Improved Fast Direct Methods for Gaussian Processes

Eric Han

May 17, 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 via Cholesky decomposition scales quadratically. The main contribution to the complexity is due to the factorization of the positive-semidefinite matrix $\widehat{\mathbf{K}}_{XX}$ into $L \times L^\top$:

- In general, the time complexity is $O(n^3)$.
- Space complexity is $O(n^2)$, needed to store the lower triangular factor L along with $\widehat{\mathbf{K}}_{XX}$.

Related Work

Broadly, there are generally 2 approaches; attempting to scale the factorization while preserving exact solution or performing approximations to speed-up computation.

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

The key differece between the 2 approaches is the accurate computation of the log likelihood. Specifically, the approximate methods sacrifice accuracy for speed. This work discusses the the impact of numerical issues with the computation on the hyperparameter learnt by the model: Numerical issues in maximum likelihood parameter estimation for Gaussian process interpolation.

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.

Process

Without forming the kernel matrix directly, using partitioned kernel MVM,

- Time complexity of $O(\frac{n^3}{p})$ assuming that the computation is parallelized by p partitions.
- Space complexity of $O(\frac{n^2}{p})$.

Note: Time complexity of the process is not discussed in the paper, which needs further checking to be confident.

MVM-based GP inference. Partition the kernel matrix to perform all matrix-vector multiplications without forming kernel matrix explicitly.

1. Partition $\mathbf{X} = \big[\mathbf{X}^{(1)}; \cdots; \mathbf{X}^{(l)}; \cdots; \mathbf{X}^{(p)}\big]$ to p parts (row)
2. For each l part:
    1. Compute $\widehat{\mathbf{K}}_{X^{(l)}X}$ with $\mathbf{X}^{(l)}, \mathbf{X}$
    2. Compute $\widehat{\mathbf{K}}_{X^{(l)}X}\mathbf{u}$

Memory requirement $O(n^2/p)$; and when $p \to n$: $O(n)$.

Conjugate Gradients (CG) Computation. 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.

Sampling m Inducing points

1. Sparse and Variational Gaussian Process (SVGP) [1] selects inducing points Z using a regularized objective.
2. Sparse Gaussian Process Regression (SGPR) [2] introduces a set of variational parameters that can be optimized using minibatch training.

Process

Along with finding the m-inducing points, the complexity required:

- Time complexity of $O(nm^2)$
- Space complexity of $O(nm)$

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 factorization precision ϵ. 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.

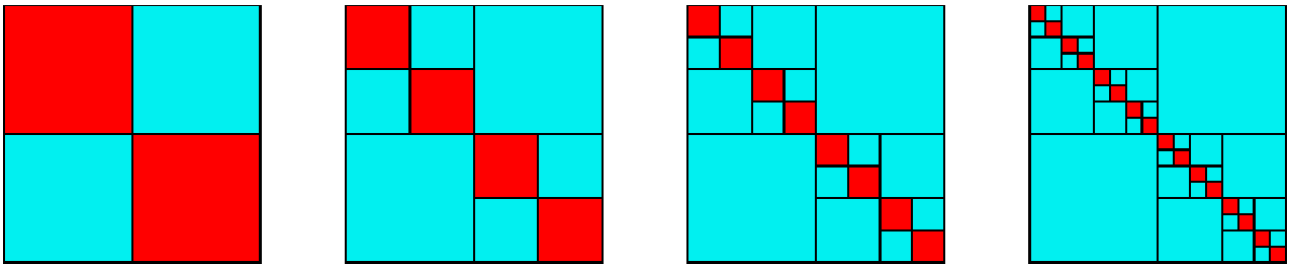


Figure 1: Pictorial representation of the HOLDR decompositons.

Process

The overall runtime to obtain the matrix inverse of $\widehat{\mathbf{K}}_{XX}$ and its determinant $|\widehat{\mathbf{K}}_{XX}|$ is $O(n \log^2 n)$:

- The factorization of an $n \times n$, κ-level (where $\kappa \sim \log n$) HODLR matrix takes $O(n \log^2 n)$.
- Given a HODLR-type factorization, finding the inverse takes $O(n \log n)$.
- Given a HODLR-type factorization, finding the determinant takes $O(n \log n)$.

The overall space required scales to the number of dense matrices $n/2^\kappa \times n/2^\kappa$:

- $O(2^\kappa \times \frac{n^2}{4^\kappa}) = O(\frac{n^2}{2^\kappa})$, assuming $\kappa = \log_2 n$, $O(n)$.

Note: Original paper did not discuss space complexity, need to put more thought and experiments; It is also not practical to subdivide completely, the authors implementation subdivide until a certain specified size.

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.

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

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

Metrics commonly used in Literature

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.

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

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.
3. Numerical issues in maximum likelihood parameter estimation for Gaussian process interpolation:
    1. Accuracy - Root mean squared prediction error (ERMSPE)
    2. Optimization is run on the model to find the optimal hyperparameters, then:
        1. Accuracy - Lengthscale
        2. Accuracy - Variance
        3. Accuracy - Negative log-likelihood
        4. Accuracy - Values of each individual term in the computation of the GP model.

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?

Motivation

Our key motivations for improving *Holdr* is as follows:

1. *Holdr* is fast; The runtime complexity is very good when compared to the next state-of-the-art.
2. *Holdr* is space efficient; The space complexity is very good when compared to the next state-of-the-art.
3. *Holdr* is not accurate; it relies on the assumption that $\widehat{\mathbf{K}}_{XX}$ resembles a HODLR matrix; failing which, the accuracy of the matrix inversion and determininant computations is affected in unpredictable ways. The accuracy of the computation improved by setting the factorization precision ϵ to a smaller number, but the computation can still fail unpredictably.
4. $\widehat{\mathbf{K}}_{XX}$ can be conditioned, into an equlivant hierarchical matrix $\widehat{\mathbf{K}}_{X'X'}$ by performing some reordering operations on $X \to X'$, improving the matrix structure.

Holdr is fast

The main advantage of *Holdr* when compared to other methods is that it is very fast (at scale). Even with a small $\epsilon = 0.000001$, *Holdr* is competitive on time. All methods use only the CPU; Alex is converted from running on GPU to running on CPU. We let Alex select the optimal kernel size based on the avaliable memory, thereby parallelizing on the CPU. Experiments here are done on the Synthetic Function, $\sin(x)$, each datapoint is the average over 100 independent runs, over the best of the 10 measurements done for each run.

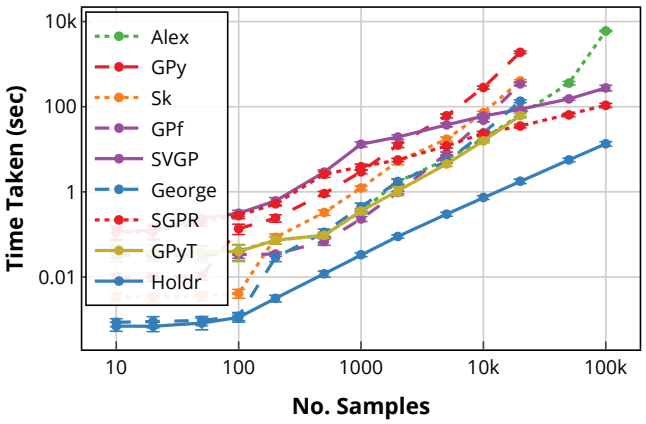
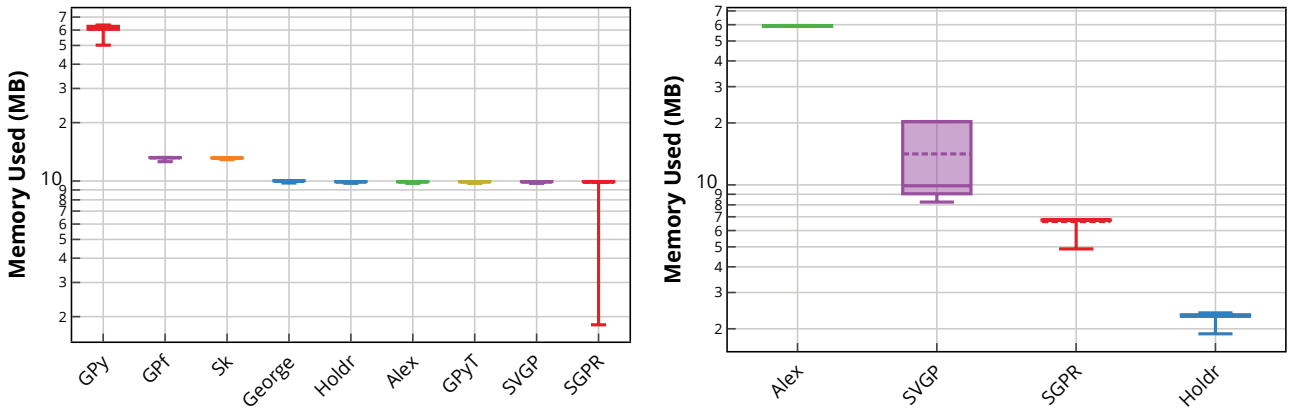


Figure 2: Take taken, *Holdr* with $\epsilon = 0.000001$ when compared with other methods.

Note: Some numerical instability with GPyT due to the implementation on PyTorch; the computation of Cholesky decomposition could not go through for <5% of the experiments and it is not reflected here. Will fix this shortly.

Holdr is space efficient

The next key advantage of *Holdr* when compared to other methods is that it is very space efficient (at scale).



(a) Memory used at $N = 20000$.

(b) Memory used at $N = 100000$.

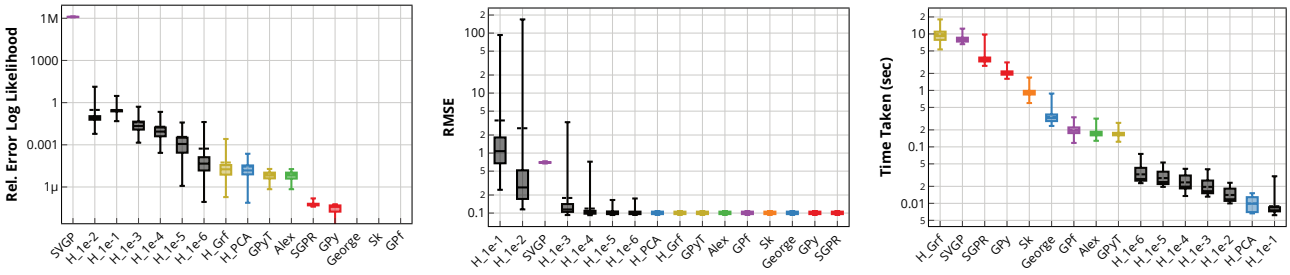
Figure 3: Holdr with $\epsilon = 0.000001$ when compared with other methods.

Note: Some issues with the how the memory required is measured, but comparisons within the same N is OK.

Holdr is not accurate

From Inconsistent results with HODLRSolver, we see that there are accuracy issues in its use. The accuracy of the computation improved by setting the factorization precision ϵ to a smaller number, but the computation can still fail unpredictably.

We can run *Holdr* with different factorization precisions $\epsilon \in \{1e^{-i} : i \in [1, 6]\}$ to determine its effects on accuracy and runtime complexity. 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 with the various ϵ.



(a) Effects on $\frac{|\ell^* - \ell_{\text{HODLR}}|}{\ell^*}$. (b) Effects on RMSE. (c) Effects on Time Taken.

Figure 4: Holdr with different factorization precisions.

We see clearly from 4a that with a small ϵ, it improves the average error but at an expense of high deviation. In other words, factoring with a small factorization precision yields better results on the average but can fail unpredictably. This give rise to the instability of the computation of log likelihood, which can adversely affect the optimization of model hyperparameters and subsequently RMSE (see 4b).

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.

Most importantly, the time taken needed to compute $\epsilon = 1e^{-i}$ scales well. Even at a small factorization precision $i = 6$, it is still faster than the fastest competition. This gives us a window of opportunity spend some compute time to condition $\widehat{\mathbf{K}}_{XX}$, even at $i = 6$.

$\widehat{\mathbf{K}}_{XX}$ can be conditioned

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].

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

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 $\log p(\mathbf{y}|\mathbf{X}, \theta)$.

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

Proof-of-Concept

Here, we demonstrate the feasibility of our concept by using the *optimal* metric with Genetic Algorithm. 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 $|\ell^* - \ell^{(i)}|$

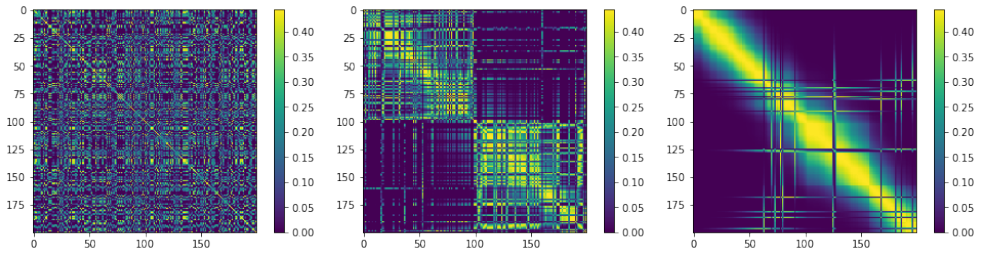


Figure 5: Heatmap of the $\widehat{\mathbf{K}}_{XX}$ matrices, showing from $X_0 \to X_{20} \to X_{1000}$.

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:

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

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.

Theory

Background using Travelling Salesman Problem (TSP)

TSP decision problem is NP-Complete.

- Input: Graph G and budget b
- Output: Does G admit a tour of weight at most b?

TSP search problem is NP-Complete.

- Input: Graph G and budget b
- Output: Find a tour of G of weight at most b, if it exists.

TSP optimization problem is NP-Hard.

- Input: Graph G
- Output: Find a tour of G with minimum weight.

Finding the reordered X'

Decision version: Given X and some absolute difference δ_ℓ, determine if there is a rearrangement X' such that the absolute difference of the accurate log-likelihood and the rearranged log-likelihood is at most δ_ℓ, ie. $|\ell^* - \ell^{(i)}| \leq \delta_\ell$.

- Input: Initial X and δ_ℓ
- Output: Can X be rearranged to X' of accuracy at most δ_ℓ?

We reduce a known NP-Hard problem to this problem. Specifically, we reduce from TSP to this problem.

We now reduce TSP to finding X', ie. show TSP $\leq_p$ finding X'. Every instance of TSP consists of a complete graph $G(V, E)$ with the budget b as the inputs. Given $G(V, E)$ with $|V| = n$, we can convert the complete graph G into an adjacency matrix $\widehat{\mathbf{K}}_{XX}$ in polynomial time, ie. over all edges $O(n^2)$. In this setup, we note that the kernel distance between the variables $k(x_i, x_j) = c(v_i, v_j)$ is equlivant to the distance between the edges.

Genetic Algorithm

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.

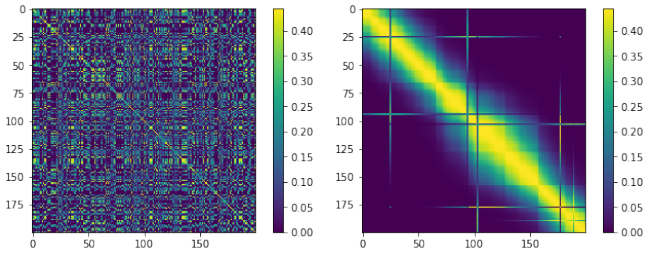


Figure 6: Heatmap of the $\widehat{\mathbf{K}}_{XX}$ matrices - left is the kernel matrix of X_0, right is after.

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

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.

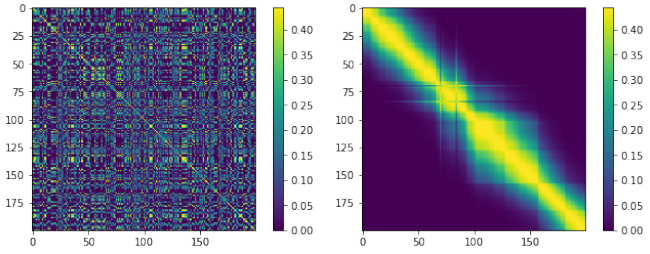


Figure 7: Heatmap of the $\widehat{\mathbf{K}}_{XX}$ matrices - left is the kernel matrix of X_0, right is after.

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

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.

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

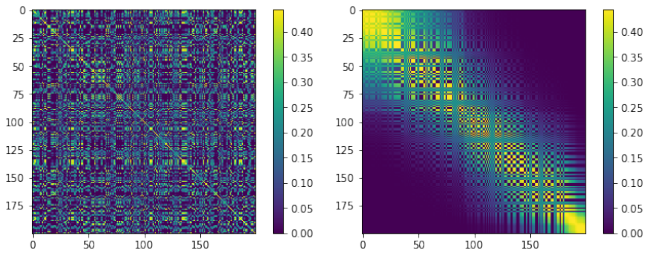


Figure 8: Heatmap of the $\widehat{\mathbf{K}}_{XX}$ matrices - left is the kernel matrix of X_0, right is after.

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.

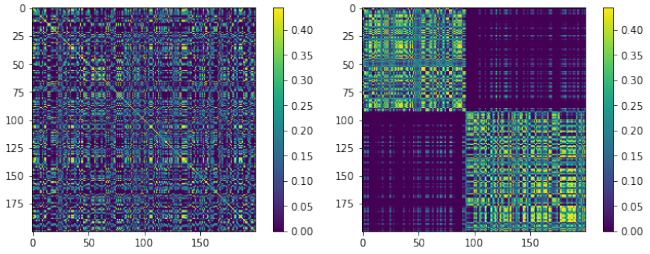


Figure 9: Heatmap of the $\widehat{\mathbf{K}}_{XX}$ matrices - left is the kernel matrix of X_0, right is after.

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

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.

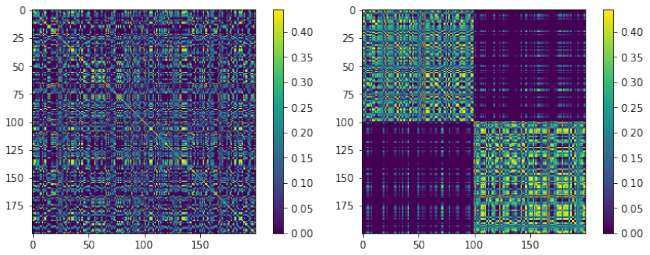


Figure 10: Heatmap of the $\widehat{\mathbf{K}}_{XX}$ matrices - left is the kernel matrix of X_0, right is after.

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

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.

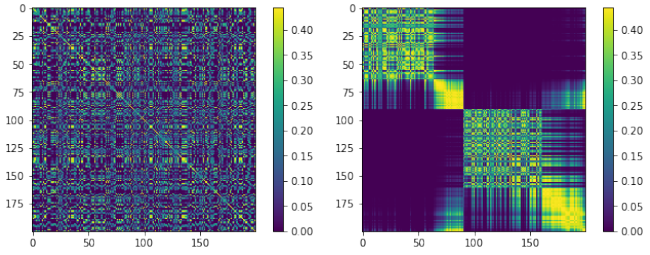


Figure 11: Heatmap of the $\widehat{\mathbf{K}}_{XX}$ matrices - left is the kernel matrix of X_0, right is after.

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

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.

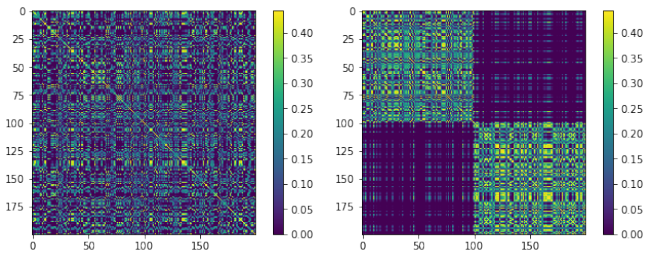


Figure 12: Heatmap of the $\widehat{\mathbf{K}}_{XX}$ matrices - left is the kernel matrix of X_0, right is after.

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

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.

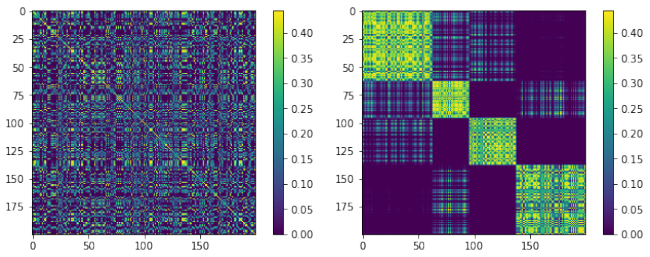


Figure 13: Heatmap of the $\widehat{\mathbf{K}}_{XX}$ matrices - left is the kernel matrix of X_0, right is after.

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

Leiden Community Detection

leidenalg - yet to try

Linear Algebra

Principal Component Analysis (PCA)

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.

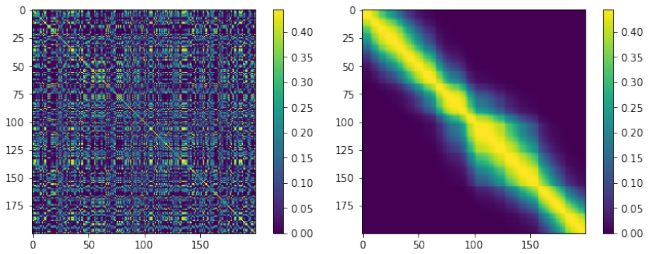


Figure 14: Heatmap of the $\widehat{\mathbf{K}}_{XX}$ matrices - left is the kernel matrix of X_0, right is after.

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

Kernel Principal Component Analysis

Instead, in kernel PCA, a non-trivial, arbitrary Φ function is 'chosen' that is never calculated explicitly, allowing the possibility to use very-high-dimensional Φ's if we never have to actually evaluate the data in that space. Since we generally try to avoid working in the Φ-space, which we will call the 'feature space', we can create the N-by-N kernel

$$K = k(\mathbf{x}, \mathbf{y}) = (\Phi(\mathbf{x}), \Phi(\mathbf{y})) = \Phi(\mathbf{x})^T \Phi(\mathbf{y})$$

which represents the inner product space (see Gramian matrix) of the otherwise intractable feature space. In summary, kernel PCA performs PCA in the Φ-space, which is determined by kernel $k(.)$. Here, we let $K = \widehat{\mathbf{K}}_{XX}$; solving the PCA in the arbitary Φ-space that corresponds to our chosen kernel.

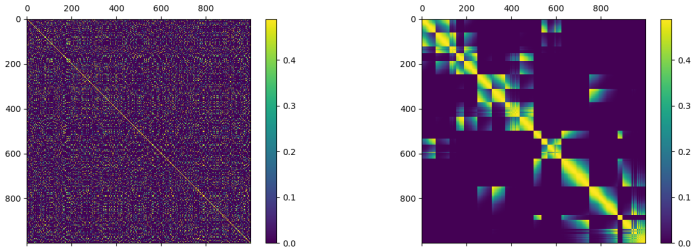


Figure 15: Heatmap of the $\widehat{\mathbf{K}}_{XX}$ matrices - left is the kernel matrix of X_0, right is after.

Note: Parts are lifted from Wikipedia, which needs to be rewritten.

Difference Sorting

https://www.youtube.com/watch?v=vE6Ds4jwxb4

Experiments

All holdr used $\epsilon = 0.0001$.

Dataset

We used a split the dataset into a training set (80%) and testing set (20%).

For all synthetic experiments, we add $z_t \sim \mathcal{N}(0, 0.1^2)$ noise to $y_t = f(\mathbf{x}_t) + z_t$ to simulate real-world noise.

[Synthetic] Function

We sample from the 1-Dimensional function $f(x) = \sin(x),$ where $x \in [-10, 10]$.

[Synthetic] GP Kernel

Here, we test our algorithm on synthetic data by sampling functions from GP. We use an RBF kernel with corresponding dimensional lengthscale and scale parameters set to $\sigma^2_{\text{opt}} = 0.5$ and $l_{\text{opt}} = 0.1$.

[Synthetic] HPOlib2

Commonly used synthetic functions $f(x)$ are from HPOlib2 [3], such as

HPOlib2	d
Forrester	1
Levy	1
Bohachevsky	2
Branin	2
Camelback	2
Hartmann6	6

[Real] UCI Data

Curated and processed UCI Dataset [4] from https://github.com/treforevans/uci_datasets [5].

UCI Dataset	Cite	n	d
3droad	[6]	434874	3
kin40k		40000	8
protein	[7]	45730	9
houseelectric		2049280	11
bike		17379	17
elevators		16599	18
keggdirected	[8]	48827	20
pol		15000	26
keggundirected	[8]	63608	27
buzz	[9]	583250	77
song	[10]	515345	90
slice	[11]	53500	385

Methods

All methods use the RBF kernel with σ_0^2 set to the population variance of the training sample and $l_0 = 1.0$. The model's noise variance is set $\eta^2 = 0.01$ to account for noisy observations.

- Stochastic Methods. Here we optimize the model parameters using the Adam Optimizer [12] which is designed for stochastic objective functions. We use the implementation of Adam from [13].
  - *SVGP*, implementation from [14]
  - *SGPR*, implementation from [14]
- Traditional, cholesky decomposition methods
  - *GPy* - popular GP implementation from Sheffield University [14]
  - *Sk* - scikit-learn implementation [15]
  - *George* - Cholesky implementation inside 'Fast Direct Methods for Gaussian Processes' [16]
  - *GPf* - scikit-learn implementation [17]
  - *GPyT* - GPyTorch implementation [18]
- Conjugate gradients method (CG)
  - *Alex* - Author's implementation for 'Exact Gaussian Processes on a Million Data Points' [19]
- Hierarchical Off-Diagonal Low-Rank (HODLR) matrix factorization method
  - *Holdr* - Author's implementation for 'Fast Direct Methods for Gaussian Processes' [20]
- Our methods:
  - *H_KM* - Holdr with K-Means

- *H_KM* - Holdr with K-Means with equal sized clusters
- *H_GA* - Holdr with Genetic Algorithm
- *H_PCA* - Holdr with PCA
- *H_KPCA* - Holdr with KPCA
- *H_Grf* - Holdr with Kernighan-Lin Graph Bisection

References

[1] J. Hensman, A. G. de G. Matthews, and Z. Ghahramani, "Scalable variational gaussian process classification," 2015.

[2] M. K. Titsias, "Variational learning of inducing variables in sparse gaussian processes," in *International conference on artificial intelligence and statistics*, 2009, pp. 567–574.

[3] K. Eggensperger *et al.*, "Towards an empirical foundation for assessing Bayesian optimization of hyperparameters," 2013.

[4] D. Dua and C. Graff, "UCI machine learning repository." University of California, Irvine, School of Information; Computer Sciences, 2017. Available: http://archive.ics.uci.edu/ml

[5] T. E. (treforevans), "UCI datasets," *GitHub repository*. https://github.com/treforevans/uci_datasets; GitHub, 2022.

[6] M. Kaul, B. Yang, and C. S. Jensen, "Building accurate 3D spatial networks to enable next generation intelligent transportation systems," in *2013 IEEE 14th international conference on mobile data management*, 2013, vol. 1, pp. 137–146. doi: 10.1109/MDM.2013.24.

[7] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," 2015. Available: https://networkrepository.com

[8] P. Shannon *et al.*, "Cytoscape: A software environment for integrated models of biomolecular interaction networks," *Genome research*, vol. 13, no. 11, p. 24982504, 2003, Available: https://genome.cshlp.org/content/13/11/2498.full.pdf+html

[9] F. Kawala, A. Douzal-Chouakria, E. Gaussier, and E. Dimert, "Prédictions d'activité dans les réseaux sociaux en ligne," 2013.

[10] T. Bertin-Mahieux, D. P. W. Ellis, B. Whitman, and P. Lamere, "The million song dataset," 2011.

[11] F. Graf, H.-P. Kriegel, M. Schubert, S. Pölsterl, and A. Cavallaro, "2d image registration in ct images using radial image descriptors," in *International conference on medical image computing and computer-assisted intervention*, 2011, pp. 607–614.

[12] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2015. Available: http://arxiv.org/abs/1412.6980

[13] J. Bayer, C. Osendorfer, S. Diot-Girard, T. Rückstiess, and S. Urban, "Climin - a pythonic framework for gradient-based function optimization," *TUM Tech Report*. http://github.com/BRML/climin; Technical University of Munich, 2022.

[14] GPy, "GPy: A gaussian process framework in python." http://github.com/SheffieldML/GPy, since 2012.

[15] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[16] S. Ambikasaran, D. Foreman-Mackey, L. Greengard, D. W. Hogg, and M. O'Neil, "Fast Direct Methods for Gaussian Processes," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 38, p. 252, Jun. 2015, doi: 10.1109/TPAMI.2015.2448083.

[17] A. G. de G. Matthews *et al.*, "GPflow: A Gaussian process library using TensorFlow," *Journal of Machine Learning Research*, vol. 18, no. 40, pp. 1–6, Apr. 2017, Available: http://jmlr.org/papers/v18/16-537.html

[18] J. R. Gardner, G. Pleiss, D. Bindel, K. Q. Weinberger, and A. G. Wilson, "GPyTorch: Blackbox matrix-matrix gaussian process inference with GPU acceleration," 2018.

[19] K. Wang, G. Pleiss, J. Gardner, S. Tyree, K. Q. Weinberger, and A. G. Wilson, "Exact gaussian processes on a million data points," in *Advances in neural information processing systems*, 2019, vol. 32. Available: https://proceedings.neurips.cc/paper/2019/file/01ce84968c6969bdd5d51c5eeaa3946a-Paper.pdf

[20] S. Ambikasaran, D. Foreman-Mackey, L. Greengard, D. W. Hogg, and M. O'Neil, "Fast direct methods for gaussian processes," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 38, no. 2, pp. 252–265, 2016, doi: 10.1109/TPAMI.2015.2448083.