STAT440: Computational Methods
Assignment 1

Ding Li

Due: May 30, 2024

Question 2: Regression with Gaussian Markov Random Fields

Part (a)

Show $\text{Cov}(\hat{\beta}) = \sigma^2(X^TQX)^{-1}$.

Solution:

Given the assumption that $Y \sim N(X\beta, \sigma^2\Sigma) = N(X\beta, \sigma^2Q^{-1}) = N(X\beta, \sigma^2(D^TD)^{-1})$, where $\Sigma, Q \in \mathbb{R}^{n\times n}$, $X \in \mathbb{R}^{n\times p}$, $\beta \in \mathbb{R}^p$, $D \in \mathbb{R}^{n\times n}$, we know that by the property of linear mappings of a multivariate Gaussian random variable Y,

$$\begin{aligned}\text{Cov}(DY) &= D\text{Cov}(Y)D^T \\ &= D\sigma^2(D^TD)^{-1}D^T \\ &= \sigma^2D(D^TD)^{-1}D^T\end{aligned}$$

Then, note that the matrix $Q = D^TD$ is invertible by the assumption, we know that $n \geq \text{rank}(D) = \min\{\text{rank}(D), \text{rank}(D^T)\} \geq \text{rank}(D^TD) = n$, which means that D^{-1} exists.

Thus, the covariance matrix of DY can be further simplified as:

$$\begin{aligned}\text{Cov}(DY) &= \sigma^2D(D^TD)^{-1}D^T \\ &= \sigma^2D(D^{-1}D^{-T})D^T \\ &= \sigma^2(DD^{-1})(D^{-T}D^T) \\ &= \sigma^2I\end{aligned}$$

, which means that we can solve an ordinary least square problem (OLS) for $DY \sim N(DX\beta, \sigma^2I)$ with the i.i.d. residuals.

Following any OSL problem solving, the analytic solution for the maximum likelihood estimate for β is:

$$\hat{\beta} = \underset{\beta\in\mathbb{R}^p}{\text{argmin}}\|DY - DX\beta\|_2^2$$

We solve it by taking the derivative of the loss function and setting it to 0:

$$\begin{aligned}-2X^TD^T(DY - DX\beta) &= 0 \\ X^TD^T(DY - DX\beta) &= 0 \\ X^TD^TDX\beta &= X^TD^TDY \\ X^TQX\beta &= X^TQY\end{aligned}$$

Assuming that X^TQX is positive definite and thus invertible,

$$\hat{\beta} = (X^TQX)^{-1}X^TQY$$

is the stationary point of the loss function. Since the loss function is a convex function, any stationary point is a global minimizer of it, so $\hat{\beta} = (X^TQX)^{-1}X^TQY$ minimizes the loss function. Thus, it's the maximum likelihood estimate for the true parameter β.

Again, by the property of linear mappings of a multivariate Gaussian random variable Y,

$$\begin{aligned}
\text{Cov}(\hat{\beta}) &= \text{Cov}((X^TQX)^{-1}X^TQY) \\
&= (X^TQX)^{-1}X^TQ\text{Cov}(Y)[(X^TQX)^{-1}X^TQ]^T \\
&= (X^TQX)^{-1}X^TQ\sigma^2Q^{-1}[(X^TQX)^{-1}X^TQ]^T \\
&= \sigma^2(X^TQX)^{-1}X^T(QQ^{-1})[Q^TX(X^TQX)^{-T}] \\
&= \sigma^2(X^TQX)^{-1}(X^TQ^TX)(X^TQX)^{-T}
\end{aligned}$$

Since $Q^T = (D^TD)^T = D^TD = Q$,

$$\begin{aligned}
\text{Cov}(\hat{\beta}) &= \sigma^2(X^TQX)^{-1}(X^TQ^TX)(X^TQX)^{-T} \\
&= \sigma^2(X^TQX)^{-1}(X^TQX)(X^TQX)^{-T} \\
&= \sigma^2[(X^TQX)^{-1}(X^TQX)](X^TQX)^{-T} \\
&= \sigma^2(X^TQX)^{-T}
\end{aligned}$$

Finally, since $(X^TQX)^{-T} = [(X^TQX)^T]^{-1} = (X^TQ^TX)^{-1} = (X^TQX)^{-1}$, we have

$$\text{Cov}(\hat{\beta}) = \sigma^2(X^TQX)^{-1}$$

Part (b)

In this question, we compare the running time for computing the inverse of X^TQX with Q stored in a dense matrix or a sparse matrix.

According to Fig.1, we see that in any case, storing Q in a sparse matrix always makes the computing faster than storing it in a dense matrix. The computing algorithm using sparse matrix can successfully complete in 1000 nanoseconds even when the number of covariates reaches 150 and the sample size reaches 1000, whereas the one with dense matrix can't.

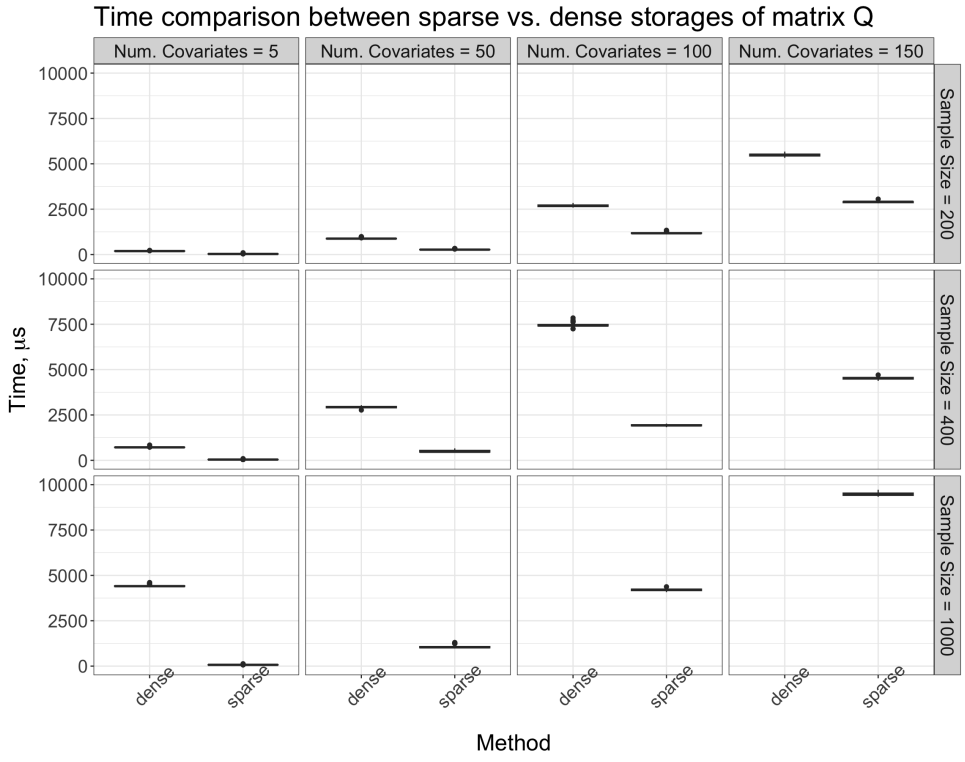

Figure 1: Time comparison between design matrix inversion through sparse matrix or dense matrix, with datasets of sizes n = 200, 400, 1000, the number of covariates p = 5, 50, 100, 150.

To be clearer, we hold the number of covariates constant and just expand the sample size and generate Fig.2. According to Fig.2, when the number of covariates holds constant, the performance for the sparse-matrix implementation doesn't show any change with respect to the increase of sample size, whereas the running time for the dense-matrix implementation rises about 10000 nanoseconds as the sample size expands from 100 to 1500.

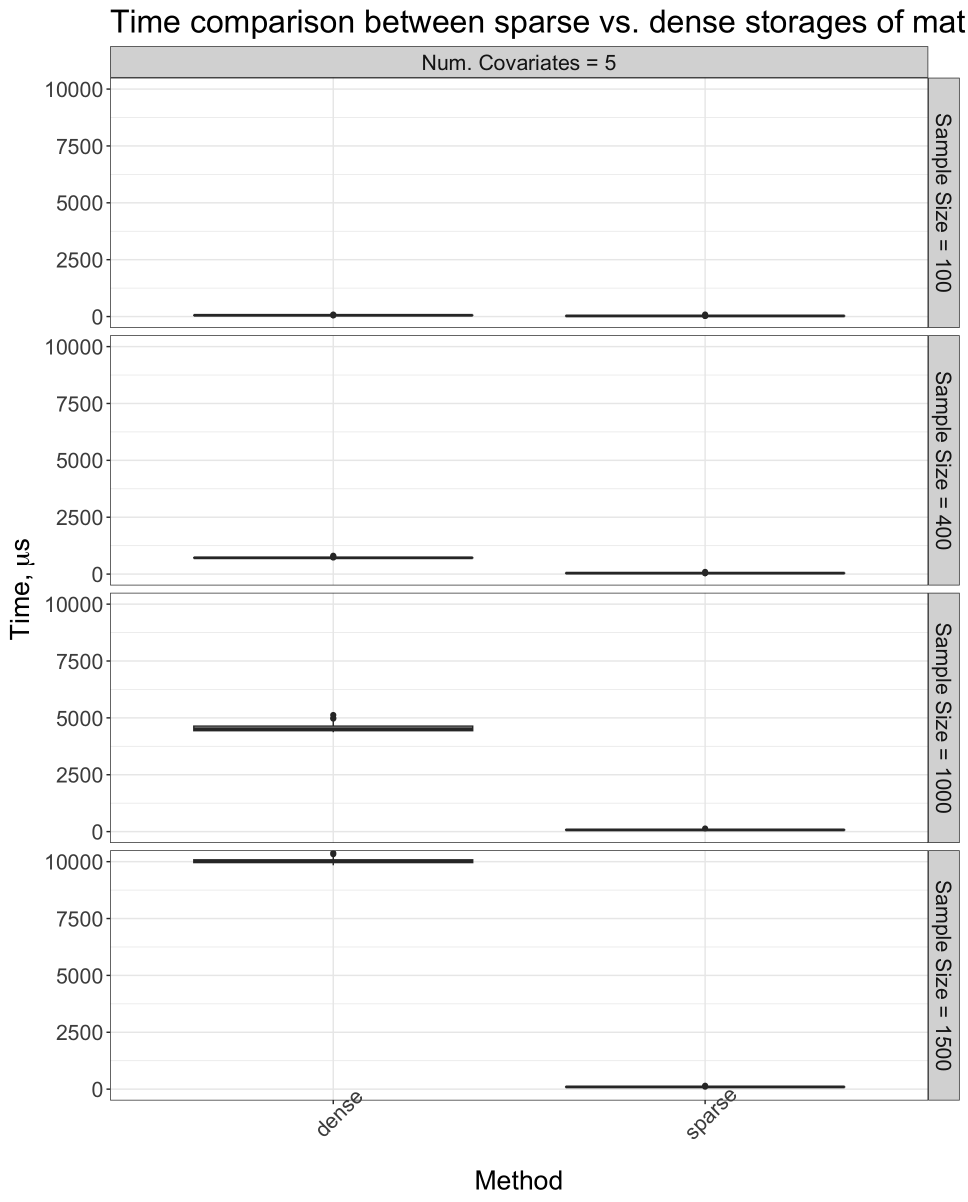

Figure 2: Time comparison between design matrix inversion through sparse matrix or dense matrix, with datasets of sizes n = 100, 400, 1000, 1500, the number of covariates p = 5.

Then, we try to hold the sample size as a constant and perturb the number of covariates, which is demonstrated in Fig.3. According to Fig.3, when the sample size is fixed to 500, the running time for both algorithms increases as the number of covariates increases. Nonetheless, the performance of the dense-matrix implementation is affected significantly worse than the sparse-matrix implementation.

In summary, the inverse computing algorithm implemented using sparse matrix runs much faster than the one implemented in dense matrix. The increase of the number of covariates and sample size all worsen the performance of the dense-matrix implementation significantly. The expansion in sample size has a very slight impact on the performance of the sparse-matrix implementation, whereas the increase in the number of covariates affects more. But the impacts are all much smaller than the ones on the dense-matrix implementation.

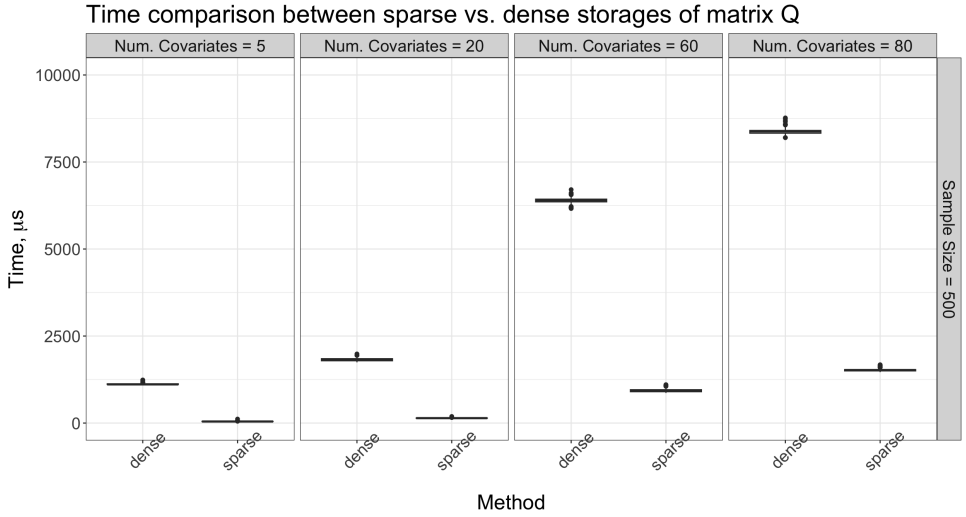

Figure 3: Time comparison between design matrix inversion through sparse matrix or dense matrix, with datasets of sizes n = 500, the number of covariates p = 5, 20, 60, 80.

Part (c)

Show $\text{Cov}(\hat{y}) = \sigma^2 X(X^T QX)^{-1} X^T$.

Solution: Given that $\text{Cov}(\hat{\beta}) = \sigma^2 (X^T QX)^{-1}$ as we derived in part (a), and $\hat{\beta}$ follows a multivariate normal distribution as a linear mapping of another normally distributed random variable Y, we know that by the properties of normal distributions:

$$\begin{aligned}\text{Cov}(\hat{y}) &= \text{Cov}(X\hat{\beta}) \\ &= X\text{Cov}(\hat{\beta})X^T \\ &= X\sigma^2(X^T QX)^{-1}X^T \\ &= \sigma^2 X(X^T QX)^{-1}X^T\end{aligned}$$

, since $\hat{y}$ is also a linear mapping of $\hat{\beta}$ that is a random variable following a multivariate normal distribution.

Part (d)

Let $(X^T QX)^{-1} = LL^T$ be the Cholesky decomposition. Show that

$$\text{Cov}(\hat{y})_{ii} = \sum_{j=1}^{p} \left((L^{-1}X^T)_{ji}\right)^2$$

Answer: Given the Cholesky decomposition of $X^T QX = LL^T$, and that $\text{Cov}(\hat{y}) = \sigma^2 X(X^T QX)^{-1}X^T$ as we derived in part (c), we have:

$$\begin{aligned}\text{Cov}(\hat{y}) &= \sigma^2 X(LL^T)^{-1}X^T \\ &= \sigma^2 XL^{-T}L^{-1}X^T \\ &= \sigma^2 (L^{-1}X^T)^T(L^{-1}X^T)\end{aligned}$$

when we plug in the first equation to the second one.

Then, to get the diagonal entries $\text{Cov}(\hat{y})_{ii}$ for any i, we follow the definition of matrix multiplications, which is the inner product between the (transpose of the) i-th row of matrix $(L^{-1}X^T)^T$ and the i-th column of the matrix $(L^{-1}X^T)$. Note that the two matrices are transposed of each other, so $\text{Cov}(\hat{y})_{ii}$ is just the inner product of the i-th column of the matrix $(L^{-1}X^T)$ and itself, i.e., the Euclidean norm of the i-th column of the matrix $(L^{-1}X^T)$. Thus, we know that:

$$\text{Cov}(\hat{y})_{ii}/\sigma^2 = \sum_{j=1}^{p} \{(L^{-1}X)_{ji}\}^2$$

which is exactly the Euclidean norm of the i-th column of the matrix $(L^{-1}X^T)$. Finally, we arrange to get

$$\text{Cov}(\hat{y})_{ii} = \sigma^2 \sum_{j=1}^{p} \{(L^{-1}X)_{ji}\}^2$$

Part (e)

In this question, we devise an algorithm that can compute the diagonal entries of $\text{Cov}(\hat{y})$ without having to store or calculate $n \times n$ matrices.
Let $x_1, \ldots, x_n \in \mathbb{R}^p$ be the data records in X, i.e., $X = [x_1, \ldots, x_n]^T$. Then, we know that:

$$L^{-1}X = L^{-1}[x_1, \ldots, x_n] = [L^{-1}x_1, \ldots, L^{-1}x_n]$$

For any $i = 1, 2, \ldots, n$, define $y_i = L^{-1}x_i$, then the i-th column of $L^{-1}X$ is just y_i. Hence, according to what I claimed in Q2d, $\text{Cov}(\hat{y})_{ii}$ is just the inner product of y_i.
We solve the linear system equation $Ly_i = x_i$ for $y_i, \forall i = 1, 2, \ldots, n$, which is convenient through forward substitution since L is lower triangular.
Overall, the algorithm is:

1. Compute $DX \in R^{n \times p}$ by keeping the first row unchanged, and then for each row $i = 2, 3, \ldots, n$, assign $x_i^T \leftarrow x_i^T - x_{i-1}^T$, which is an in-place operation that doesn't need extra spaces for matrix storage. The calculation itself only involves around $O(nd)$ entries, which is much smaller than $n \times n$. The operation should have been handled well in the sparse matrix multiplication operations in R, so I don't have to really implement one by hand.
2. Compute X^TQX by using $(DX)^TDX$. Note that $DX \in \mathbb{R}^{n \times p}$ and the matrix itself is $X^TQX \in \mathbb{R}^{p \times p}$, so the computation doesn't include any storage or operations over a large $n \times n$ matrix.
3. Pre-compute L through Cholesky decomposition on $X^TQX = LL^T$.
4. For each $i = 1, 2, \ldots, n$, solve $Ly_i = x_i$ using forward substitution, then finally obtain $\text{Cov}(\hat{y})_{ii} = y_i^T y_i$.

Part (f)

Based on the hardware performance of the machine I used to run the algorithms, I chose to run each of:

1. Direct computation and dense storage of $X(X^TQX)^{-1}X^T$ with Q stored densely, which is called "dense_direct" in the result plot;
2. Computation using the algorithm I devised in part (e) with Q stored densely, which is called "dense_mine" in the result plot;
3. Direct computation and dense storage of $X(X^TQX)^{-1}X^T$ with Q stored sparsely, which is called "sparse_direct" in the result plot;
4. Computation using the algorithm I devised in part (e) with Q stored densely, which is called "sparse_mine" in the result plot;

1000 times, with different input dimensions of $X \in \mathbb{R}^{n \times p}$ with sizes setting to $n = 200, 500$ and $p = 1, 5, 50$.

The result is shown below in Fig.4:

According to the result plot in Fig.4, we can see that when both the number of covariates and the sample size is small, the performances of different strategies are approximately equal. The sparse implementation of the direct method is the fastest with a tiny advantage, and other methods spend almost an equal amount of time. The case for the same sample size but adding the number of covariates to 5 doesn't change much. When we keep the same sample size but level up the number of covariates to 50, we can see a very significant performance difference: My devised methods outperform direct methods, no matter if they use sparse or dense matrix implementations. That being said, even though my devised method is implemented using dense matrices, it's still better than the direct method implemented using sparse matrices. Of course, the sparse-matrix implementation of the direct method outperforms the dense-matrix implemented direct method with observable advantage. But for the devised algorithm, implementing it in sparse matrices doesn't have obvious improvement upon the one implemented using dense matrices. For only one covariate, the dense implementation of the direct method is obviously

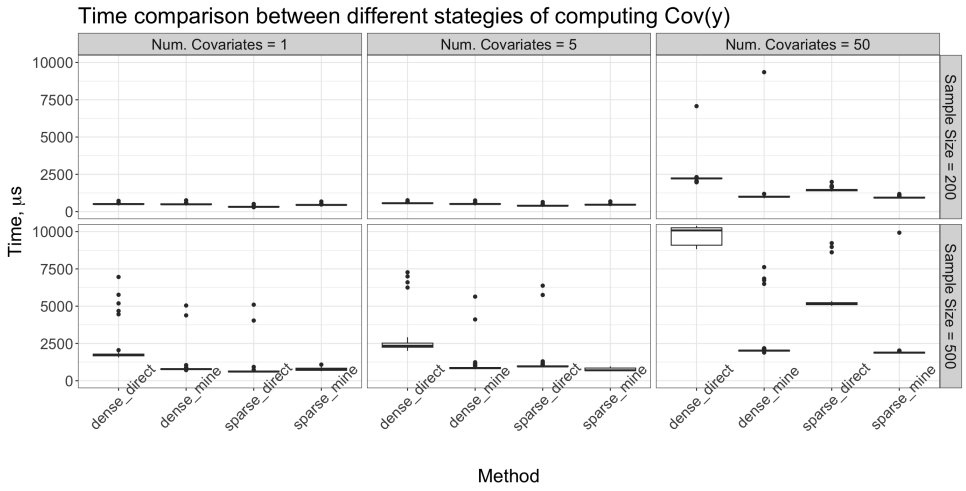

Figure 4: Time comparison between different strategies of computing covariance matrices for Markov Random Field regression predictions, with datasets of sizes $n = 200, 500$, $p = 1, 5, 50$.

worse than the one implemented in sparse matrices and my devised method as the sample size goes larger. When the number of covariates increases to 5, the dense implementation of the direct method is even worse. And we can also see that the sparse implementation of the direct method starts to have an observable disadvantage compared to any of my devised algorithms. If we expand the number of covariates to 50, then we can see that the direct methods are much slower than the devised ones, with the sparse implementation much better. The devised algorithms are over 2 times faster than the direct methods. But the difference between the sparse implementation and dense implementation for the devised algorithm is not significant.

To further compare the performances between the dense/sparse-matrix implementations of my devised algorithm, we need to expand the sample size and the number of covariates. In Fig.5, the number of covariates holds constant and is small, and we perturb the sample size. We can see that when the sample size increases, the performance difference between the densely implemented direct method and the other three gets larger and larger quickly, whereas the sparse/dense versions of the devised algorithm doesn't change significantly as we fix the number of covariates constant.

In summary, the direct methods only have a very slight advantage in speed against my devised algorithm when the sample size and the number of covariates is extremely small. That is because when the size of the dataset is small, doing matrix decomposition consumes more than the performance gain. Then, the in-group performance comparison shows that, for the same algorithm, a sparse-matrix implementation is always better than a dense-matrix implementation, but the direct algorithm benefits more from using sparse matrices. That is because when using the direct method, sparse matrices can avoid bad matrix computing, whereas my devised algorithm doesn't include any. The between-group comparison indicates that my devised algorithm is much more advantageous when the sample size is incredibly large. The devised algorithm doesn't take much more time when the sample size expands, whereas the direct methods highly depend on the sample size. That is because the direct methods contain computing an $n \times n$ matrix, which quadratically increases the running time when n increases, whereas the devised algorithm doesn't contain any. Also, from Fig.6, we can observe that the devised algorithm preserves much better performance than the direct methods when the number of covariates increases, but the preservation of performance is not as beneficial as we increase the sample size. This is because the devised methods exploit the lower triangular matrix to do easy substitution, whereas the performance of the direct methods collapses significantly when they have to calculate the inverse of a $p \times p$ matrix, when p becomes larger. Nonetheless, the devised methods still suffer from the increase of the number of covariates, since their performance depend on p, especially when they calculate the Cholesky decomposition of $X^T Q X$.

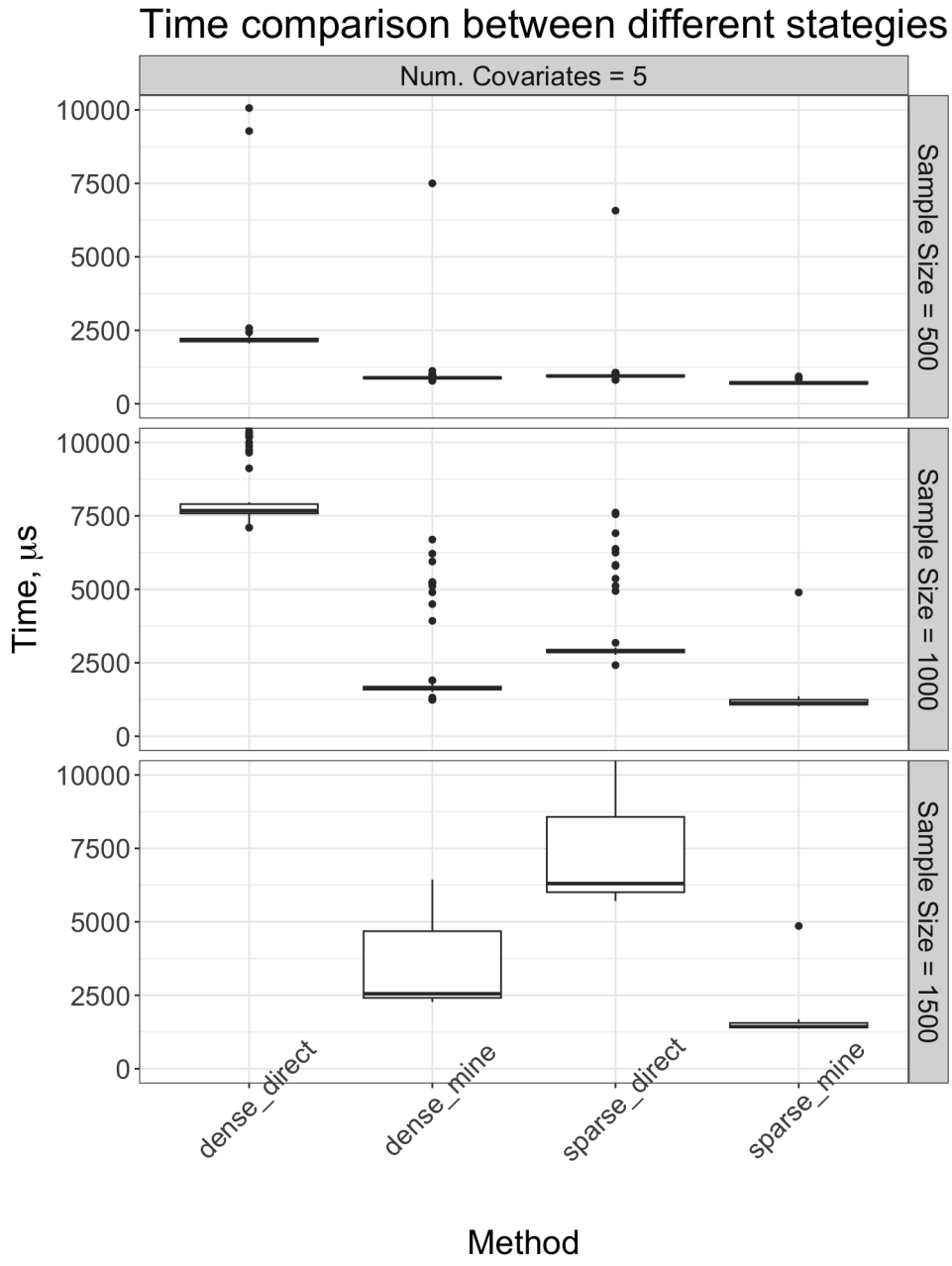

Figure 5: Time comparison between different strategies of computing covariance matrices for Markov Random Field regression predictions, with datasets of sizes n = 500, 1000, 1500, p = 5.

Question 3: Paper Review

Background

Due to my lack of background in statistics, I would like to summarize the background of the series problems the paper tries to solve. Gaussian random field is widely used in spatial statistics, where the spatial units are jointly Gaussian but not independent from each other. Hence, the covariance matrix involves computing a sample-size-large square matrix for its design matrix (e.g. The conclusion in Q2 part a). When the sample size is large, which is usually the case in reality, the computation is expensive. As such, many downstream tasks, such as sampling from the field, or computing conditional densities, becomes impossible for very large sample size. A Gaussian Markov random field (GMRF) is a type of Gaussian field where the conditional distribution of a spatial unit on all other spatial units is equal to the one on only its neighbors. The paper contributes to tackling the computation challenge by exploiting the Markov property of GMRFs through computational linear algebra techniques.

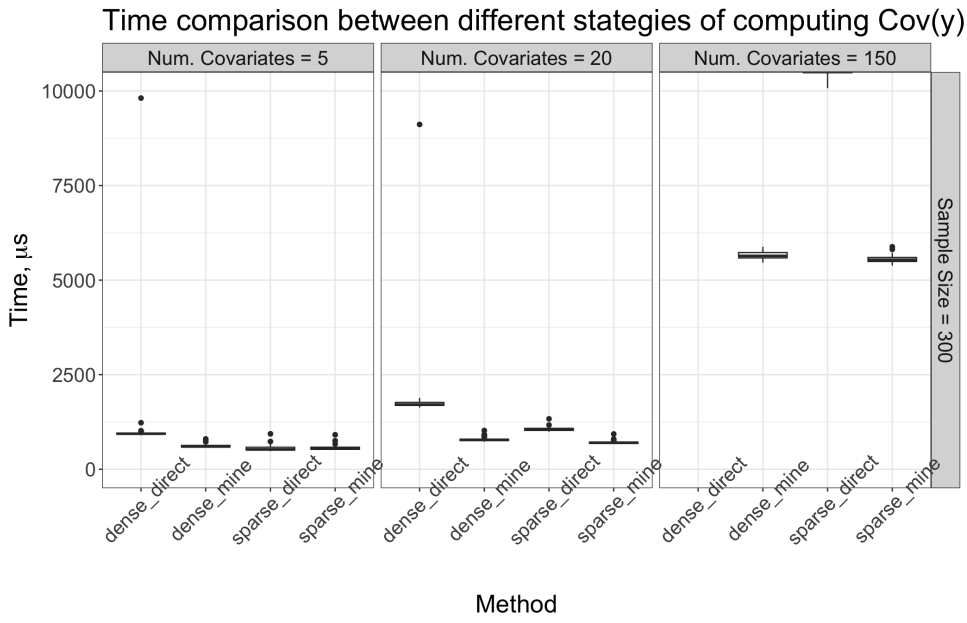

Figure 6: Time comparison between different strategies of computing covariance matrices for Markov Random Field regression predictions, with datasets of sizes n = 300, p = 5, 20, 150.

Paper contribution summary

The paper originally proposed taking advantage of the Markov property GMRFs to do fast computing for the precision matrix. The Markov property means that the precision matrix should be sparse. Intuitively, since not every pair of observations in the field are conditionally important to each other, the computation cost should not be the square of the entire sample size, but change one of the dimensions of the computing cost into the size of the neighborhood.

The algorithm defines a graph-structured correlation function for the GMRF, which gives the flexibility of how we define a local neighbourhood the strengths of their correlations. Specifically, the connected nodes in the graph are neighbors, where the entries in the precision matrix are non-zero.

Overall, the paper links the locality of Markov property to the sparsity of its precision matrix. Then, it employs graph theory to convert the precision matrix to a narrow-band matrix. By using Cholesky decomposition, the precision matrix can be pre-computed as a lower-triangular matrix that also has the narrow band. Finally, the applications, such as (replicated) sampling and likelihood computing, are handled by using the pre-computed lower-triangular matrix (i.e., the Cholesky decomposition of the precision matrix) through efficient forward/backward substitution.

The achievement of the algorithm is that it speeds up and saves memories for the sampling and likelihood computation in such aspects:

1. Pre-computing the Cholesky decomposition of the precision matrix once to reuse it infinitely many times for sampling and likelihood computing; 2. By narrowing the band of the precision matrix (and thus its Cholesky decomposition), the matrix multiplication and inversion costs (in terms of speed and memory use) in the sampling and likelihood computing are reduced from the sample-size scale to neighborhood-size scale. 3. Through divide-and-conquer strategy, computing and storing the band Cholesky decomposition with GMRFs defined on very large graphs can be split and solved into smaller tasks on a limited memory. 4. Has the potential be more widely extended to other Gaussian fields defined with more general Gaussian correlation functions, since they were proven to be approximated by GMRFs with some specific neighborhood definition within some errors.

Relationships between Section 2 and Question 2

The subject of the paper overlaps a lot with Section 2 of the lecture note and question 2 of the assignment. Section 2 of the lecture shows how computing costs explode when we introduce a correlation assumption. The paper gives a perfect example of how a Gaussian random field introduces such a cost. Question 2 is a simple example of a GMRF defined in the paper (except that the mean is not 0 as mentioned in the paper), with the neighborhood defined by a previous and a latter observation along a line. The model in

question 2 is easy because the precision matrix is forced to already be band-1 without needing to do node permutation on the neighborhood graph definition. But the actual matrix computing parts still involve matrices of big sample sizes. The lecture is organized to introduce computational linear algebra methods that are used to speed up and decrease memory use by exploiting the matrix structure. The focus is on what methods to use for specific positive definite matrices that are most common in a statistical context. The paper uses Cholesky decomposition to preserve the sparse and low-band structure of the precision matrices for fast backward/forward substitutions in solving linear equation systems with fewer unknowns involved, which is a covered technique in the lecture. In question 2, the same Cholesky decomposition is applied to the design matrix to enable fast forward substitution operations. Also, the pre-computing of the Cholesky decomposition saved time and memory for repeatedly computing the diagonal entries of the prediction covariance matrix in the assignment. The technique is introduced in the lecture and employed in the paper in replicated sampling and likelihood evaluation.

Overall, the lecture, assignment, and the paper all focus on taking advantage of the locality of the Markov property to save running time and memory for computing with GMRFs. The methods all rely highly on linking the property to a typical sparse structure of the precision matrix, then employing an existing fast computational linear algebra algorithm to save time and memory. The methods can finally reduce some of the dimensions of computational costs from the large sample size to the model complexity/neighborhood size.