**DSA5208 Project 1: Kernel Ridge Regression**

**Objective:** Implement and evaluate kernel ridge regression (KRR) with Gaussian kernel for predicting house price in California

**Dataset:** California Housing Dataset

**Data Pre-processing:**

* Standardized input features
  + Ensured input features is scaled to a range of [0,1] for faster convergence, using MinMaxScalar
  + Input features includes all columns except the target feature (`medianHouseValue`)
* Split data into training and testing sets (70:30 ratio)
  + Used random state 2 to split 70:30 between training and testing sets

**Method:**

* The Gaussian Kernel was used:

A close-up of a number

Description automatically generated

s is a hyperparameter that will be tuned to achieve appropriate smoothness

* Linear system to solve:

A black text on a white background

Description automatically generated

Lambda is the regularization parameter which will be tuned to reduce overfitting

A mathematical equation with numbers

Description automatically generated

* The linear system will be solved using the above conjugate gradient method in a distributed manner to get the optimal alpha.

A black and white math symbol

Description automatically generated

* Predictions can then be performed with alpha vector and the gaussian kernel.

**Steps:**

Initialisation of MPI – The mpi4py library will be used to perform distributed computing.

* MPI Communicator: `comm = MPI.COMM\_WORLD`
* Rank: `rank = comm.Get\_rank()` # identifies the process ID
* Size: `size = comm.Get\_size()` # number of processes

Data Initialisation and Distribution

* After data pre-processing to standardise the data and splitting data into train and test sets, broadcast the entire training and test data to all the processes using comm.bcast

Distributed computation of kernel matrix

* The local coefficient matrix is computed within each process. The matrix variable local\_A contains the kernel computation between the portion of X\_train that is distributed to that process and the entire dataset X\_train. local\_A is a M\*N matrix where M is the number of rows distributed to the respective process while N is the number of rows in the entire train dataset.
* The regularization parameter lambda is added to the diagonal elements of the coefficient matrix while doing the kernel computation

Conjugate gradient method

* After initialising the local coefficient matrix, local\_alpha can be calculated within each process using the distributed local\_conjugate\_gradient function.
* While following the conjugate gradient algorithm, communication between the processes is required when performing matrix-vector multiplication and calculating the total squared error (using inner product).
* After setting a threshold of 1e-6 and maximum iterations of 1000, the weights alpha is initialised to be the zero vector and local\_r = local\_y; local\_p = local\_r
* Within the loop, all the local\_p vectors are gathered to all the processes within the global\_p variable. This is so that matrix-vector multiplication can be performed with the coefficient matrix A.
* comm.allreduce is also used to gather any inner product results in all the processes
* For example, the total squared error is calculated using comm.allreduce(np.dot(local\_r, local\_r), op=MPI.SUM)

Prediction and test using root mean squared error (RMSE)

* local\_alpha is used to get partial results for y in each process. Each local\_y is then summed up using comm.Reduce and final prediction results are returned to the root process.
* The root\_mean\_squared\_error function is imported from sklearn.metrics to perform RMSE calculation between predicted y values and true y values, for both the training and test sets.

Hyperparameter tuning

* Iterates through all possible combinations of s and lambda given by a range and calculate the RMSE\_train and RMSE\_test. Pick the best combination of hyperparameters based on the lowest RMSE\_test value
* Coarse grain tuning: utilise a large range of [1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1] for s and [1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1, 10, 100] for lambda
* Result: Best s 1.00e-01, Best lambda 1.00e-01, Best RMSE\_train 39133.2197, Best RMSE\_test 59770.2649
* Hence, proceeded with fine grain tuning, utilising a range of [0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.10, 0.11, 0.12] for s and [0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19, 0.20] for lambda
* Result: Best s 1.20e-01, Best lambda 1.10e-01, Best RMSE\_train 42310.3940, **Best RMSE\_test 56805.2882**