Biostatistics 615 - Statistical Computing

Topic 3
Matrix Computation

Hyun Min Kang

Topic Overview

1. Algorithms for matrix computation
2. Matrix decomposition and solving linear systems
3. Sparse matrices

Topic Overview

1. Algorithms for matrix computation
2. Matrix decomposition and solving linear systems
3. Sparse matrices

Programming with Matrix

Why matrix matters?

- Many statistical models can be well represented as matrix operations
  - Linear regression
  - Logistic regression
  - Mixed models
- Efficient matrix computation can make difference in the practicality of a statistical method
- Understanding R implementation of matrix operation can expedite the efficiency by orders of magnitude

Recap : Divide-and-conquer algorithms

Solve a problem recursively, applying three steps at each level of recursion

Divide the problem into a number of subproblems that are smaller instances of the same problem

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to subproblems into the solution for the original problem

Matrix multiplication

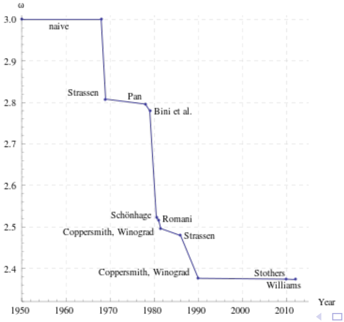
Let $A \in \mathbb{R}^{n \times p}$, and $B \in \mathbb{R}^{p \times m}$. Define $C \in \mathbb{R}^{n \times m}$ such that

$$C_{ij} = \sum_{k=1}^{p} A_{ik} B_{kj} \qquad (i \in \{1, \ldots, n\}, j \in \{1, \ldots m\})$$

- What is the time complexity for the algorithm to implement this?
- When $n = m = p$, what is the complexity? Any faster way?

Time complexity of square matrix multiplication

- For two matrices of dimension $n \times n$:
- Naive algorithm : $O(n^3)$
- Strassen algorithm (1969): $O(n^{2.807})$ (the fastest practical algorithm)
- Coppersmith-Winograd algorithm (1990): $O(n^{2.376})$
- François Le Gall (2014): $O(n^{2.373})$ (the best known algorithm)
- The best known lower bound: $\Omega(n^2)$ (or $\Omega(n^2 \log n)$ with certain assumptions).



Strassen algorithm (Volker Strassen, 1969)

Goal: Given A, B, compute $C = AB$, where A, B, C are matrices of size $n \times n$ where $n = 2^k$.

Step 1: Partition A, B, C into submatrices of size $2^{k-1} \times 2^{k-1}$:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}, B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}, C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}.$$

Step 2: Compute the followings matrices:

$$\begin{aligned} M_1 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\ M_2 &= (A_{2,1} + A_{2,2})B_{1,1} \\ M_3 &= A_{1,1}(B_{1,2} - B_{2,2}) \\ M_4 &= A_{2,2}(B_{2,1} - B_{1,1}) \\ M_5 &= (A_{1,1} + A_{1,2})B_{2,2} \\ M_6 &= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \\ M_7 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}) \end{aligned}$$

(http://en.wikipedia.org/wiki/Strassen_algorithm)

Strassen algorithm (cont.)

Step 3: Compute the followings matrices:

$$
\begin{aligned}
C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} = M_1 + M_4 - M_5 + M_7 \\
C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} = M_3 + M_5 \\
C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} = M_2 + M_4 \\
C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2} = M_1 - M_2 + M_3 + M_6
\end{aligned}
$$

Time complexity analysis

$$T(n) = 7\,T(n/2) + O(n^2)$$

Applying the master theorem from the last lecture,
$T(n) = O(n^{\log_2 7}) = O(n^{2.807})$.

Time complexity is not everything

- Time complexity only explains how scalable the algorithm is *relatively* to the increase in the size of input data.
- The *absolute* computational time of an algorithm may depend on the implementation details.
- For example, using loop inside is R is not usually recommended, because it can slow down the implementation by a lot.

Hands-on Session

Visit https://bit.ly/615top03r

- Triple-loop matrix multiplication
- Double-loop matrix multiplication

R implementations of matrix multiplication

Lessons learned:

- Double-loop implementation is an order of magnitude faster than Triple-loop implementation.
  - Did the time complexity change? If not, why is it faster?

R implementations of matrix multiplication

Lessons learned:

- Double-loop implementation is an order of magnitude faster than Triple-loop implementation.
  - Did the time complexity change? If not, why is it faster?
  - Avoiding loops and using built-in R functions helps the efficiency.

R implementations of matrix multiplication

Lessons learned:

- Double-loop implementation is an order of magnitude faster than Triple-loop implementation.
  - Did the time complexity change? If not, why is it faster?
  - Avoiding loops and using built-in R functions helps the efficiency.
- Matrix multiplication built in R is still orders of magnitude faster than `v2`
  - Do you think the difference is due to time complexity?
  - Built-in R matrix operations are very well optimized through BLAS/LAPACK.
  - Naive Rcpp implementation would be faster than naive R implementation, but still slower than built-in R multiplication

Matrix operation with BLAS/LAPACK is very fast

What is BLAS/LAPACK?

- BLAS (Basic Linear Algebra Subprograms) is a software library that implements low-level routines for linear algebra
- LAPACK (Linear Algebra PACKage) is a software library that implements key algorithms for linear algebra such as matrix decomposition and linear systems solver

Why are BLAS/LAPACK so fast?

Vectorization BLAS/LAPACK uses low-level hardware/software techniques that expedite operations on arrays.

Multithreading Many BLAS/LAPACK makes use of multiple CPUs via multithreading whenever possible.

Cache Optimization BLAS/LAPACK are carefully implemented to optimize the utilization of cache, which is much faster (but limited) storage in CPU than typical memory.

GenAI's explanation on this question: ChatGPT, Gemini, Claude.

BLAS/LAPACK Implementations

There are 4 BLAS/LAPACK implementations typically available in major operating systems.

(GNU) default BLAS/LAPACK
: Typically installed. Typically not the fastest, and could be improved by replacing with other implementations.

OpenBLAS
: Open under BSD license. Typically faster than the default ones.

Intel MKL
: Open under Intel's ISSL license (used to have non-public license). Considered the fastest, but could be trickier to install.

ATLAS
: Open under BSD-like license. Similar performance with OpenBLAS, but installation may be trickier.

In this course, we will not teach how to install different BLAS/LAPACK libraries in your system, but it could affect your R/python performance dramatically.

Hands-on Session

Visit https://bit.ly/615top03r

- Understanding BLAS/LAPACK

Time complexity for matrix inversion

- Matrix inversion can be reduced to matrix multiplication!

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} = \begin{bmatrix} K^{-1} & -K^{-1}BD^{-1} \\ -D^{-1}CK^{-1} & D^{-1} + D^{-1}CK^{-1}BD^{-1} \end{bmatrix}$$

  where $K = A - BD^{-1}C$.
- Time complexity: $f(n) = 2f(n/2) + 6T(n/2) + O(n^2)$, where $T(n)$ is the time for matrix multiplication.
- Applying the master theorem, $f(n) = \Theta(T(n)) = O(n^{2.584})$.
- Best known algorithm: $O(n^{2.373})$
- Best known lower bound: $\Omega(n^2 \log n)$.

(http://en.wikipedia.org/wiki/Invertible_matrix#Methods_of_matrix_inversion)

Time complexity for matrix determinant

Determinant

- Laplace expansion : $O(n!)$
- LU decomposition : $O(n^3)$
- Bareiss algorithm : $O(n^3)$
- Matrix determinant can also be reduced to matrix multiplication : $O(n^{2.373})$

(http://en.wikipedia.org/wiki/Determinant#Calculation)

Computational considerations in matrix operations

Avoiding expensive computation

- Computation of $\mathbf{u}'AB\mathbf{v}$

Computational considerations in matrix operations

Avoiding expensive computation

- Computation of $\mathbf{u}'AB\mathbf{v}$
- If the order is $(((\mathbf{u}'(AB))\mathbf{v})$
  - $O(n^3) + O(n^2) + O(n)$ operations
  - $O(n^3)$ overall

Computational considerations in matrix operations

Avoiding expensive computation

- Computation of $\mathbf{u}'AB\mathbf{v}$
- If the order is $(((\mathbf{u}'(AB))\mathbf{v})$
  - $O(n^3) + O(n^2) + O(n)$ operations
  - $O(n^3)$ overall
- If the order is $(((\mathbf{u}'A)B)\mathbf{v})$
  - Two $O(n^2)$ operations and one $O(n)$ operation
  - $O(n^2)$ overall

Quadratic multiplication

Same time complexity, but one is slightly more efficient

- Computing $\mathbf{x}'A\mathbf{y}$.
- $O(n^2) + O(n)$ if ordered as $(\mathbf{x}'A)\mathbf{y}$.
- Can be simplified as $\sum_i \sum_j x_i A_{ij} y_j$

A symmetric case

- Computing $\mathbf{x}'A\mathbf{x}$ where $A = LL'$ (Cholesky decomposition)
- $\mathbf{u} = L'\mathbf{x}$ can be computed more efficiently than $A\mathbf{x}$.
- $\mathbf{x}'A\mathbf{x} = \mathbf{u}'\mathbf{u}$

(http://en.wikipedia.org/wiki/Cholesky_decomposition)

Topic Overview

1. Algorithms for matrix computation
2. Matrix decomposition and solving linear systems
3. Sparse matrices

Solving linear systems

Problem

Find $\boldsymbol{x}$ that satisfies $A\boldsymbol{x} = \boldsymbol{b}$

A simplest approach

- Calculate A^{-1}, and $\boldsymbol{x} = A^{-1}\boldsymbol{b}$
- Time complexity is $O(n^3) + O(n^2)$
- A has to be invertible
- Potential issue of numerical instability
- http://en.wikipedia.org/wiki/Invertible_matrix#Methods_of_matrix_inversion

Using matrix decomposition to solve linear systems

LU decomposition

- $A = LU$, making $U\mathbf{x} = L^{-1}\boldsymbol{b}$
- A needs to be square and invertible.
- Fewer additions and multiplications
- Precision problems may occur
- http://en.wikipedia.org/wiki/LU_decomposition#Algorithms

Cholesky decomposition

- A is a square, symmetric, and positive definite matrix.
- $A = U^T U$ is a special case of LU decomposition
- Computationally efficient and accurate
- http://en.wikipedia.org/wiki/Cholesky_decomposition#Computation

QR decomposition

- $A = QR$ where A is $m \times n$ matrix
- Q is orthogonal matrix, $Q^T Q = I$.
- R is $m \times n$ upper-triangular matrix, $R\mathbf{x} = Q^T\mathbf{b}$.
- http://en.wikipedia.org/wiki/QR_decomposition#Computing_the_QR_decomposition

Singular value decomposition

Definition

A $m \times n (m \geq n)$ matrix A can be represented as $A = UDV^T$ such that

- U is $m \times n$ matrix with orthogonal columns $(U^T U = I_n)$
- D is $n \times n$ diagonal matrix with non-negative entries
- V^T is $n \times n$ matrix with orthogonal matrix $(V^T V = VV^T = I_n)$.

Computational complexity

- $4m^2 n + 8mn^2 + 9m^3$ for computing U, V,and D.
- $4mn^2 + 8n^3$ for computing V and D only.
- The algorithm is numerically very stable
- http://en.wikipedia.org/wiki/Singular_value_decomposition#Calculating_the_SVD

THE book for matrix computations

Golub, Gene; Van Loan, Charles (2012) Matrix Computations, 4th edition.

Linear Regression

Linear model

- $\boldsymbol{y} = X\beta + \epsilon$, where X is $n \times p$ matrix
- Under normality assumption, $y_i \sim N(X_i\beta, \sigma^2)$.

Key inferences under linear model

- Effect size : $\hat{\beta} = (X^T X)^{-1} X^T \boldsymbol{y}$
- Residual variance : $\widehat{\sigma^2} = (\boldsymbol{y} - X\hat{\beta})^T(\boldsymbol{y} - X\hat{\beta})/(n-p)$
- Variance/SE of $\hat{\beta}$: $\widehat{\mathrm{Var}}(\hat{\beta}) = \widehat{\sigma^2}(X^T X)^{-1}$
- p-value for testing $H_0 : \beta_i = 0$ or $H_o : R\beta = 0$.

Hands-on Session

Visit https://bit.ly/615top03r

- Implementing linear regression with matrix operations

A case for simple linear regression

A simpler model

- $\boldsymbol{y} = \beta_0 + \boldsymbol{x}\beta_1 + \epsilon$
- $X = [1 \;\; \boldsymbol{x}]$, $\beta = [\beta_0 \;\; \beta_1]^T$.

Question of interest

Can we leverage this simplicity to make a faster inference?

A faster inference with simple linear model

Ingredients for simplification

- $\sigma_y^2 = (\mathbf{y} - \overline{y})^T(\mathbf{y} - \overline{y})/(n-1)$
- $\sigma_x^2 = (\mathbf{x} - \overline{x})^T(\mathbf{x} - \overline{x})/(n-1)$
- $\sigma_{xy} = (\mathbf{x} - \overline{x})^T(\mathbf{y} - \overline{y})/(n-1)$
- $\rho_{xy} = \sigma_{xy}/\sqrt{\sigma_x^2\sigma_y^2}$.

Making faster inferences

- $\hat{\beta}_1 = \rho_{xy}\sqrt{\sigma_y^2/\sigma_x^2}$
- $\mathrm{SE}(\hat{\beta}_1) = \sqrt{\sigma_y^2(1-\rho_{xy}^2)/(n-2)/\sigma_x^2}$
- $t = \rho_{xy}\sqrt{(n-2)/(1-\rho_{xy}^2)}$ follows t-distribution with d.f. $n-2$

Hands-on Session

Visit https://bit.ly/615top03r

- Computational efficiency of linear regression
- A faster implementation for simple linear regression

Least square estimates using matrix decompositions

Suppose that we have the following matrix decompositions already computed:

- SVD: $X = UDV^T$
- QR: $X = QR$
- Cholesky: $A = X^T X = U^T U$

How can we solve linear system $\boldsymbol{y} = X\boldsymbol{\beta} + \epsilon$ efficiently?

Least square estimates using SVD

Note that $U^T U = V^T V = I$, and $(VDV^T)^{-1} = VD^{-1}V^T$.

$$
\begin{aligned}
X &= UDV^T \\
\hat{\boldsymbol{\beta}} &= (X^T X)^{-1} X^T \mathbf{y} \\
&= (VDU^T UDV^T)^{-1} VDU^T \mathbf{y} \\
&= (VD^2 V^T)^{-1} VDU^T \mathbf{y} \\
&= VD^{-2} V^T VDU^T \mathbf{y} \\
&= VD^{-1} U^T \mathbf{y}
\end{aligned}
$$

Would this approach numerically more stable than the naive approach?

Least square estimates using QR decomposition

$$
\begin{aligned}
X &= QR \\
(X^T X)\hat{\boldsymbol{\beta}} &= X^T \mathbf{y} \\
R^T Q^T Q R \hat{\boldsymbol{\beta}} &= R^T Q^T \mathbf{y} \\
R\hat{\boldsymbol{\beta}} &= Q^T \mathbf{y}
\end{aligned}
$$

R is an upper-triangular matrix, so obtaining $\hat{\boldsymbol{\beta}}$ is straightforward.

Least square estimates using Cholesky decomposition

First, define A and $\boldsymbol{b}$ as follows:

$$\begin{aligned} A &= X^T X = U^T U \\ \boldsymbol{b} &= X^T \boldsymbol{y} \end{aligned}$$

Then, we have

$$U^T U \hat{\boldsymbol{\beta}} = X^T X \hat{\boldsymbol{\beta}} = \boldsymbol{b}$$

Now, solve the lower/upper triangular systems sequentially:

1. $U^T \boldsymbol{z} = \boldsymbol{b}$
2. $U \hat{\boldsymbol{\beta}} = \boldsymbol{z}$

Hands-on Session

Visit https://bit.ly/615top03r

- Linear regression with SVD
- Linear regression with QR decomposition
- Linear regression with Cholesky decomposition
- Evaluation of different methods

Discussion : Matrix Computation

- Make a group of two or three students.
- Review the linear regression results of different matrix decomposition methods
  - Which method is the fastest and why?
  - What are the advantages and disadvantages of each method?
  - Can you describe specific case where one method is more recommended than others?
- Five minutes
- After discussion, you may raise up your hand and ask your questions directly.

Topic Overview

1. Algorithms for matrix computation
2. Matrix decomposition and solving linear systems
3. Sparse matrices

Sparse Matrices

- **Sparse matrix**: contain mostly zero values
- **Dense matrix**: most of the values are non-zero
- **Large sparse matrices** are common in many applications
  - Text mining and natural language processing
  - Spatial epidemiology
  - Genetics
  - Genomics
  - Imaging
- Computationally expensive to represent and work with sparse matrices as though they are dense
- Much improvement in performance can be achieved by using representations and operations that specifically handle the matrix sparsity.

Sparsity

- **Sparsity** of a matrix: the number of zero values in the matrix divided by the total number of elements in the matrix.

$$\text{Sparsity} = \frac{\text{Number of zeros}}{\text{Total number of elements}}$$

- **Examples**:
  - A 2×5 matrix:

$$\boldsymbol{A} = \begin{pmatrix} 0 & -1 & 1 & 0 & 0 \\ -1 & -1 & 0 & 0 & -2 \end{pmatrix}$$

    The sparsity of $\boldsymbol{A}$ is 0.5.
  - A banded matrix $\boldsymbol{A} = (a_{i,j})_{n \times n}$:

$$a_{i,j} = \begin{cases} |i-j| & |i-j| \leq 1 \\ 0 & |i-j| > 1 \end{cases}$$

    The sparsity of $\boldsymbol{A}$ is ?

Space complexity

- Large matrix requires a lot of memory, but if the matrix is sparse, we can save memory.
- Examples for large sparse matrices:
  - A word occurrence matrix for words in one book against all known words in English ($\approx$ 273,000).
  - A brain functional connectivity matrix for all voxels in the human brain ($\approx$ 200,000).
  - A link matrix for all of websites ($\approx$ 1 billion).
- Memory cost for an $m \times n$ matrix:
  - Dense matrix: $O(mn)$.
  - Sparse matrix with q non-zero elements: $O(q)$, where $q \leq mn$.

Working with Sparse Matrices

- Use a special data structure to represent the sparse matrix.
- The zero values can be ignored and only the data or non-zero values in the sparse matrix need to be stored or acted upon.
- Typical methods to construct a sparse matrix $\boldsymbol{A} = (a_{i,j})_{m \times n}$.
  - **Dictionary of Keys**: A dictionary is used where a row and column index is mapped to a value. Let $\mathcal{S} = \{(i,j) : a_{i,j} \neq 0\}$,

$$f(i,j) \rightarrow \begin{cases} 0 & (i,j) \notin \mathcal{S} \\ a_{i,j} & (i,j) \in \mathcal{S} \end{cases} .$$

  - **List of Lists**: Each row of the matrix is stored as a list, with each sublist containing the column index and the value.

$$\{r(i) : a_{i,j} \neq 0, \text{for at least one } j\}, \quad r(i) = \{(j, a_{i,j}) : a_{i,j} \neq 0\}.$$

  - **Coordinate List (COO)**: A list of triplets is stored with each triplet containing the row index, column index, and the value.

$$\{(i, j, a_{i,j}) : a_{i,j} \neq 0\}.$$

Compressed row oriented representation

- Compressed row oriented representation, also known as compressed sparse row, or CSR.
- Instead of holding the row of each non-zero entry, the row vector holds the locations in the column vector where a row is increased.
- Example:

$$\boldsymbol{A} = (a_{i,j}) = \begin{pmatrix} 0 & -1 & 1 & 0 & 0 \\ -1 & -1 & 0 & 0 & -2 \end{pmatrix}$$

$$a_{i,c_k} = v_k; \text{ for } r_i + 1 \leq k \leq r_{i+1}, 1 \leq i \leq m, \quad r_{m+1} = |\boldsymbol{v}|$$

	Non-zero index k	1	2	3	4	5
	Row index i	1	2	-	-	-
$\boldsymbol{r} = (r_i)$	Row pointer (0-based)	0	2	5	-	-
$\boldsymbol{c} = (c_k)$	Column indices (0-based)	1	2	0	1	4
$\boldsymbol{v} = (v_k)$	Non-zero values	-1	1	-1	-1	-2

Compressed column oriented representation

- Compressed column oriented representation, also known as compressed sparse column, or CSC.
- The column vector holds the locations in the row vector where a column is increased.
- Example:

$$\boldsymbol{A} = (a_{i,j}) = \begin{pmatrix} 0 & -1 & 1 & 0 & 0 \\ -1 & -1 & 0 & 0 & -2 \end{pmatrix}$$

$$a_{r_k,j} = v_k; \text{ for } c_j + 1 \leq k \leq c_{j+1}, 1 \leq j \leq n, \qquad c_{n+1} = |\mathbf{v}|.$$

	Non-zero index k	1	2	3	4	5	
	Column index j	1	2	3	4	5	6
$\boldsymbol{c} = (c_j)$	Column pointer (0-based)	0	1	3	4	4	5
$\boldsymbol{r} = (r_k)$	Row indices (0-based)	1	0	1	0	1	
$\mathbf{v} = (v_k)$	Non-zero values	-1	-1	-1	1	-2	

Triplet representation

- Triplet representation, holds the locations of row and column indices of nonzero elements.
- Example:

$$\boldsymbol{A} = (a_{i,j}) = \begin{pmatrix} 0 & -1 & 1 & 0 & 0 \\ -1 & -1 & 0 & 0 & -2 \end{pmatrix}$$

No particular order of indices is required.
Easiest-to-understand, but could be less efficient than other formats.

	Non-zero index k	1	2	3	4	5
	Column index j	1	2	3	4	5
$\boldsymbol{c} = (r_k)$	Row indices (0-based)	1	0	1	0	1
$\boldsymbol{r} = (c_k)$	Column indices (0-based)	0	1	1	2	4
$\boldsymbol{v} = (v_k)$	Non-zero values	-1	-1	-1	1	-2

The "Matrix" Package

- The Matrix package provides facilities to deal with dense and sparse matrices.
- There are provisions to provide integer and complex (stored as double precision complex) matrices.
- The sparse matrix classes include:
  - `TsparseMatrix`: a virtual class of the various sparse matrices in triplet representation.
  - `CsparseMatrix`: a virtual class of the various sparse matrices in CSC representation.
  - `RsparseMatrix`: a virtual class of the various sparse matrices in CSR representation.

Double precision sparse matrix classes

- For matrices of real numbers, stored in double precision, the Matrix package provides the following (non virtual) classes:
  - `dgTMatrix`: a general sparse matrix of doubles, in triplet representation.
  - `dgCMatrix`: a general sparse matrix of doubles, in CSC representation.
  - `dsCMatrix`: a symmetric sparse matrix of doubles, in CSC representation.
  - `dtCMatrix`: a triangular sparse matrix of doubles, in CSC representation.

Hands-on Session

Visit https://bit.ly/615top03r

- Using `Matrix` R package to handle sparse matrices
- Memory cost for sparse matrices.
- Compute cost for sparse matrices
- Solving linear systems with sparse matrix
- Real-world examples of sparse matrices

Discussion

- Consider a linear regression model, for $i = 1, \ldots, n$,

$$y_i = \beta_0 + \sum_{j=1}^{p} \beta_j X_{i,j} + \epsilon_i, \quad \mathrm{E}(\epsilon_i) = 0, \quad \mathrm{Var}(\epsilon_i) = \sigma^2.$$

  where $n = 10^6$, $p = 10^3$, and $X_{i,j} \in \{0, 1, 2, \ldots, 10\}$. Suppose we know that

$$\sum_{i=1}^{n} \sum_{j=1}^{n} I(X_{i,j} \neq 3) \leq 10^3$$

- Design and implement an algorithm in R to efficiently estimate the β_j, for $j = 0, \ldots, p$?