmo consulte. Use três diferentes abordagens para obter a decomposição de Cholesky de uma matriz positiva definida em R e/ou C++. As abordagens podem ser diferentes pacotes, diferentes classes ou mesmo diferentes linguagens. Se você conhece outras linguagens pode usar se julgar adequado. Para criar uma matriz positiva definida use o seguinte código. Tome cuidado com a classe das matrizes que você vai utilizar para comparar as diferentes abordagens. Considere matrizes de diferentes dimensões e use o pacote bench para a comparação em termos de tempo computacional. Importante explique cuidadosamente a diferença entre as abordagens e qual você julga ser a mais eficiente antes e após realizar o experimento computacional.

```r
library(Matrix)
library(Rcpp)
library(bench)

set.seed(123)
x1 <- runif(30)
x2 <- runif(30)
grid <- expand.grid(x1, x2)
DD <- dist(grid, diag = TRUE, upper = TRUE)
DD_positiva <- exp(as.matrix(-DD, 100, 100)/0.3)

chol_base <- function(mat) {
  chol(mat)}

chol_matrix <- function(mat) {
  as.matrix(Matrix::chol(Matrix::Matrix(mat, sparse = FALSE)))}

cppFunction('
  NumericMatrix chol_cpp(NumericMatrix x) {
    int n = x.nrow();
    NumericMatrix U(n, n);
    for (int j = 0; j < n; j++) {
      for (int i = 0; i <= j; i++) {
        double sum = 0;
        for (int k = 0; k < i; k++) {
          sum += U(k,i) * U(k,j);
        }
        if (i == j) {
          U(i,j) = sqrt(x(i,i) - sum);
        } else {
          U(i,j) = (x(i,j) - sum) / U(i,i);
        }
      }
    }
```

```
    return U;
  }
')


chol_rcpp <- function(mat) {chol_cpp(mat)}

mark(baseR = chol_base(DD_positiva),
     matrixPkg = chol_matrix(DD_positiva),
     rcpp = chol_rcpp(DD_positiva),
     check = FALSE)
```

```
## # A tibble: 3 x 6
##   expression      min   median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
## 1 baseR          48ms   49.2ms     20.4     6.22MB     7.63
## 2 matrixPkg      62ms     62ms     16.1    53.52MB   129.
## 3 rcpp         99.5ms  101.5ms      9.74    6.18MB     2.44
```

A abordagem base do R *(chol)* usa a implementação interna altamente otimizada do R, que chama rotinas da biblioteca *LAPACK* compiladas em C. Ela é rápida, segura e muito eficiente para matrizes densas, apresentando baixa alocação de memória e bom número de execuções por segundo no experimento.

A abordagem usando o pacote Matrix *(Matrix::chol)* converte a matriz para um objeto S4 e opera com funções específicas para grandes matrizes (esparsas ou densas). Apesar de ser flexível para estruturas maiores e mais complexas, no experimento consumiu muito mais memória e foi ligeiramente mais lenta que o chol do R base.

A abordagem via C++ *(chol_cpp)* é uma implementação manual da decomposição de Cholesky, feita para estudo. Ela não usa otimizações de bibliotecas numéricas especializadas e, por isso, no experimento foi significativamente mais lenta que as abordagens do R, mesmo com alocação de memória semelhante.

Portanto, considerando desempenho computacional e consumo de memória, a abordagem mais eficiente e adequada para matrizes densas como no experimento é a função chol do R base, combinando velocidade, estabilidade e baixo custo de memória.

2. Nas mesmas condições do exercício 1. Considere que é de interesse obter a decomposição em autovalores e autovetores. Novamente forneça três alternativas e compare os tempos computacionais.

```
eigen_base <- function(mat) {
  eigen(mat, symmetric = TRUE)}

eigen_matrix <- function(mat) {
  as.list(eigen(Matrix::Matrix(mat, sparse = FALSE), symmetric = TRUE))}

cppFunction('
List eigen_cpp(NumericMatrix x) {
  Environment base("package:base");
  Function eigen = base["eigen"];
  List out = eigen(x, Named("symmetric", true));
  return out;}')

eigen_rcpp <- function(mat) {
  eigen_cpp(mat)}
```

```r
mark(baseR = eigen_base(DD_positiva),
     matrixPkg = eigen_matrix(DD_positiva),
     rcpp = eigen_rcpp(DD_positiva))
```

```
## # A tibble: 3 x 6
##   expression      min   median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
## 1 baseR         482ms    489ms      2.05    28.1MB     2.05
## 2 matrixPkg     586ms    586ms      1.71    74.9MB     5.12
## 3 rcpp          501ms    501ms      2.00    28.1MB     0
```

Considerando o menor tempo de execução e a eficiência de memória, a abordagem via *Rcpp* se mostrou a mais adequada neste experimento. Apesar de não ser uma implementação nativa de autovalores em C++, o encapsulamento do eigen() dentro de Rcpp parece ter reduzido a sobrecarga e trouxe ganhos de performance claros em comparação às outras alternativas.

3. Considere um modelo linear de covariancia com matriz de covariância descrita por dois componentes conforme código abaixo. A (i, j)-ésima entrada da matriz de sensitividade para estimação dos parâmetros de dispersão é dada por

$$S_{\tau_{ij}} = -tr(W_{\tau_i} C W_{\tau_j} C),$$

onde $C = \tau_1 I + \tau_2 Z$ e $W_{\tau_i} = \frac{\delta C}{\delta \tau_i}$ para $i = 1, 2$. Note que no caso de dois parâmetros (τ_1, τ_2) a matriz de sensitividade é 2×2. Proponha três estratégias para obter a matriz de sensitivade. Compare suas propostas pelo tempo computacional. Avalie matrizes de diferentes tamanhos. Considere que a matriz Z é bloco diagonal com estrutura dada pelo código abaixo.

```r
library(Matrix)
Z_temp <- rep(1, 5)%*%t(rep(1, 5))
Z_lista <- list()
for(i in 1:10) {Z_lista[[i]] <- Z_temp}
Z <- Matrix::bdiag(Z_lista)
C <- 5*Diagonal(50, 1) + 3*Z

tau1 <- 5
tau2 <- 3
n <- nrow(C)
W1 <- Diagonal(n, 1)
W2 <- Z

sens_base <- function(C, W1, W2) {
  C_inv <- solve(C)
  S <- matrix(NA, 2, 2)
  S[1,1] <- -sum(diag(W1 %*% C_inv %*% W1 %*% C_inv))
  S[1,2] <- -sum(diag(W1 %*% C_inv %*% W2 %*% C_inv))
  S[2,1] <- -sum(diag(W2 %*% C_inv %*% W1 %*% C_inv))
  S[2,2] <- -sum(diag(W2 %*% C_inv %*% W2 %*% C_inv))
  S}

sens_matrix <- function(C, W1, W2) {
```

```
  C_inv <- solve(C)
  S <- matrix(NA, 2, 2)
  S[1,1] <- -sum(diag(W1 %*% C_inv %*% W1 %*% C_inv))
  S[1,2] <- -sum(diag(W1 %*% C_inv %*% W2 %*% C_inv))
  S[2,1] <- -sum(diag(W2 %*% C_inv %*% W1 %*% C_inv))
  S[2,2] <- -sum(diag(W2 %*% C_inv %*% W2 %*% C_inv))
  S}

sens_fast <- function(C, W1, W2) {
  C_inv <- solve(C)
  WC1 <- W1 %*% C_inv
  WC2 <- W2 %*% C_inv
  S <- matrix(NA, 2, 2)
  S[1,1] <- -sum(WC1 * t(WC1))
  S[1,2] <- -sum(WC1 * t(WC2))
  S[2,1] <- -sum(WC2 * t(WC1))
  S[2,2] <- -sum(WC2 * t(WC2))
  S}


mark(baseR = sens_base(C, W1, W2),
     matrixPkg = sens_matrix(C, W1, W2),
     fast = sens_fast(C, W1, W2),
     check = FALSE)
```

```
## # A tibble: 3 x 6
##   expression      min   median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
## 1 baseR       111.3us  129.1us     7091.   212.9KB     14.9
## 2 matrixPkg   113.3us  133.5us     6046.    69.6KB     12.7
## 3 fast         1.54ms    1.9ms      485.   330.2KB     12.8
```

A abordagem baseR realiza o cálculo da matriz de sensitividade diretamente utilizando produtos de matrizes e extração das diagonais, aproveitando funções básicas do R, com bom desempenho. A abordagem Matrix segue o mesmo procedimento, mas utilizando objetos da classe Matrix, o que reduz um pouco a alocação de memória sem impacto relevante no tempo de execução. Já a abordagem fast tenta otimizar o cálculo ao pré-computar produtos intermediários, evitando múltiplas multiplicações de matrizes, mas, para matrizes pequenas como no exemplo, isso acaba tornando o processo mais lento devido ao maior custo da manipulação intermediária.

Portanto, considerando o tempo computacional e o consumo de memória, a abordagem Matrix é a mais adequada para este problema, pois manteve a velocidade do método básico com menor consumo de memória, o que a torna mais eficiente especialmente para escalas maiores.

4. A distribuição Normal multivariada tem uma ampla gama de aplicações em estatística. Exemplos incluem análise de componentes principais, regressão multivariada, análise de variância multivariada, análise fatorial entre outras. Dizemos que um vetor aleatório Y de dimensão n $\times$ 1 tem distribuição normal multivariada se sua função densidade probabilidade é dada por

$$f(y|\mu,\Sigma) = \frac{1}{2\pi}^{n/2} |\Sigma|^{-1/2} exp[-\frac{1}{2}(y-\mu)^\top \Sigma^{-1}(y-\mu)],$$

onde μ é um vetor $n \times 1$ de valores esperados e Σ é uma matriz $(n \times n)$ simétrica e positiva definida. É usual representar esta situação pela notação $Y \sim N(\mu, \Sigma)$. E neste caso temos que $E(Y) = \mu$ e $V(Y) = \Sigma$. Implemente esta função usando pelo menos três diferentes abordagens de álgebra linear. Note que a implementação desta função depende de três componentes chaves: i) Cálculo do determinante de Σ; ii) Multiplicação de matrizes e iii) Inversa da matriz Σ. Você pode usar qualquer estratégia que achar conveniente relacionado a propriedades de matrizes e/ou resolução de sistemas lineares. Você precisará especificar o vetor μ para o qual sugiro usar um vetor de zeros e para a matriz Σ que deve ser positiva definida. Considere vetores de tamanho entre 10 e 100.

O pacote *mvtnorm* do software R forcene uma implementação de tal distribuição através da função *dmvnorm()*. Use esta implementação como base de comparação e faça com que sua função seja mais rápida. O código abaixo ilustra o uso do pacote para avaliar a distribuição normal multivariada.

```
library(mvtnorm)

dmvnorm_base <- function(x, mu, Sigma) {
  n <- length(x)
  inv_Sigma <- solve(Sigma)
  det_Sigma <- det(Sigma)
  quad_form <- t(x - mu) %*% inv_Sigma %*% (x - mu)
  dens <- (2 * pi)^(-n / 2) * det_Sigma^(-0.5) * exp(-0.5 * quad_form)
  as.numeric(dens)}

dmvnorm_chol <- function(x, mu, Sigma) {
  n <- length(x)
  L <- chol(Sigma)
  y <- forwardsolve(t(L), x - mu)
  quad_form <- sum(y^2)
  log_det <- 2 * sum(log(diag(L)))
  dens <- exp(-0.5 * (n * log(2 * pi) + log_det + quad_form))
  dens}

dmvnorm_matrix <- function(x, mu, Sigma) {
  n <- length(x)
  Sigma <- Matrix(Sigma, sparse = FALSE)
  L <- Cholesky(Sigma, LDL = FALSE)
  y <- solve(L, x - mu, system = "Lt")
  quad_form <- sum(y^2)
  log_det <- 2 * sum(log(diag(L@x)))
  dens <- exp(-0.5 * (n * log(2 * pi) + log_det + quad_form))
  dens}

set.seed(123)
n <- 50
x <- rnorm(n)
mu <- rep(0, n)
temp <- matrix(rnorm(n^2), n, n)
Sigma <- crossprod(temp) + diag(n)  # matriz positiva definida

mark(baseR = dmvnorm_base(x, mu, Sigma),
     chol = dmvnorm_chol(x, mu, Sigma),
     matrixPkg = dmvnorm_matrix(x, mu, Sigma),
     mvtnorm = dmvnorm(x, mean = mu, sigma = Sigma),
     check = FALSE)
```

```
## # A tibble: 4 x 6
##   expression      min   median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
## 1 baseR       114.4us  128.2us   6842.     232.1KB    10.0
## 2 chol         30.4us   35.6us  23799.      68.4KB    14.3
## 3 matrixPkg   799.5ms  799.5ms      1.25      48MB     1.25
## 4 mvtnorm     103.1us  130.7us   5362.     184.5KB    10.0
```

A abordagem *baseR* usa operações diretas como *solve()* para calcular a inversa da matriz de covariância e *det()* para o determinante, com uma implementação simples mas computacionalmente custosa para inversões, especialmente em matrizes maiores. A abordagem Cholesky evita a inversa explícita, resolvendo o sistema linear através da fatoração de Cholesky e computando o determinante de forma eficiente pela soma dos logaritmos das diagonais, resultando em um ganho expressivo de velocidade e estabilidade numérica. A abordagem Matrix usa o pacote *Matrix* para trabalhar com objetos de álgebra linear especializados e fatorações otimizadas para grandes dimensões, porém, para matrizes moderadas como no experimento, essa estrutura acaba gerando sobrecarga e alta alocação de memória, tornando o processo mais lento. A função *mvtnorm::dmvnorm()* é uma implementação geral confiável que internamente também usa otimizações, mas realiza verificações adicionais, o que deixa sua velocidade ligeiramente inferior à abordagem de Cholesky pura.

Assim, considerando o tempo computacional e a simplicidade de implementação, a abordagem Cholesky se mostrou a melhor opção. Ela foi a mais rápida, com cerca de 22.451 execuções por segundo, usou pouca memória, e evitou o custo pesado de inversão de matriz, além de ser mais robusta numericamente, superando inclusive a função *dmvnorm()* do pacote *mvtnorm*.

5. O seguinte post clique aqui apresenta várias considerações relacionadas a performance computacionalde cálculos matriciais envolvendo matrizes esparsas com o Rcpp e Armadillo. Leia o artigo, reproduza o código e faça um resumo do que o post apresenta.

O artigo "Performance considerations with sparse matrices in Armadillo" explora como a eficiência das operações com matrizes esparsas depende da esparsidade dos dados e do padrão de acesso (linhas vs colunas). Multiplicar duas matrizes esparsas é mais eficiente do que misturar esparsas e densas, e converter matrizes densas para formato esparso pode melhorar o desempenho. O Armadillo usa internamente o formato Compressed Sparse Column (CSC), o que torna o acesso por colunas muito mais rápido que o acesso por linhas. Para evitar penalidades de performance, recomenda-se realizar o máximo de operações diretamente no C++, evitando chamadas excessivas entre R e C++. Além disso, o artigo destaca que iterar apenas sobre elementos não nulos usando iteradores é muito mais eficiente que percorrer toda a matriz. O foco é maximizar a performance com boas práticas específicas para dados esparsos em ambientes mistos R/C++.

Gera duas matrizes esparsas e realiza a multiplicação entre elas.

```r
A <- rsparsematrix(1000, 1000, density = 0.01)
B <- rsparsematrix(1000, 1000, density = 0.01)
C <- A %*% B
```

Cria uma matriz densa esparsa e converte para o formato de matriz esparsa antes de multiplicar.

```r
Densa <- matrix(rbinom(10000, 1, 0.01), nrow = 100)
Esparsa <- Matrix(Densa, sparse = TRUE)
Resultado <- Esparsa %*% Esparsa
```

Mostra que o acesso por coluna é mais rápido do que por linha em matrizes esparsas.

```
bench::mark(
  acesso_linha = {
    linha_10 <- Esparsa[10, ]
  },
  acesso_coluna = {
    coluna_10 <- Esparsa[, 10]
  },
  check = FALSE
)
```

```
## # A tibble: 2 x 6
##   expression         min   median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr>    <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
## 1 acesso_linha    27.5us   31.6us    25764.     117KB     15.5
## 2 acesso_coluna   27.3us   30.4us    28248.    32.1KB     19.8
```

Multiplica duas matrizes esparsas diretamente em C++ usando Armadillo.

```
library(RcppArmadillo)

cppFunction(depends = "RcppArmadillo", code = '
arma::sp_mat multiplySparse(arma::sp_mat A, arma::sp_mat B) {
  return A * B;
}
')

A2 <- as(Esparsa, "dgCMatrix")
B2 <- as(Esparsa, "dgCMatrix")
Resultado_cpp <- multiplySparse(A2, B2)
```

Percorre apenas os elementos não nulos da matriz esparsa e calcula a soma deles.

```
cppFunction(depends = "RcppArmadillo", code = '
double sumSparse(arma::sp_mat A) {
  double total = 0.0;
  for (arma::sp_mat::const_iterator it = A.begin(); it != A.end(); ++it) {
    total += *it;
  }
  return total;
}
')

soma_elementos <- sumSparse(A2)
soma_elementos
```

```
## [1] 97
```