Improved Fast Direct Methods for Gaussian Processes

Eric Han

February 11, 2022

Introduction

For Gaussian Process, the posterior update is given by

$$\mu_{t+1}(\mathbf{x}) = \mathbf{k}_t(\mathbf{x})^\top (\mathbf{K}_t + \eta^2 \mathbf{I}_t)^{-1} \mathbf{y}_t,$$
$$\sigma_{t+1}(\mathbf{x}) = k(\mathbf{x}, \mathbf{x}) - \mathbf{k}_t(\mathbf{x})^\top (\mathbf{K}_t + \eta^2 \mathbf{I}_t)^{-1} \mathbf{k}_t(\mathbf{x}).$$

Also, the log likelihood used to select hyperparameters is given as:

$$\log p(\mathbf{y}|\mathbf{X}, \theta) = -\frac{1}{2}\mathbf{y}^\top (\mathbf{K} + \eta^2 \mathbf{I})^{-1} \mathbf{y} - \frac{1}{2}\log|\mathbf{K} + \eta^2 \mathbf{I}| - \frac{n}{2}\log 2\pi$$

For large n, the computational cost of inverting $\widehat{\mathbf{K}}_{XX} = \mathbf{K}_t + \eta^2 \mathbf{I}_t$ and finding $|\widehat{\mathbf{K}}_{XX}|$ is expensive; the traditional/direct methods for dense systems scales quadratically.

Traditionally, Cholesky decomposition is used, factoring the positive-semidefinite matrix $\widehat{\mathbf{K}}_{XX}$ into $L \times L^\top$.

Related Work

There are generally 2 approaches to address the scaling issue:

1. Exact solutions - Find the exact solution to the matrix inversion.
    1. LU decomposition
    2. Cholesky decomposition
        - Implementation: Scikit-Learn (via Numpy, Blas/Lapack)
        - Implementation: GPy (via Scipy, Blas/Lapack)
        - Implementation: GPflow (via Tensorflow)
    3. Conjugate gradients method (CG)
        1. Exact Gaussian Processes on a Million Data Points
            - Author's Implementation (GPyTorch): Exact GPs with GPU Acceleration
    4. Hierarchical Off-Diagonal Low-Rank (HODLR) matrix factorization
        1. Fast Direct Methods for Gaussian Processes
            - Author's Implementation: George
2. Approximate solutions - Trade-off accuracy to improve computational cost.
    1. Mixture-of-experts
    2. Sampling m Inducing points
    3. Random feature expansions

Exact Gaussian Processes on a Million Data Points

Key Idea: Instead of solving for $\widehat{\mathbf{K}}_{XX}^{-1}$ to find $\mathbf{x}$, we find $\mathbf{x}$ directly:

$$\mathbf{x}^* = \arg\min_{\mathbf{x}} \Big(\frac{1}{2}\mathbf{x}^\top \widehat{\mathbf{K}}_{XX}\mathbf{x} - \mathbf{x}^\top \mathbf{y} \Big)$$

- Unique minimizer $\mathbf{x}$ exist due to positive definite $\widehat{\mathbf{K}}_{XX}$
- Iterative algorithm with matrix-vector multiplication
- Iterations can be speed up with preconditioning
- Able to decide tolerence of solution $\epsilon = ||\widehat{\mathbf{K}}_{XX}\mathbf{x}^* - \mathbf{y}||/||\mathbf{y}||$

- Similar to gradient descent, the difference is that CG enforces that the search direction $\mathbf{p}_i$ is conjugate to each other.

Note: In the absence of round-off error, it converge to exact solution after n steps.

Fast Direct Methods for Gaussian Processes

Key Idea: Assume that $\widehat{\mathbf{K}}_{XX}$ is Hierarchical Off-Diagonal Low-Rank (HODLR) matrix, and can be factored

$$\widehat{\mathbf{K}}_{XX} = K^{(0)} = \begin{bmatrix} K_1^{(1)} & U_1^{(1)} V_1^{(1)\top} \\ V_1^{(1)} U_1^{(1)\top} & K_2^{(1)} \end{bmatrix},$$

where the diagonal blocks are recursive HODLR decompositions

$$K_1^{(1)} = \begin{bmatrix} K_1^{(2)} & U_1^{(2)} V_1^{(2)\top} \\ V_1^{(2)} U_1^{(2)\top} & K_2^{(2)} \end{bmatrix}, \quad K_2^{(1)} = \begin{bmatrix} K_3^{(2)} & U_2^{(2)} V_2^{(2)\top} \\ V_2^{(2)} U_2^{(2)\top} & K_4^{(2)} \end{bmatrix},$$

and that $U_i^{(j)}, V_i^{(j)}$ are $\frac{n}{2^j} \times r$ matrices. We note that the off-diagonal matrices are low-rank matrices and can be approximated by $U_i^{(j)}, V_i^{(j)}$ up to a certain accuracy. The decompositions can be performed recursively up to the κ-level where $\kappa \sim \log n$; i.e. $\widehat{\mathbf{K}}_{XX} = K_\kappa K_{\kappa-1} \cdots K_0$, see below.

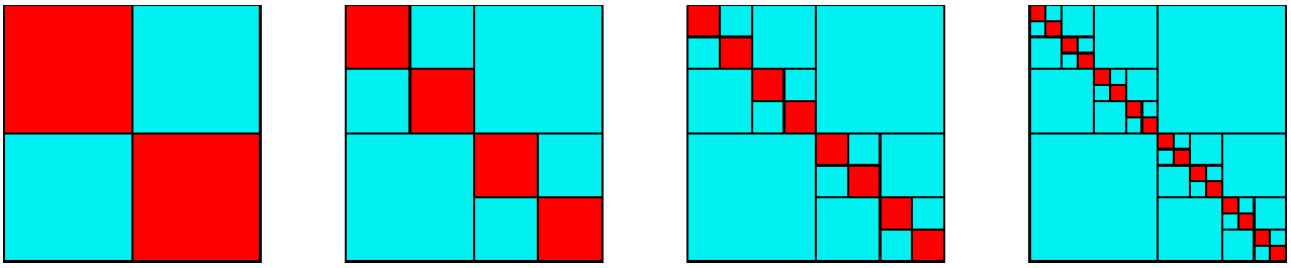

Figure 1: Pictorial representation of the HOLDR decompositons.

Process

Matrix Inversion. The summarized process to obtain the inverse (details are in the paper):

1. Computing the low-rank factorization of all off-diagonal block
2. Using these low-rank approximations to recursively factor the matrix into roughly $O(\log n)$ pieces.
3. Sherman-Morrison-Woodbury formula can be applied to the $O(\log n)$ factors to find the inverse.

The overall runtime to obtain the matrix inverse of $\widehat{\mathbf{K}}_{XX}$ is $O(n \log n)$.

Determinant Computation. Using the HOLDR factorization, the determinant can be computed directly:

$$|\widehat{\mathbf{K}}_{XX}| = |K_\kappa| \times |K_{\kappa-1}| \times \cdots \times |K_0|$$

The determinant can be computed in $O(\kappa n)$.

Methods Comparison

The current state-of-the art for fast GP Regression on large n is using CG. Here, we perform an experiment to compare the speed for which each of the methods:

1. Cholesky decomposition
   - `GPy` Implementation in GPy
   - `sklearn` Implementation in Scikit-Learn
   - `basic` Cholesky implementation in George
2. Hierarchical Off-Diagonal Low-Rank (HODLR) matrix factorization
   - `HOLDR` Implementation in George

3. Conjugate gradients method (CG)
    - `CG-CPU` Implementation in GPyTorch via CPU.
    - `CG-GPU` Implementation in GPyTorch via GPU.

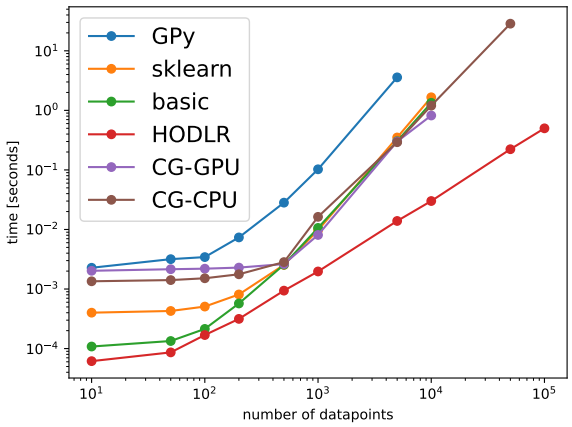

Figure 2: Comparing HOLDR vs CG on a 1D synthetic function.

Advantage

The major advantage of `HODLR` when compared to other methods is that it is very fast (also on scaling), even when compared to CG via the GPU.

Disadvantage

At first glance, there seems to be no issue with `HODLR`.

However, from Inconsistent results with HODLRSolver, we see that there are performace issues in its use. Following their example, we can construct a similar example to illustrate the poor accuracy in the computation of log likelihood ℓ; we setup the synthetic function $y = \sin(x)$ with 200 data points. We let the accurate log likelihood of the X to be ℓ^*, computed using Cholesky decomposition. We compute the log likelihood ℓ_{HODLR} using the HODLR matrix factorization without tweaking any defaults.

$$\ell^* = 31.344534039948236, \quad \ell_{\text{HODLR}} = 23.518503030261652$$

The relative difference would compute to be $\frac{|\ell^* - \ell_{\text{HODLR}}|}{\ell^*} = 24.97\%$.

This issue was discussed in passing in the paper, see below; The authors concluded that it is not an issue and presented a suggestion to use kd-tree sort to condition the matrix $\widehat{\mathbf{K}}_{XX}$. However, the authors reversed this viewpoint in Issue 128.

> We note that the authors claimed that when the data points at which the kernel to be evaluated at are not approximately uniformly distributed, the performance of the factorization may suffer, but only slightly. A higher level of compression could be obtained in the off-diagonal blocks if the hierarchical tree structure is constructed based on spatial considerations instead of point count, as is the case with some kd-tree implementations.

Metrics Used

In this section, we review the various metrics used by the various papers, in a attempt to find the best for us:

1. Exact Gaussian Processes on a Million Data Points: Dataset is split into $4:2:3$ - training, validating and testing sets. Validation is used to tune the CG parameters, the metrics are computed on test set.
    1. Accuracy - Root-Mean-Square Error on the test set, and is heavily used to discuss the performance of the method, on 12 UCI datasets.

$$RMSE = \sqrt{\frac{1}{n}\Sigma_{i=1}^{n}\left(y_i^* - y_i\right)^2}$$

2. Accuracy - Negative log-likelihood of the exact GP is shown along with the approximate methods, but it is not used to discuss at all.
3. Compute Time - Training Time, Testing Time

2. Fast Direct Methods for Gaussian Processes:
    1. Accuracy - Relative l_2 precision in the solution to a test problem $Cx = b$, where b was generated a priori from the known vector x, where C is synthetically created from a linear combination of:
        1. Gaussian covariance
        2. Multiquadric covariance matrice
        3. Exponential covariance
    2. Accuracy - Difference of the Root-Mean-Square Errors of the exact method vs HOLDR on a synthetic function $\sin(2x) + \frac{1}{8}e^x + \epsilon$.
    3. Compute Time - Assembly Time, Factor Time, Solve Time and Determinant Compute Time.

Some thoughts

1. RMSE on test set must be used for accuracy comparison.
2. We can test synthetic accuracy using $Cx = b$.
3. We need to get a notion of the computational relative floating point error for the operations that we care about.
4. What is there a theoretical error that we can achieve?

Proof of Concept

The key insight is that the the closer the kernel matrix is to an hierarchical matrix, the better the accuracy. Since `HOLDR` is fast, we want to pay some computational cost to condition the matrix. We see that we can perform some reordering operations on $X = [x_0, \cdots, x_n] \to X'$ to massage[1] the kernel matrix $\widehat{\mathbf{K}}_{XX}$ into an equlivant hierarchical matrix $\widehat{\mathbf{K}}_{X'X'}$. Here, equlivant means that the reordering operation does not change values of the posterior updates and the marginal likelihood [2].

$$\widehat{\mathbf{K}}_{XX} = \begin{array}{c} \begin{matrix} x_0 & x_1 & x_2 & x_3 \end{matrix} \\ \begin{bmatrix} 1 & 0 & 4 & 0 \\ 0 & 1 & 0 & 2 \\ 4 & 0 & 1 & 0 \\ 0 & 2 & 0 & 1 \end{bmatrix} \end{array} \begin{matrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{matrix} \to \begin{array}{c} \begin{matrix} x_0 & x_2 & x_1 & x_3 \end{matrix} \\ \begin{bmatrix} 1 & 4 & 0 & 0 \\ 4 & 1 & 0 & 0 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 2 & 1 \end{bmatrix} \end{array} \begin{matrix} x_0 \\ x_2 \\ x_1 \\ x_3 \end{matrix} = \widehat{\mathbf{K}}_{X'X'}$$

In this proof of concept, we will find the hierarchical matrix $\widehat{\mathbf{K}}_{X'X'}$ and its corresponding reordered X' for which allows us to achieve a good accuracy of the log likelihood $\log p(\mathbf{y}|\mathbf{X}, \theta)$.

We formulate the problem as a optimization problem and use Genetic Algorithm (GA) to find X', with each i-iteration candidate solution to be X_i. We run the GA with 1000 iterations and population size of 20:

- Mutation operation: Swapping of 2 points x_i, x_j where i, j are indices from different groups.
- Fitness score: Absolute difference of the accurate log likelihood and the current log likelihood $|\ell^* - \ell^{(i)}|$

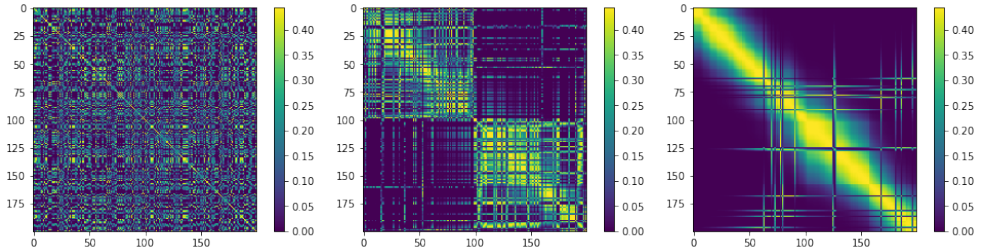

Figure 3: Heatmap of the $\widehat{\mathbf{K}}_{XX}$ matrices, showing from $X_0 \to X_{20} \to X_{1000}$.

From above, we see that the concept works, choosing a solution X_i that is incrementally closer to the better solution than X_{i-1}. We summarize the results as follows:

[1] Do we need to show that such set of operations exist?
[2] Do we need a proof?

$$\ell^{(0)} = 23.52 \rightarrow \ell^{(1000)} = 31.34 \sim \ell^* = 31.34$$

Hence, we are confident that it would be possible to perform a reordering operation on X such that its kernel matrix is a hierarchical matrix.

Genetic Algorithm

We have used the *optimal* metric in the Proof of Concept which will not be practical; we are after all aiming to achieve that as a result of our optimization, so using it in our optimization would be cheating, and not to mention not cheap computationally. Here, we will use the same settings as in the proof of concept, with the exception of the fitness function.

Sum of off-diagonal matrix

$$\widehat{\mathbf{K}}_{XX} = K^{(0)} = \begin{bmatrix} K_1^{(1)} & U_1^{(1)} V_1^{(1)\top} \\ V_1^{(1)} U_1^{(1)\top} & K_2^{(1)} \end{bmatrix},$$

Here, we are using the sum over the cells in the off-diagonal matrix $V_1^{(1)} U_1^{(1)\top}$. We aim to find a reordering such that the sum over the off-diagonal matrix is minimum.

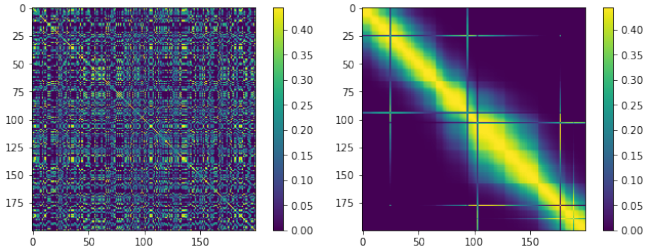

Figure 4: Heatmap of the $\widehat{\mathbf{K}}_{XX}$ matrices - left is the kernel matrix of X_0, right is after.

$$\ell^{(0)} = 23.52 \rightarrow \ell^{(\text{GA-Sum})} = 31.44, \frac{|\ell^* - \ell^{(\text{GA-Sum})}|}{\ell^*} = 0.307\%$$

Rank of off-diagonal matrix

Similar to the sum as above, instead we use the rank of the off-diagonal matrix $V_1^{(1)} U_1^{(1)\top}$. This is from the definition of hierarchical matrix.

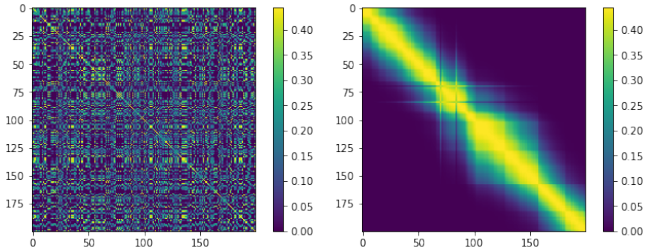

Figure 5: Heatmap of the $\widehat{\mathbf{K}}_{XX}$ matrices - left is the kernel matrix of X_0, right is after.

$$\ell^{(0)} = 23.52 \rightarrow \ell^{(\text{GA-Rank})} = 31.44, \frac{|\ell^* - \ell^{(\text{GA-Rank})}|}{\ell^*} = 0.307\%$$

We see that there is not much difference between using the rank and the sum of cells, with the exception of the increased computation needed to calculate the rank.

Sorting

KD-Tree

As a simplified example, we build a KD-Tree on X and query the tree on the closest n points to x_0.

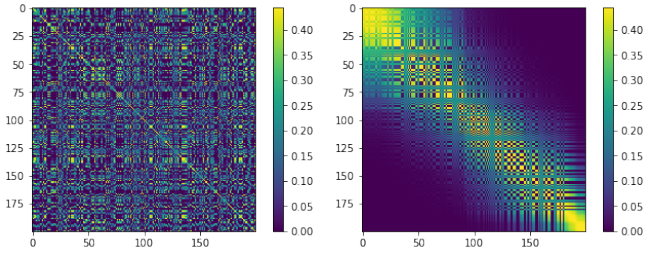

Figure 6: Heatmap of the $\widehat{\mathbf{K}}_{XX}$ matrices - left is the kernel matrix of X_0, right is after.

$$\ell^{(0)} = 23.52 \to \ell^{(\text{Sort-KD})} = 29.14, \frac{|\ell^* - \ell^{(\text{Sort-KD})}|}{\ell^*} = 7.03\%$$

We observe that this method works marginally; However, KD-Trees are unsuitable for finding nearest neighbours in high dimensions due to curse of dimensionality. We expect the performance of this method to rapidly degrade in higher dimensionlity.

Clustering

Vanilla K-Means

Here, we perform vanilla K-means on X to find 2 clusters. We do not tweak/restrict the distance metric and the cluster size. Then, we sort the points based on the point's cluster identity.

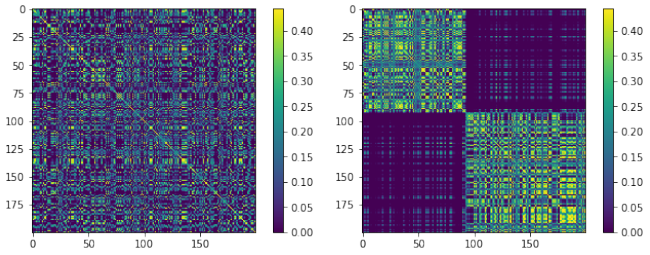

Figure 7: Heatmap of the $\widehat{\mathbf{K}}_{XX}$ matrices - left is the kernel matrix of X_0, right is after.

$$\ell^{(0)} = 23.52 \to \ell^{(\text{Cluster-Vanilla})} = 31.36, \frac{|\ell^* - \ell^{(\text{Cluster-Vanilla})}|}{\ell^*} = 0.0618\%$$

Since we did not make any restrictions; we can expect that the cluster might be imbalanced.

Sized K-Means

We take into account the equal sized constraint, as given in Same-size k-Means Variation. The implementation used to test is given in https://github.com/ndanielsen/Same-Size-K-Means.

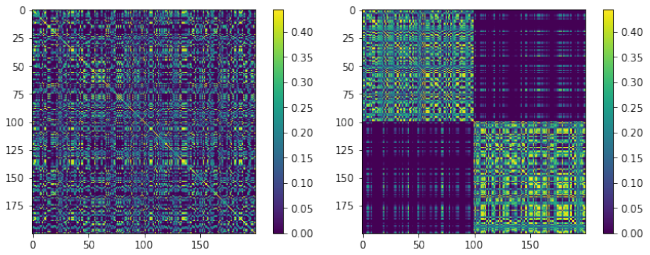

Figure 8: Heatmap of the $\widehat{\mathbf{K}}_{XX}$ matrices - left is the kernel matrix of X_0, right is after.

$$\ell^{(0)} = 23.52 \to \ell^{(\text{Cluster-Sized})} = 31.35, \frac{|\ell^* - \ell^{(\text{Cluster-Sized})}|}{\ell^*} = 0.0171\%$$

Distanced K-Means

We take into account the kernel distance $k(x_i, x_j)$, replacing the default euclidean distance. Implementation used is from pyclustering.

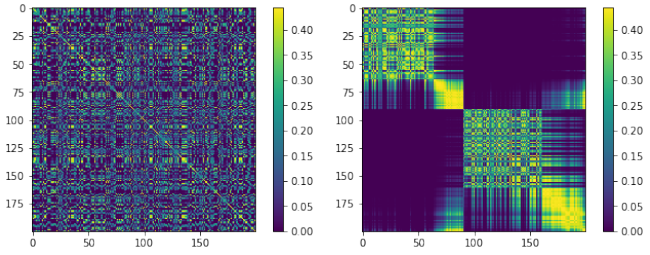

Figure 9: Heatmap of the $\widehat{\mathbf{K}}_{XX}$ matrices - left is the kernel matrix of X_0, right is after.

$$\ell^{(0)} = 23.52 \to \ell^{(\text{Cluster-Distanced})} = 31.29, \frac{|\ell^* - \ell^{(\text{Cluster-Distanced})}|}{\ell^*} = 0.169\%$$

We expect this method to perform better, also the matrix looks a little odd.

Graph

We can model the problem as a graph problem:

1. Vertex: Each x_i
2. Edge: $x_i \to x_j$, where weight is given by $k(x_i, x_j)$

Then, we are trying to find the a balanced, minimum cut of the graph. This problem is known also:

1. Planted Bisection Problem
2. Stochastic block model with two communities
3. Minimum bisection problem

Related resources:

1. On Minimum Bisection and Related Cut Problems in Trees and Tree-Like Graphs
2. Exact Recovery in the Stochastic Block Model
3. Community Detection in Networks: Algorithms, Complexity, and Information Limits
4. MIT - mathematics of data science fall 2015
5. Minimum Cut Graphs
6. An efficient heuristic procedure for partitioning graphs
7. A Linear-Time Heuristic for Improving Network Partitions

1. Implementaion is avaliable here kshitij1489/Graph-Partitioning, but it is in Cpp.

8. From Louvain to Leiden: guaranteeing well-connected communities

Kernighan-Lin Graph Bisection

Partition a graph into two blocks using the Kernighan–Lin algorithm; see An efficient heuristic procedure for partitioning graphs.

This algorithm partitions a network into two sets by iteratively swapping pairs of nodes to reduce the edge cut between the two sets. The pairs are chosen according to a modified form of Kernighan-Lin, which moves node individually, alternating between sides to keep the bisection balanced.

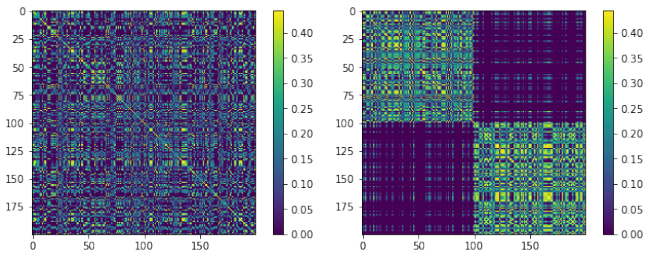

Figure 10: Heatmap of the $\widehat{\mathbf{K}}_{XX}$ matrices - left is the kernel matrix of X_0, right is after.

$$\ell^{(0)} = 23.52 \to \ell^{(\text{Graph-Kernighan})} = 31.38, \frac{|\ell^* - \ell^{(\text{Graph-Kernighan})}|}{\ell^*} = 0.108\%$$

Louvain Community Detection

Compute the partition of the graph nodes which maximises the modularity using the Louvain heuristices. This is the partition of highest modularity, i.e. the highest partition of the dendrogram generated by the Louvain algorithm. Implementation is from python-louvain, also see Fast unfolding of communities in large networks.

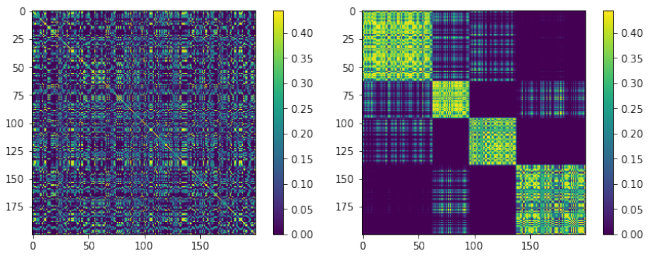

Figure 11: Heatmap of the $\widehat{\mathbf{K}}_{XX}$ matrices - left is the kernel matrix of X_0, right is after.

$$\ell^{(0)} = 23.52 \to \ell^{(\text{Graph-Louvain})} = 29.44, \frac{|\ell^* - \ell^{(\text{Graph-Louvain})}|}{\ell^*} = 6.065\%$$

Leiden Community Detection

leidenalg - yet to try

Linear Algebra

Principal Component Analysis

We sort the points using the transformed coordinate using PCA on X; we consider only the principal component. In this version, we use the default empirical sample covariance $X^\top X$; it would be ideal to use $\widehat{\mathbf{K}}_{XX}$, which should improve the performance. However, that would require us to implement PCA.

The general steps of PCA is as follows:

1. Standardize the dataset.

2. Calculate the covariance matrix for the features in the dataset.
3. Calculate the eigenvalues and eigenvectors for the covariance matrix.
4. Sort eigenvalues and their corresponding eigenvectors.
5. Pick k eigenvalues and form a matrix of eigenvectors.
6. Transform the original matrix.

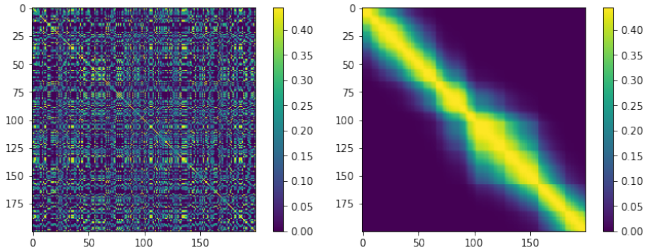

Figure 12: Heatmap of the $\widehat{\mathbf{K}}_{XX}$ matrices - left is the kernel matrix of X_0, right is after.

$$\ell^{(0)} = 23.52 \to \ell^{(\text{La-Pca})} = 31.44, \frac{|\ell^* - \ell^{(\text{La-Pca})}|}{\ell^*} = 0.308\%$$