DSA5208 Project 1

Parallel implementation of kernel ridge regression using MPI

Alon Rafael Landmann

A0276620E

June 12, 2024

1 Introduction

The objective of this assignment is to build a regression model for a data set containing data on different housing blocks in California, and to predict average housing prices in other blocks. Each sample in the data contains eight features such as total rooms, housing age, etc., as well as the median house value. Since this is a large dataset containing over 20,000 samples, we want to leverage the MPI protocol to train and test the model using multiple CPU cores.

2 Methodology

First, a bash script was written to split the data randomly into 70% reserved for training and 30% reserved for testing. It is important to have done this randomly, as this prevents the testing data from being biased away from the well trained domain by the initial ordering of the data. Testing also showed that the average prediction error was lower using the shuffled split. Furthermore, all the feature data was first standardized so that each feature has mean zero and variance one.

For this particular implementation, it was decided to use 6 CPU cores on an 8 core machine. Therefore, using the same script, both the training and the testing data were again split into 6 different files each, one for each process.

Implementing kernel ridge regression involves computing the similarity measure, known as the kernel, between each pair in the data set. The kernel used for this implementation was the Radial Basis Function or Gaussian kernel, given by

$$k(\mathbf{x_1}, \mathbf{x_2}) = \exp\left(-\frac{\|\mathbf{x_2} - \mathbf{x_1}\|^2}{2s^2}\right), \tag{1}$$

where $\mathbf{x_1}$ and $\mathbf{x_2}$ are the feature vectors without the dependent variable. s is one of the hyper-parameters used to tune the regression model, as it describes the length scale over which the function can vary.

All of the pairs computed according to Equation 1 are to be stored in the kernel matrix K. To achieve this in a distributed setting, each process stores only as many rows as it has samples. In a circular fashion, data from one process is sent to a process that hasn't seen the data yet, the respective kernels are computed and stored in the matrix, and then the memory is freed for the next transfer.

The posterior mean of the predicted housing value of a given test sample $\mathbf{x}$ is then given by

$$\mu(\mathbf{x}) = \mathbf{k}(\mathbf{x})^T (K + \lambda I)^{-1} \mathbf{y}, \tag{2}$$

where $\mathbf{k}(\mathbf{x})$ is the vector containing all the kernel function values obtained from matching the test vector $\mathbf{x}$ with each of the training vectors, $\mathbf{y}$ contains the true labels of the training data, and λ is the ridge hyper-parameter that is used to prevent over-fitting by discouraging large coefficients.

The matrix $K + \lambda I$ can be shown to be symmetric positive-definite (as long as λ is positive), and so finding the vector $(K + \lambda I)^{-1}\mathbf{y}$ is equivalent to solving an SPD linear system. We can therefore use the conjugate gradient method to obtain this vector efficiently. Doing so in a distributed environment involves parallel implementation of matrix-vector multiplication and inner product computation. Once a satisfactory accuracy (10^{-5} in our case) is achieved, the training of the model is done.

To test the model we compute μ according to Equation 2, and then compute the root mean squared error, given the true housing prices. This was done both for the training and testing data, and again, temporary sharing of one dataset with another process was used to implement an efficient computation of the error.

Finally, a grid search was performed to find optimal values for the hyper parameters λ and s. Figure 1 shows a one-dimensional example of how the choice of hyper parameters might affect the posterior mean. The parameter l in the legend refers to the length scale parameter s that was discussed. We can see that if s is too small, then a Gaussian regression model will spike to fit the individual data points, and then revert to the mean too quickly in between data points. If s is too large however, the model will not be able to vary enough locally to capture the natural variation present in the data. Finally, as discussed, a larger value of λ will prevent large coefficients and lead to an overall decrease in variation. We can see the effects of these choices in the final results.

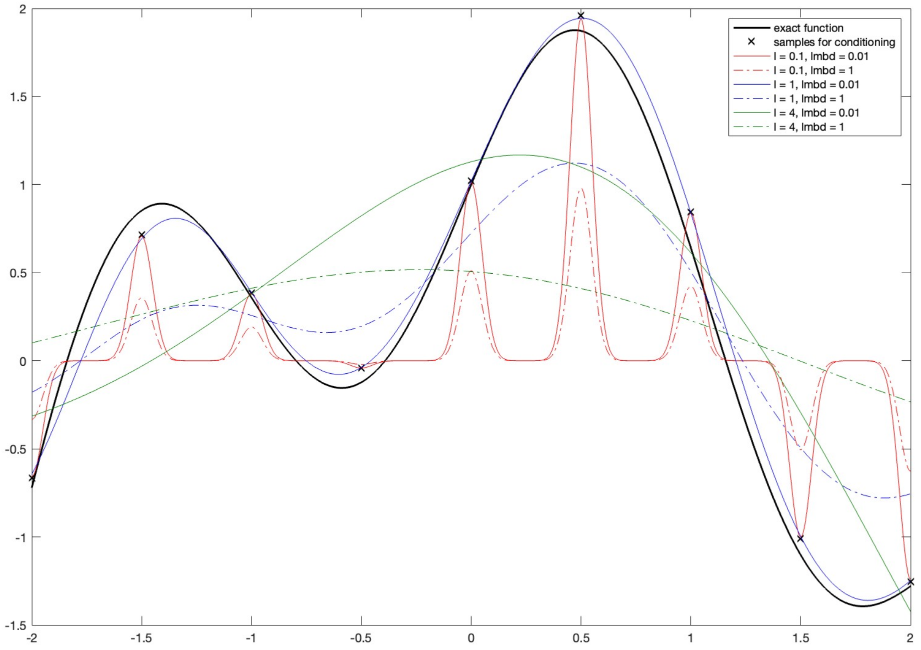

Figure 1: Example illustrating hyper-parameter tuning.

3 Results

All of the computations were done using the C language for scripting, and using MPICH for communication between the processes. Six CPU cores of the Apple M2 chip were used. The computational time for training and testing the model for a given set of hyper-parameters ranged from a few seconds to about 3 to 4 minutes, depending mainly on the number of iterations required for the conjugate gradient method to solve the linear system.

A few iterations of grid searches were performed to find better values for the hyper-parameters. The tables below show in each cell, in order, the number of iterations required to solve the linear system, the RMSE for the training data, and the RMSE for the test data.

$s\backslash\lambda$	**0.001**	**0.01**	**0.1**	**1.0**
0.1	88, 0.219, 216	84, 2.17, 217	61, 19.9, 218	27, 112, 224
0.3	2645, 1.90, 108	886, 6.65, 106	295, 18.9, 106	99, 63.5, 117
1.0	7434, 33.5, 69.5	2327, 39.4, 60.1	744, 45.0, 57.2	246, 52.1, 59.5
3.0	4446, 50.6, 55.6	1393, 53.1, 55.7	466, 55.7, 57.2	164, 58.5, 59.2
10.0	84, 2.17, 217	61, 19.9, 218	27, 112, 224	60, 66.0, 66.0

Table 1: Results for various λ and s values. The cells show the number of iterations to solve the linear system, the training RMSE, and the test RMSE from left to right. RMSE values are in thousands.

Table 1 shows the broadest search done. We can see the phenomena demonstrated by Figure 1 come into effect. For instance, a very low value of s leads to significant local over-fitting. The training RMSE is very low, but the model performs very poorly away from the individual data points, leading to a very large RMSE for the test data.

Searching around some of the better combinations from Table 1, we obtained Table 2. All of these results are already quite similar. Tables 3 and 4 show the final searches. The best result achieved was a test RMSE of 54,804 with $\lambda = 0.01$ and $s = 2.05$.

$s\backslash\lambda$	**0.005**	**0.01**	**0.05**
2.0	2786, 48.4, 55.1	1988, 49.3, 54.8	903, 51.4, 54.9
3.0	1976, 52.3, 55.4	1393, 53.1, 55.7	648, 54.9, 56.7
4.0	1424, 54.7, 57.1	1018, 55.4, 57.4	471, 56.8, 58.0
5.0	1988, 49.3, 54.8	903, 51.4, 54.9	378, 58.2, 59.0
6.0	1393, 53.1, 55.7	648, 54.9, 56.7	307, 59.1, 59.9
7.0	1018, 55.4, 57.4	471, 56.8, 58.0	251, 59.8, 60.5

Table 2: Results for various λ and s values. The cells show the number of iterations to solve the linear system, the training RMSE, and the test RMSE from left to right. RMSE values are in thousands.

$s\backslash\lambda$	**0.01**
1.5	2285, 46.1, 55.6
1.8	2114, 48.2, 54.9
2.2	1864, 50.2, 54.8
2.5	1653, 51.5, 55.0

Table 3: Results for various λ and s values. The cells show the number of iterations to solve the linear system, the training RMSE, and the test RMSE from left to right. RMSE values are in thousands.

$s\backslash\lambda$	**0.01**
1.9	2047, 48.745, 54.837
1.95	2014, 49.019, 54.818
2.0	1988, 49.283, 54.807
2.05	1951, 49.536, 54.804
2.1	1927, 49.779, 54.807

Table 4: Results for various λ and s values. The cells show the number of iterations to solve the linear system, the training RMSE, and the test RMSE from left to right. RMSE values are in thousands.