Erasmus University Rotterdam
Erasmus School of Economics
Bachelor Thesis Econometrics and Operations Research

A Novel Approach to Dynamic Ensemble Regression Using Validation Loss-Based Weighting for Improved Predictive Accuracy

Amr Tahio (523397)

Supervisor:	Hakan Akyuz
Second assessor:	Dr. Xiaomeng Zhang
Date final version:	1st July 2024

Abstract

Given the ongoing advances in technologies like self-driving vehicles and the pressing need for more stringent data security, data-decentralised environments are increasingly common. In sensitive fields such as defence or healthcare, data-sharing is often difficult or even prohibited. This poses challenges in implementing global predictive models which typically rely on the presence of high-quality centralised data. Dynamic ensemble methods offer a promising solution by aggregating predictions from multiple local models, and dynamically adjusting their weights based on performance. We propose a novel weighting mechanism that considers not only the predictive uncertainty of the models but also their generalisation performance, quantified by validation loss. We simulate a data-decentralised environment by partitioning the datasets and using the subsets to train separate regression neural networks. To compare the performance of the mechanism against benchmark methods, nine publicly available datasets are used from the UCI machine learning repository and the KEEL dataset repository. We find that incorporating validation loss into ensemble network weighting yields modest performance improvements compared to most existing methods.

1 Introduction

Given the ongoing advances in data analysis and machine learning techniques, maintaining data security and privacy has become a priority. Data-sharing between entities in sensitive sectors such as defence, telecommunications, and healthcare is therefore not always possible, or advisable. This has led to an increase in situations where data decentralisation is present and necessary as a tool to limit the impact of breaches. Improvements in technologies such as IoT (Internet of Things), blockchain, and self-driving vehicles, have also contributed to this, as data in these fields is often collected and distributed across a network of individual nodes rather than a centralised database. With practical difficulties in data-sharing across all the local nodes, it becomes challenging to deploy a global predictive model. Traditional machine learning models typically rely heavily on the availability of high-quality centralised data. Deploying models locally and training them on data available at their local node is a possible solution although it has been shown that the generalisation performance of these models is compromised if data distributions differ across local nodes (Lee & Kang, 2024).

This has given rise to dynamic ensemble methods, which have demonstrated superior performance to traditional ensemble methods such as classic bagging and boosting across a wide range of applications (Opitz & Maclin, 1999). Dynamic ensembles aggregate the predictions made by several models to produce a single prediction. This can help offset the poor generalisation performance of some of the individual local models. Unlike static

ensemble methods, dynamic methods assign and adjust weights dynamically, providing larger weights to better-performing individual models. This means that the influence of each model on the final prediction is variable, and dependent on its performance using the query which it has been provided. The weighting mechanism used to combine the individual predictions heavily influences the performance of these models and remains a challenge. Improvements in these mechanisms allow for the creation of robust, accurate predictive models that crucially, eliminate the need for data-sharing across nodes.

Recent approaches such as the one adopted by Lee & Kang (2024) have generally resolved the data-sharing limitations, but find that the performance of individual nodes heavily influences overall model performance. That is to say, model generalisation remains an issue, and noisy data at the node level has a significant impact on the performance of the ensemble. Despite the model generalisation metric validation loss typically being used as a stop criterion for neural network training Prechelt (2002) (training stops when validation loss does not improve), it has so far never been incorporated into the calculation of ensemble weights. To address this, we attempt to improve on the weighting mechanism presented by Lee & Kang (2024). This paper introduces a novel weighting mechanism that considers not only the predictive uncertainty of individual models but also generalisation performance, extending the work of Lee & Kang (2024). The central research question of the paper is: How can the weighting mechanism in dynamic ensemble regression neural networks be improved by incorporating prediction errors?

To answer the question, we follow a methodology similar to that of Lee & Kang (2024), but also incorporate validation loss, a measure of generalisation performance, into the weighting mechanism. First, we replicate the results of Lee & Kang (2024), and then we introduce the extension. We make use of nine open-source regression datasets from the UCI machine learning repository (Dua et al., 2017), and the KEEL dataset repository (Derrac et al., 2015). We simulate a data-decentralised environment by partitioning each dataset into several subsets with distinct characteristics and distributions, each representing a local node. Each subset is then used to train a separate regression neural network and produce a prediction and a weight, based on predictive uncertainty and generalisation performance. Nodes with higher predictive accuracy and lower validation loss post-training, are given larger weights. The weights and predictions are then combined to form a final ensemble prediction.

The results show that in general, the incorporation of validation loss into the weighting mechanism of ensemble networks improves on the performance of most existing baseline methods. These findings contribute to the field of machine learning and data security, indicating a potential for further improvement with other approaches to validation loss-based weighting.

The paper is structured as follows: Section 2 presents the theoretical framework and background, Section 3 presents the datasets used, and Section 4 presents the methodology, formulating the methods used mathematically. Finally, Section 5 presents our findings, and Section 6 concludes, summarising and providing suggestions for future research.

2 Literature Review

2.1 Static and Dynamic Ensemble Methods

Ensemble methods aim to make better predictions by aggregating forecasts from multiple models. These models often capture the distribution of the underlying data more accurately (Shahhosseini et al., 2022). Early ensemble methods include Bayesian averaging, Bagging (Bootstrap Aggregating) and Boosting. Dietterich (2000) reviewed these methods and found that they frequently outperform any single model within the ensemble. These methods are used widely today and are examples of static ensembles. The defining feature of a static ensemble is the fact that model weights are not adjusted based on the input data - The technique for merging the models does not change once it has been determined (Breiman (2001); Freund & Schapire (1997)). On the contrary, in a dynamic ensemble, model weights are adjusted continuously, with better models being assigned larger weights depending on the query(Britto Jr et al. (2014);Kolter & Maloof (2007)).

Lee & Kang (2024) state that static ensembles may not perform optimally when the data distributions differ within the ensemble. In such cases, dynamic ensembles are often more effective. The individual model weighting within an ensemble plays a crucial role in the overall model performance.

2.2 Ensemble Weighting Techniques

Several papers have looked into different weighting methods for ensemble networks. Zhang et al. (2019) presents a dynamic weighting framework for dynamic ensemble selection (DES) systems that uses local information around the query sample, improving classifier selection and fusion. Soares & Araújo (2015) proposes a dynamic ensemble regression (DOER) method that adapts to time-varying environments, updating model weights based on their recent performance. Qiao & Wang (2021) apply the theory of ensemble networks to temperature prediction in ladle furnaces for the first time. They propose a method which selects only the most competent base model from a pool of base models for each test pattern.

2.3 Quantifying Uncertainty of Ensemble Regression Neural Networks

Neural networks are often referred to as black boxes due to their lack of interpretability. Along with this, comes the challenge of quantifying the predictive uncertainty of a neural network. Abdar et al. (2021) conducted an extensive review of existing techniques for uncertainty quantification, citing several challenges. One of the techniques they examine is MC-Dropout, proposed by Gal & Ghahramani (2016). MC-Dropout is a method which obtains multiple predictions from a single model and takes the variance in these predictions as a measure of the predictive uncertainty of that model. It is a flexible technique that does not aggressively restrict the architecture of neural networks.

3 Data

Throughout the paper, nine publicly available regression datasets are used, originating from two sources: Bikesharing, Ctscan, Indoor- loc, and Telemonitoring from the UCI machine learning repository (Dua et al., 2017), and Compactiv, Cpusmall, Mv, Pole, and Puma32h from the KEEL dataset repository (Derrac et al., 2015). These datasets are widely used in the field of machine learning for model training and validation. To ensure accurate replication of the study performed by Lee & Kang (2024) we use their exact files, unchanged, and provided to us in the form of nine CSV files.

Table 1 shows a description of the datasets and their corresponding domains, number of features and observations. We partition each dataset into subsets by method of K-means clustering. The number of local nodes represents the number of clusters formed, and their corresponding size is also shown. This will be further explained in Section 4.

Table 1: Description of datasets and their subsets

Dataset	No. instances	No. features	Domain	No. local nodes	Local dataset sizes
Bikesharing	17,397	12	Bike Sharing System	9	[1173, 1368, 1558, 1722, 1949, 2145, 2277, 2312, 2375]
Compactiv	8,192	21	Computer System	4	[419, 734, 1345, 4470]
Cpusmall	8,192	12	Computer System	5	[412, 1204, 1227, 1902, 2375]
Ctscan	53,500	385	Computed Tomography	8	[3212, 3460, 4364, 5065, 5501, 7727, 8197, 13025]
Indoorloc	19,937	525	Indoor Positioning System	6	[1225, 2007, 2135, 2409, 4702, 4703]
Mv	40,768	10	Synthetic	10	[2991, 3240, 3394, 3495, 4148, 4242, 4287, 4303, 5111, 5557]
Pole	14,998	26	Telecommunication	5	[1338, 1572, 2675, 3067, 4287]
Puma32h	8,192	32	Robot Arm	10	[779, 783, 797, 806, 809, 817, 828, 848, 854, 871]
Telemonitoring	5,875	20	Parkinson's Disease	7	[474, 535, 630, 679, 715, 1055, 1421]

4 Methodology

This Section outlines the methods used to obtain predictions and predictive uncertainties from the neural network ensemble. First, the datasets are partitioned into subsets by K-means clustering and each subset is used to train a separate regression neural network. These are referred to as local nodes, and we assume them to be separate from each other. This is done to simulate a data-decentralised environment. Then, Monte Carlo (MC) dropout is applied to obtain numerous predictions from each neural network, which are then used to quantify the predictive uncertainty of said network. We also take the mean of these predictions, to arrive at a single prediction per neural network. The predictive uncertainty, and the validation loss of each network are combined to form a weight for that neural network. Finally, we combine the prediction and weight output by each neural network, to form a single ensemble prediction. Figure 1 shows an illustration of the full process, as detailed in Lee & Kang (2024). We explain this process in more detail below, building up to an explanation of our proposed weighting method.

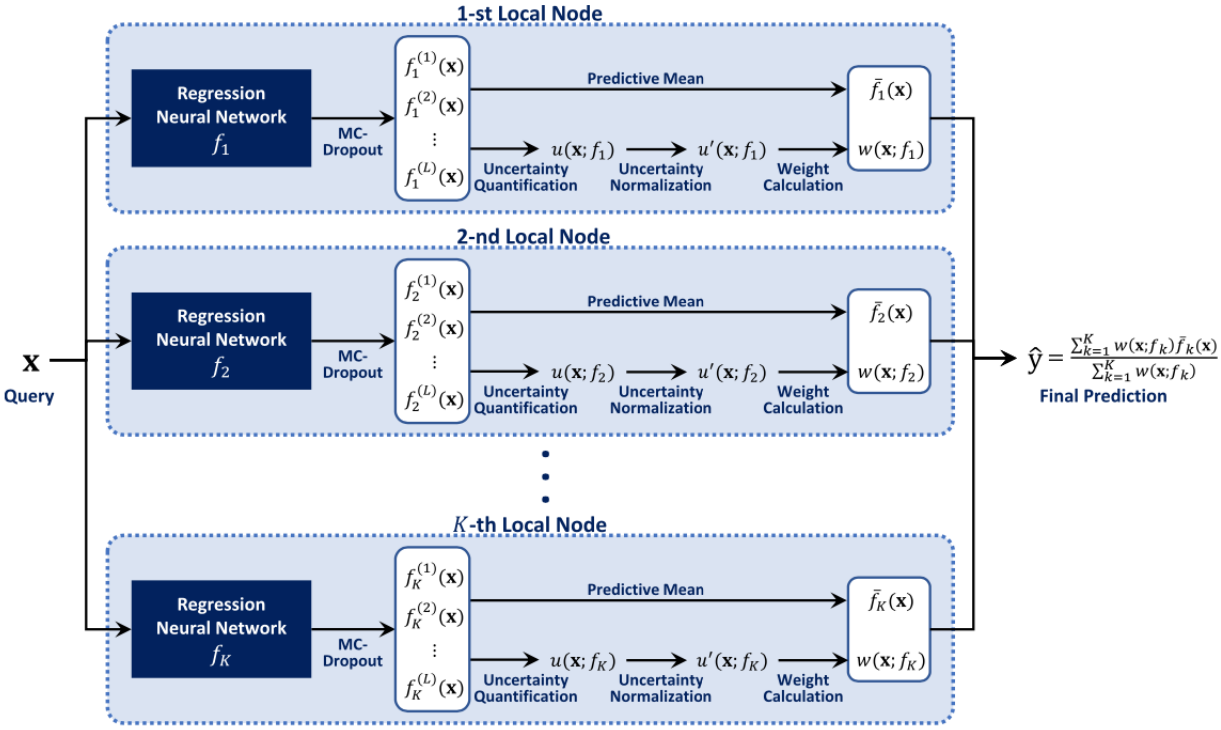

Figure 1: Model architecture, from Lee & Kang (2024)

4.1 Assumptions of the Data-Decentralised Environment

In order to best simulate a data-decentralised environment, we make several assumptions. They represent the constraints that would need to be met in a real-life scenario where data security and privacy are essential:

1. **No data-sharing between local nodes:** We assume that data used in one local node cannot be accessed by other nodes. Applying this to the real world scenario of smart self-driving vehicles, this would mean that data from one vehicle could not be accessed by any other vehicle in the network.

2. **Regression Neural networks are locally trained and maintained**: We assume that each neural network is local, and has no access to the underlying datasets or their distributions during prediction. This means that following model training, the ensemble can simply be provided with a query, and it is able to output a final prediction. This assumption, introduced by Lee & Kang (2024) deviates from existing methods, and adds another layer of data security.

3. **Datasets at different nodes have differing distributions**: As would likely be the case in real-world situations, every local node is trained on data with a unique distribution. We simulate this using K-means clustering, as will be explained in subsection 4.2.

4.2 Partitioning the Datasets

In order to simulate a data-decentralised environment and split the datasets into distinct subsets, we first apply principal component analysis (PCA) to reduce the dimensionality of the datasets. PCA aims to lower the number of features in a dataset, reducing them to a set of uncorrelated principal components (Abdi & Williams, 2010). This leads to less noisy data found in the original datasets, allowing us to create well-defined clusters using K-means clustering. We performed PCA with a maximum dimensionality of 10, meaning that the number of features in each dataset was reduced to at most 10.

Next, we apply K-means clustering to split the datasets into 10 disjoint clusters with distinct characteristics. We remove any cluster which forms less than 5% of the total dataset as they may be too small to train a neural network effectively. We denote the clusters as $D_1, D_2, ..., D_K$. Each cluster represents a subset in a local node. As an example, for the Bikesharing dataset, $K = 9$, meaning the dataset was split into 10 subsets, one of which was removed, and nine corresponding neural networks were included in the ensemble, as shown in Table 1.

4.3 Uncertainty Quantification: MC-Dropout

For every dataset D_k in a local node, we train a corresponding neural network f_k as shown in Figure 1. We split every dataset D_k into a training subset D_k^{trn} and a validation subset D_k^{val} which we use to train the network. We elaborate on the details of the functioning of the regression neural networks $f_1, f_2, ..., f_K$ in further subsections, but for now, they can

be considered black boxes that can be provided with a query $\mathbf{x}$ and return predictions for a target variable.

A key component in the weight assigned to each local node in the ensemble is the predictive uncertainty of the neural network corresponding to that node. In order to quantify the uncertainty of regression neural network f_k, we use MC-dropout (Gal & Ghahramani, 2016). The core idea of MC-dropout is to obtain multiple predictions from a single neural network and take the variance in these predictions as a measure of the predictive uncertainty of that neural network. This works by passing data through the network L times. With each forward pass, there is a probability p that certain hidden units in f_k are deactivated. This is a randomisation step which provides us with slightly different prediction values for each forward pass, as the underlying neural network structure is slightly altered. The end result from neural network f_k is L different predictions $f_k^1(\mathbf{x}), ..., f_k^L(\mathbf{x})$. We then take the mean of these predictions as the final prediction for f_k and the variance as its predictive uncertainty:

$$\bar{f}_k(\mathbf{x}) = \frac{1}{L}\sum_{l=1}^{L} f_k^{(l)}(\mathbf{x}) \tag{1}$$

$$u(\mathbf{x}; f_k) = \frac{1}{L}\sum_{l=1}^{L}\left(f_k^{(l)}(\mathbf{x}) - \bar{f}_k(\mathbf{x})\right)^2 \tag{2}$$

As a final step, we normalise the predictive uncertainty for each neural network. This is due to our earlier assumption that datasets at local nodes have different data distributions, meaning that comparing predictive uncertainties between networks cannot be readily done. We do this by first averaging the predictive uncertainties of f_k on the validation subset D_k^{val} to obtain $\bar{u}_k$:

$$\bar{u}_k = \frac{1}{|D_k^{\text{val}}|}\sum_{(x_i,y_i)\in D_k^{\text{val}}} u(\mathbf{x}_i; f_k). \tag{3}$$

We then divide the original predictive uncertainty by $\bar{u}_k$ to obtain the normalised predictive uncertainty of network f_k shown in Figure 1:

$$u'(\mathbf{x}; f_k) = \frac{u(\mathbf{x}; f_k)}{\bar{u}_k}. \tag{4}$$

We aim to obtain a final prediction $\hat{y}$ from the ensemble, as shown in Figure 1:

$$\hat{y} = \frac{\sum_{k=1}^{K} w(\mathbf{x}; f_k)\bar{f}_k(\mathbf{x})}{\sum_{k=1}^{K} w(\mathbf{x}; f_k)}, \tag{5}$$

where $w(\mathbf{x}; f_k)$ is a weight function based on the predictive uncertainties presented above, and the validation loss of each network. Our proposed weighting mechanism is

explained next.

4.4 Proposed Novel Weight Calculation Method

We design a weighting function that uses predictive uncertainty and validation loss of each network, to assign a weight for each node in the ensemble. Additionally, we normalise the uncertainties as this was shown to be ideal by Lee & Kang (2024) using $u'(x; f_k)$ as first presented in equations 3 and 4

$$w(x; f_k) = \exp(\frac{\alpha}{u'(x; f_k)} \cdot \frac{1}{V_k}) \tag{6}$$

where

$$V_k = \frac{1}{|D_k^{\text{val}}|} \sum_{i=1}^{|D_k^{\text{val}}|} (y_i - \hat{y}_i)^2 \tag{7}$$

is the validation loss corresponding to f_k. This is simply the mean squared error (MSE) of the neural network's predictions on unseen data in the validation set D_k^{val}. Since validation loss is a measure of model generalisation performance, it is inversely related to the model performance. Networks that exhibit a higher validation loss, have a lower generalisation performance and should be assigned a lower weight, hence we multiply the first part of the expression by $\frac{1}{V_k}$. We use α as a scaling factor that represents the variation in the assigned weights. If $\alpha = 0$ then all nodes are assigned equal weights, and a larger α means more importance is given to networks with low uncertainty and validation loss, meaning a larger disparity amongst the weights. In our experiments, we set α to 0.1,1 or 10 depending on the scale of the validation losses and the normalised uncertainties.

What differentiates our proposed weighting mechanism from that of Lee & Kang (2024) and other literature, is the incorporation of validation loss in the weight of the nodes in the ensemble. Validation loss is a metric frequently used during model training as a stop condition but has never been used to influence ensemble weights. In our experiments, for example, the regression neural networks conclude training if the validation loss of the network does not improve for 20 successive forward passes, or epochs (This will be explained in more detail in subsection 5.1. After training, we store the validation loss of each network f_k based on the validation subset D_k^{val}, and incorporate it in the weighting mechanism as in Equation 6. Our proposed method is an addition to the method proposed by Lee & Kang (2024), which excludes the validation loss term as follows:

$$w(x; f_k) = \exp(\frac{\alpha}{u'(x; f_k)}) \tag{8}$$

4.5 Compared Methods

We compare the performance of our proposed method with the following baseline methods used in Lee & Kang (2024):

- **Mean:** The first method simply assigns an equal weight to each local node, with the following weighting function:

$$w(\mathbf{x}; f_k) = \frac{1}{K}. \tag{9}$$

  This means that the final ensemble prediction is simply the mean of all the predictions:

$$\hat{y} = \frac{1}{K} \sum_{k=1}^{K} \bar{f}_k(\mathbf{x}). \tag{10}$$

- **Median:** This method takes the median of the individual predictions as the final ensemble prediction:

$$\hat{y} = \text{median}\{\bar{f}_1(\mathbf{x}), \bar{f}_2(\mathbf{x}), \ldots, \bar{f}_K(\mathbf{x})\}. \tag{11}$$

- **Dynamic Selection using Unnormalised Predictive Uncertainty (D-SEL(u)):** This method assigns one neural network with a weight of 1, namely the one with the lowest predictive uncertainty, and assigns 0 to all other networks. This is done using an indicator function, which returns a value of 1 for the k at which the predictive uncertainty of the corresponding network f_k is minimal:

$$w(\mathbf{x}; f_k) = I\left(k = \underset{j \in \{1,\ldots,K\}}{\text{argmin}}\ u(\mathbf{x}; f_j)\right). \tag{12}$$

- **Dynamic Selection using normalised Predictive Uncertainty (D-SEL(u')):** This method is identical to the above but uses the normalised uncertainties presented in Equation 4.3

$$w(\mathbf{x}; f_k) = I\left(k = \underset{j \in \{1,\ldots,K\}}{\text{argmin}}\ u'(\mathbf{x}; f_j)\right). \tag{13}$$

- **Dynamic Ensemble using Unnormalised Predictive Uncertainty (D-ENS(u)):** This method is a variation of the method proposed by Lee & Kang (2024) in Equation 8. The key difference is that it uses the unnormalised predictive uncertainties as opposed to the normalised ones:

$$w(\mathbf{x}; f_k) = \exp\left(\frac{\alpha}{u(\mathbf{x}; f_k)}\right). \tag{14}$$

- **Oracle:** Finally, oracle is used as a reference point, and considered to be a proxy for the best possible performance. This is because it disregards the earlier assumption that networks do not have access to the underlying datasets post-training. It uses this information to select the neural network from which the input query $\mathbf{x}$ originated. It does so by finding the k_* that minimises the distance between query $\mathbf{x}$ and the elements in the dataset D_k corresponding to neural network f_k:

$$\hat{y} = \bar{f}_{k_*}(\mathbf{x}), \quad \text{where } k_* = \underset{k \in \{1,\ldots,K\}}{\operatorname{argmin}} \min_{(\mathbf{x}_i, y_i) \in D_k} \|\mathbf{x} - \mathbf{x}_i\|. \tag{15}$$

5 Results

This section discusses the setup of our experiments, and analyses our main findings.

5.1 Experimental Settings

The architecture shown in Figure 1 shows a higher level overview of the ensemble of neural networks. The networks were all trained in an identical fashion, and with the same settings. As in Lee & Kang (2024), we use a standard feed-forward architecture with three hidden layers, each containing 128 hidden units and using the ReLU activation function. We use mean squared error (MSE) as the loss function, and we terminate network training if this value does not improve for 20 successive epochs, or after 500 epochs have been reached. An epoch is one complete pass through the entire training dataset. During one epoch, the model's parameters are updated based on the training data.

We applied MC-Dropout with a probability of 0.1 for each hidden layer, setting L=20 (The number of predictions per local node, averaged to produce the final prediction of that node, see figure 1). This means that each hidden layer has a probability of dropping out equal to 0.1, and this is how 20 slightly different predictions are obtained for each local node. Hyperparameter α, introduced and explained in Section 4.4, was set to 10, except in a few cases where weights grew too large due to the addition of the validation loss term, and it was set to 0.1. The experiment was repeated 10 times using random seeds.

5.2 Results and Discussion

Firstly, we replicate the experiments carried out by Lee & Kang (2024). Tables 2 and 3 show the generalisation performance of the method proposed by Lee & Kang (2024) and

other baseline methods for the nine benchmark problems, in terms of MAE and RMSE. The final row shows an average rank for each method, over the nine benchmark problems (Excluding oracle as this is simply a reference method). For every row, the number in bold represents the best performing method for the corresponding dataset (Once again excluding oracle from the comparison). The results align with those of Lee & Kang (2024), clearly showing that their proposed method outperforms baseline methods for three of the benchmark problems in terms of MAE and six in terms of RMSE.

Additionally, we confirm the pattern noted by Lee & Kang (2024): Their method performs worse than basic measures such as the mean, when oracle performs badly. Weak performance from the oracle method is a clear indication that individual neural networks are not performing well at their local nodes. As explained in section 4.5, oracle is used as a reference point and indicator of the best possible performance of the ensemble. This is because it disregards our earlier assumption about inaccessibility of underlying data by local nodes, and selects the neural network from which the query originated. In short, poor performance of individual neural networks in the ensemble, heavily compromises the performance of Lee & Kang (2024) weighting method.

We also notice that the Lee & Kang (2024) method seems to exhibit stronger performance when we observe RMSE. It is the top performing method on six datasets in terms of RMSE, as opposed to only three when we look at MAE. This could be due to several factors and the differences between these two metrics. Firstly, RMSE is more sensitive to outliers because it squares the individual prediction errors before averaging them. This means that larger errors are disproportionately more influential in the final RMSE value. Therefore, a method that minimizes large errors will show a big reduction in RMSE. In contrast, Mean Absolute Error (MAE) averages the absolute differences between predicted and actual values without squaring them, treating all errors equally. This means that MAE is less affected by outliers and large deviations. Hence the improved performance metrics for RMSE, could be an indication that the method by Lee & Kang (2024) is particularly strong at handling outliers and large deviations. This makes sense since the method uses variation in predictions as the main performance measure, assigning less weight to nodes where predictions exhibit higher variation.

We also note the superior performance of D-SEL(u') over D-SEL(u) in both tables 2 and 3. This confirms the importance of uncertainty normalisation, and is the reason we choose to normalise the uncertainties in our novel proposed method.
Next, we move to Tables 4 and 5 to analyse the performance of our novel proposed method. Due to computational constraints, we asses the performance of our method using only the five smallest datasets out of the nine introduced earlier. Both tables show that our novel method outperforms all of the baseline methods, with the exception of that of Lee &

Table 2: Performance comparison of baseline and proposed methods in terms of MAE (mean ± standard deviation).

Dataset	Oracle	Mean	Median	D-SEL(u)	D-SEL(u')	D-ENS(u)	Method Lee & Kang (2024)
Bikesharing	0.2358 ± 0.0061	0.8033 ± 0.0970	0.4762 ± 0.0263	0.4994 ± 0.0560	0.4070 ± 0.0389	0.4954 ± 0.0550	**0.4004 ± 0.0340**
Compactiv	0.1218 ± 0.0064	0.2429 ± 0.0377	0.2440 ± 0.0414	0.2552 ± 0.1412	0.1753 ± 0.0115	0.2550 ± 0.1413	**0.1733 ± 0.0110**
Cpusmall	0.1169 ± 0.0014	**0.2076 ± 0.0050**	0.2452 ± 0.0039	0.2465 ± 0.0053	0.2470 ± 0.0102	0.2462 ± 0.0054	0.2248 ± 0.0096
Ctscan	0.0310 ± 0.0010	0.5782 ± 0.0175	0.5132 ± 0.0148	0.3518 ± 0.0340	**0.1201 ± 0.0158**	0.3515 ± 0.0340	0.1320 ± 0.0157
Indoorloc	0.0476 ± 0.0010	0.9779 ± 0.0675	0.6289 ± 0.0367	0.0807 ± 0.0129	**0.0614 ± 0.0037**	0.0807 ± 0.0129	0.0646 ± 0.0041
Mv	0.0149 ± 0.0006	0.5505 ± 0.0218	0.2229 ± 0.0200	0.8761 ± 0.0480	**0.0661 ± 0.0118**	0.9573 ± 0.0600	0.0668 ± 0.0095
Pole	0.0643 ± 0.0032	0.5542 ± 0.0491	0.1655 ± 0.0122	0.0814 ± 0.0047	0.0791 ± 0.0056	0.0810 ± 0.0047	**0.0752 ± 0.0045**
Puma32h	0.7813 ± 0.0018	**0.7773 ± 0.0005**	0.7787 ± 0.0010	0.7888 ± 0.0061	0.7852 ± 0.0026	0.7885 ± 0.0061	0.7797 ± 0.0017
Telemonitoring	0.1771 ± 0.0037	0.7050 ± 0.0098	0.6164 ± 0.0244	0.6220 ± 0.0293	**0.4288 ± 0.0322**	0.6182 ± 0.0298	0.4332 ± 0.0264
Average rank	–	4.22	3.56	4.78	2.22	4.11	**1.78**

Table 3: Performance comparison of baseline and proposed methods in terms of RMSE (mean ± standard deviation).

Dataset	Oracle	Mean	Median	D-SEL(u)	D-SEL(u')	D-ENS(u)	Method Lee & Kang (2024)
Bikesharing	0.3727 ± 0.0079	1.0226 ± 0.0874	0.7655 ± 0.0450	0.8668 ± 0.0733	0.7272 ± 0.0622	0.8568 ± 0.0717	**0.6859 ± 0.0534**
Compactiv	0.2347 ± 0.0069	**0.4214 ± 0.0208**	0.4953 ± 0.0202	0.5437 ± 0.0934	0.4778 ± 0.0173	0.5433 ± 0.0936	0.4726 ± 0.0143
Cpusmall	0.2012 ± 0.0043	**0.5439 ± 0.0167**	0.7868 ± 0.0247	0.8032 ± 0.0262	0.7781 ± 0.0433	0.8029 ± 0.0268	0.6649 ± 0.0479
Ctscan	0.0623 ± 0.0038	0.7442 ± 0.0180	0.7698 ± 0.0232	0.8104 ± 0.0430	0.4181 ± 0.0430	0.8073 ± 0.0433	**0.3646 ± 0.0350**
Indoorloc	0.0746 ± 0.0030	1.0887 ± 0.0922	0.7326 ± 0.0550	0.1853 ± 0.0326	0.1328 ± 0.0165	0.1845 ± 0.0327	**0.1228 ± 0.0126**
Mv	0.0249 ± 0.0013	0.6962 ± 0.0265	0.4525 ± 0.0213	1.3548 ± 0.0418	0.2011 ± 0.0276	1.3857 ± 0.0450	**0.1547 ± 0.0188**
Pole	0.1409 ± 0.0050	0.6965 ± 0.0589	0.3020 ± 0.0215	0.1950 ± 0.0156	0.1967 ± 0.0187	0.1927 ± 0.0159	**0.1735 ± 0.0119**
Puma32h	0.9930 ± 0.0018	**0.9872 ± 0.0007**	0.9883 ± 0.0015	1.0052 ± 0.0085	0.9974 ± 0.0040	1.0049 ± 0.0084	0.9919 ± 0.0028
Telemonitoring	0.3130 ± 0.0052	0.8724 ± 0.0154	0.8227 ± 0.0247	0.9820 ± 0.0276	0.7564 ± 0.0424	0.9726 ± 0.0270	**0.6741 ± 0.0353**
Average rank	–	3.56	3.56	5.22	2.78	4.44	**1.44**

Kang (2024). On average, our method ranks 2.8 out of the seven methods, over the five datasets. Our method is the best performer for one dataset in terms of RMSE, and three for MAE.

These results are contrary to the method by Lee & Kang (2024) which saw better performance metrics when observing RMSE. This is a potential indication that our method is worse at handling outliers and large deviations in predictions. When looking at table 5, we also observe that while our method performs best for three out of the five datasets, overall, it is still second in terms of average rank. This is because for the datasets pole and telemonitoring, our method not only performs poorly, but is one of the two worst performing methods. It seems that our method is either the best performer, or the worst, with no real in-between. These observations all point to overfitting as a possible explanation. It might be that the addition of validation loss into the weighting mechanism has led to a method that works very well for some datasets with certain characteristics, but badly for others.

We also observe that many of the datasets on which our method performs better, were ones where Oracle exhibited weak performance, and so did the method of Lee & Kang

(2024). For these specific datasets (for example cpusmall and puma32 in terms of MAE, Table 2 and Table 5) our method shows promising improvements. A possible explanation for this could be the fact that our method punishes the poor performance of individual neural networks more heavily. Not only does it penalise networks with high variation in predictions, but it also penalises networks with poor generalisation performance over a validation set. This leads to even smaller weights for these poorly performing networks than what would be assigned by the method of Lee & Kang (2024).

Overall, our method outperforms all the baseline methods except that of Lee & Kang (2024), and seems to perform better on datasets with poor individual network performance.

Table 4: Performance comparison of baseline and proposed methods including our proposed method, in terms of RMSE (mean ± standard deviation).

Dataset	**Mean**	**Median**	**D-SEL(u)**	**D-SEL(u')**	**D-ENS(u)**	**Method Lee & Kang (2024)**	**Novel Proposed Method**
Compactiv	**0.4214 ± 0.0208**	0.4953 ± 0.0202	0.5437 ± 0.0934	0.4778 ± 0.0173	0.5433 ± 0.0936	0.4726 ± 0.0143	0.4250 ± 0.04616
Cpusmall	0.5439 ± 0.0167	0.7868 ± 0.0247	0.8032 ± 0.0262	0.7781 ± 0.0433	0.8029 ± 0.0268	0.6649 ± 0.0479	**0.2214 ± 0.0119**
Pole	0.6965 ± 0.0589	0.3020 ± 0.0215	0.1950 ± 0.0156	0.1967 ± 0.0187	0.1927 ± 0.0159	**0.1735 ± 0.0119**	0.2953 ± 0.0299
Puma32h	**0.9872 ± 0.0007**	0.9883 ± 0.0015	1.0052 ± 0.0085	0.9974 ± 0.0040	1.0049 ± 0.0084	0.9919 ± 0.0028	0.9868 ± 0.0007
Telemonitoring	0.8724 ± 0.0154	0.8227 ± 0.0247	0.9820 ± 0.0276	0.7564 ± 0.0424	0.9726 ± 0.0270	**0.6741 ± 0.0353**	0.8364 ± 0.0230
Average rank	3.20	4.20	6.00	4.00	5.40	**2.40**	2.80

Table 5: Performance comparison of baseline and proposed methods including our proposed method, in terms of MAE (mean ± standard deviation).

Dataset	**Mean**	**Median**	**D-SEL(u)**	**D-SEL(u')**	**D-ENS(u)**	**Method Lee & Kang (2024)**	**Novel Proposed Method**
Compactiv	0.2429 ± 0.0377	0.2440 ± 0.0414	0.2552 ± 0.1412	0.1753 ± 0.0115	0.2550 ± 0.1413	0.1733 ± 0.0110	**0.1580 ± 0.0135**
Cpusmall	0.2076 ± 0.0050	0.2452 ± 0.0039	0.2465 ± 0.0053	0.2470 ± 0.0102	0.2462 ± 0.0054	0.2248 ± 0.0096	**0.1661 ± 0.0083**
Pole	0.5542 ± 0.0491	0.1655 ± 0.0122	0.0814 ± 0.0047	0.0791 ± 0.0056	0.0810 ± 0.0047	**0.0752 ± 0.0045**	0.1438 ± 0.0147
Puma32h	0.7773 ± 0.0005	0.7787 ± 0.0010	0.7888 ± 0.0061	0.7852 ± 0.0026	0.7885 ± 0.0061	0.7797 ± 0.0017	**0.7767 ± 0.0003**
Telemonitoring	0.7050 ± 0.0098	0.6164 ± 0.0244	0.6220 ± 0.0293	**0.4288 ± 0.0322**	0.6182 ± 0.0298	0.4332 ± 0.0264	0.6260 ± 0.0161
Average rank	4.60	4.00	5.80	3.60	4.80	**2.40**	2.80

6 Conclusion

This paper aimed to answer the question "how can the weighting mechanism in dynamic ensemble regression neural networks be improved by incorporating prediction errors?". With an increased need for data security, and an abundance of data-decentralised environments, dynamic ensemble methods provide an interesting solution to the problem of local deployment of neural network models. The weighting mechanism in ensemble neural networks is paramount to their performance. In this paper, a novel weighting mechanism was proposed, incorporating a measure of generalisation performance of individual models into their weighting.

We find that our novel method generally improves upon most existing baseline methods, bar one. Improvements seem to be most prominent in datasets where individual node performance is lacking. We also see some evidence of overfitting due to the inclusion of a validation loss term which incorporates information from the validation set into the weighting mechanism. While our method was not the absolute best performing method for ensemble weighting, it did show modest improvements in certain situations. This is evidence that the incorporation of prediction errors into ensemble weighting can definitely yield some improvements in performance, with the potential for much more.

The findings presented in this paper have practical and theoretical implications. Namely, the clear evidence that prediction errors of neural networks on their validation sets constitute valuable information that should be used to improve ensemble weighting. We encourage future research to look for different ways to incorporate the prediction errors into weighting so as to overcome this issue of overfitting. It would also be interesting to look into why the inclusion of prediction loss in weighting improves ensemble performance for datasets where individual nodes typically underperform.

References

Abdar, M., Pourpanah, F., Hussain, S., Rezazadegan, D., Liu, L., Ghavamzadeh, M., ... others (2021). A review of uncertainty quantification in deep learning: Techniques, applications and challenges. *Information fusion*, *76*, 243–297.

Abdi, H. & Williams, L. J. (2010). Principal component analysis. *Wiley interdisciplinary reviews: computational statistics*, *2*(4), 433–459.

Breiman, L. (2001). Random forests. *Machine learning*, *45*, 5–32.

Britto Jr, A. S., Sabourin, R. & Oliveira, L. E. (2014). Dynamic selection of classifiers—a comprehensive review. *Pattern recognition*, *47*(11), 3665–3680.

Derrac, J., Garcia, S., Sanchez, L. & Herrera, F. (2015). Keel data-mining software tool: Data set repository, integration of algorithms and experimental analysis framework. *J. Mult. Valued Logic Soft Comput*, *17*, 255–287.

Dietterich, T. G. (2000). Ensemble methods in machine learning. In *International workshop on multiple classifier systems* (pp. 1–15).

Dua, D., Graff, C. et al. (2017). Uci machine learning repository.

Freund, Y. & Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, *55*(1), 119–139.

Gal, Y. & Ghahramani, Z. (2016). Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning* (pp. 1050–1059).

Kolter, J. Z. & Maloof, M. A. (2007). Dynamic weighted majority: An ensemble method for drifting concepts. *The Journal of Machine Learning Research*, *8*, 2755–2790.

Lee, Y. & Kang, S. (2024, Feb). Dynamic ensemble of regression neural networks based on predictive uncertainty. *Computers amp; Industrial Engineering*, *190*, 110011. doi: 10.1016/j.cie.2024.110011

Opitz, D. & Maclin, R. (1999). Popular ensemble methods: An empirical study. *Journal of artificial intelligence research*, *11*, 169–198.

Prechelt, L. (2002). Early stopping-but when? In *Neural networks: Tricks of the trade* (pp. 55–69). Springer.

Qiao, Z. & Wang, B. (2021). Molten steel temperature prediction in ladle furnace using a dynamic ensemble for regression. *IEEE Access*, *9*, 18855–18866.

Shahhosseini, M., Hu, G. & Pham, H. (2022). Optimizing ensemble weights and hyperparameters of machine learning models for regression problems. *Machine Learning with Applications*, *7*, 100251.

Soares, S. G. & Araújo, R. (2015). A dynamic and on-line ensemble regression for changing environments. *Expert Systems with Applications*, *42*(6), 2935–2948.

Zhang, Z.-L., Chen, Y.-Y., Li, J. & Luo, X.-G. (2019). A distance-based weighting framework for boosting the performance of dynamic ensemble selection. *Information Processing & Management*, *56*(4), 1300–1316.

A Programming code

Code was kindly provided by Lee & Kang (2024), adjusted to include our novel proposed weighting method, and adapted to run on google colab:

```
!apt update && apt install cuda-11-8

import numpy as np
import time
import torch
import torch.nn as nn
from torch.optim import Adam
from sklearn.metrics import mean_squared_error

data_dir = '/content/data/'

class RegDataset:

    def __init__(self, X, Y):
        self.X = X
        self.Y = Y

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        x = torch.from_numpy(self.X[idx]).float()
        y = torch.from_numpy(self.Y[idx]).float()
        return x, y
```

```
class RegNN(nn.Module):

    def __init__(self, dim_x, n_layers = 3, dim_h = 128, activation =
        nn.ReLU(), prob_dropout=0.1):
        super(RegNN, self).__init__()

        layers = [nn.Linear(dim_x, dim_h), activation,
            nn.Dropout(prob_dropout)]
        for _ in range(n_layers-1): layers += [nn.Linear(dim_h, dim_h),
            activation, nn.Dropout(prob_dropout)]
        layers += [nn.Linear(dim_h, 1)]

        self.predict = nn.Sequential(*layers)

    def forward(self, x):
        y_hat = self.predict(x)
        return y_hat

class Trainer:

    def __init__(self, net, batch_size, model_path, cuda):
        self.net = net
        self.batch_size = batch_size
        self.model_path = model_path
        self.cuda = cuda
        self.optimizer = None

    def load(self):
        self.net.load_state_dict(torch.load(self.model_path, map_location =
            torch.device('cuda')))

    def training(self, train_loader, val_loader, patience = 20, max_epochs =
        500): # patience = 20 , max_epoch = 500
        loss_fn = nn.MSELoss()
        val_y =
            np.array(val_loader.dataset.dataset.Y)[val_loader.dataset.indices].flatten()
        val_log = np.zeros(max_epochs)
        for epoch in range(max_epochs):
            # training
            self.net.train()
            start_time = time.time()
```

```
            for batchidx, batchdata in enumerate(train_loader):
                batch_x, batch_y = batchdata
                batch_x, batch_y = batch_x.to(self.cuda), batch_y.to(self.cuda)
                batch_y_hat = self.net(batch_x)
                loss = loss_fn(batch_y_hat, batch_y)
                self.optimizer.zero_grad()
                loss.backward()
                self.optimizer.step()

            # validation
            val_y_hat, _ = self.inference(val_loader)
            val_log[epoch] = mean_squared_error(val_y, val_y_hat) ** 0.5

            if np.argmin(val_log[:epoch + 1]) == epoch:
                torch.save(self.net.state_dict(), self.model_path)
            elif np.argmin(val_log[:epoch + 1]) <= epoch - patience:
                best_RMSE = np.min(val_log[:epoch + 1])
                best_epoch = np.argmin(val_log[:epoch + 1])
                break
        print('epoch', epoch)
        self.load()

    def inference(self, test_loader, n_forward_passes = 20):
        self.net.eval()
        for m in self.net.modules():
            if m.__class__.__name__.startswith('Dropout'):
                m.train()

        y_hat_list = []
        for _ in range(n_forward_passes):
            y_hat = []
            with torch.no_grad():
                for _ , batchdata in enumerate(test_loader):
                    batch_x = batchdata[0]
                    batch_x = batch_x.to(self.cuda)
                    batch_y_hat = self.net(batch_x).cpu().numpy()
                    y_hat.append(batch_y_hat)
            y_hat = np.vstack(y_hat).flatten()
            y_hat_list.append(y_hat)

        return np.mean(y_hat_list, 0), np.std(y_hat_list, 0, ddof = 0)
```

```
        !pip install scikit-learn
!pip install torch torchvision

import numpy as np
import csv, sys
import os
from sklearn.cluster import KMeans
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

from sklearn.gaussian_process.kernels import Matern
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error

import torch
from torch.utils.data import DataLoader, random_split
from torch.optim import Adam

import warnings
warnings.filterwarnings("ignore")

for sys_1 in range(0, 20):
    for sys_2 in range(10):

        dname_list =
            ['bikesharing','compactiv','cpusmall','ctscan','indoorloc','mv','pole','puma32','

        def load_data(dname, sort = 'none', n_clusters = 10, cluster_thr =
            0.05, seed = 27407):
            dfile = 'data/%s.csv'%dname
            data = np.genfromtxt(dfile, delimiter=',')
            scaler = StandardScaler()
            data = scaler.fit_transform(data)

            X = data[:,:-1]
            Y = data[:,-1:]

            pca = PCA(np.min([10, X.shape[1]]), svd_solver = 'full')
            scaler2 = StandardScaler()
```

```
    X = scaler2.fit_transform(pca.fit_transform(X))

    dim_x = X.shape[1]
    print('-- dataset: %s, size: %d, n. features: %d'%(dname, len(X),
        dim_x))

    C = KMeans(n_clusters=n_clusters, random_state=seed).fit(X).labels_
    frac_clusters = np.array([np.mean(C==j) for j in range(n_clusters)])
    cluster_list = [j for j in range(n_clusters) if frac_clusters[j] >
        cluster_thr]

    print('cluster list', cluster_list)
    print('# clusters:', len(cluster_list))
    print(np.array([np.sum(C==j) for j in range(n_clusters) if
        frac_clusters[j] > cluster_thr]))

    print(X.shape)

    use_idx = [i for i in range(len(C)) if C[i] in cluster_list]
    X, Y, C = X[use_idx], Y[use_idx], C[use_idx]

    X_trn, X_tst, Y_trn, Y_tst, C_trn, C_tst = train_test_split(X, Y,
        C, test_size=None, train_size=0.8, stratify = C,
        random_state=seed)

    return X_trn, X_tst, Y_trn, Y_tst, C_trn, C_tst, cluster_list

def weight_method_1(x, temperature):
    return ((x)**temperature / ((x)**temperature).sum(axis= 0) )

def weight_method_2(x, temperature):
    return ((np.exp(- temperature / x)) / (np.exp(- temperature /
        x)).sum(axis = 0))

# based softmax (chosen)
def weight_method_3(x, alpha):
    return ((np.exp(x * alpha)) / (np.exp(x * alpha)).sum(axis = 0))

data_id = int(sys_1)
seed = 27407
batch_size = 64
```

```
device = torch.device('cuda')
print(data_id, dname_list[data_id])

X_trn, X_tst, Y_trn, Y_tst, C_trn, C_tst, cluster_list =
    load_data(dname_list[data_id], seed = seed)
dim_x = X_trn.shape[1]
net = [RegNN(dim_x).to(device) for _ in cluster_list]
mean_list = []
std_list = []
meta_list = []
val_losses = []

for i, c in enumerate(cluster_list):

    model_dir = f'./model_parameter/{data_id}'
    os.makedirs(model_dir, exist_ok=True)
    model_path = './model_parameter/%d/model_%s_%d_%d.pt'%(data_id,
        dname_list[data_id] , i, sys_2)
    trainer = Trainer(net[i], batch_size, model_path, device)

    idx_trn_c = (C_trn == c)
    X_trn_c = X_trn[idx_trn_c]
    Y_trn_c = Y_trn[idx_trn_c]

    print(model_path, i, c, len(X_trn_c))

    scalerX = StandardScaler()
    scalerY = StandardScaler()
    trn_set = RegDataset(scalerX.fit_transform(X_trn_c),
        scalerY.fit_transform(Y_trn_c))

    n_trn = int(len(X_trn_c) * 0.8)
    n_val = len(X_trn_c) - n_trn
    trn_subset, val_subset = random_split(trn_set, [n_trn, n_val],
        generator=torch.Generator().manual_seed(seed))

    trn_loader = DataLoader(dataset=trn_subset, batch_size=batch_size,
        shuffle=True, drop_last=True)
    val_loader = DataLoader(dataset=val_subset, batch_size=batch_size,
        shuffle=False)
```

```
    trainer.optimizer = Adam(net[i].parameters(), lr= 1e-4 ,
        weight_decay=1e-5) # lr = 1e-4
    trainer.training(trn_loader, val_loader)
    trainer.load()

    _, Y_val_hat_std = trainer.inference(val_loader)

    # Calculate and store the validation loss for the model
    val_y_hat, _ = trainer.inference(val_loader)
    val_loss =
        mean_squared_error(np.array(val_loader.dataset.dataset.Y)[val_loader.dataset.i
        val_y_hat)
    # Inverse of validation loss
    val_losses.append(1 / val_loss)

    tst_set = RegDataset(scalerX.transform(X_tst),
        scalerY.transform(Y_tst))
    tst_loader = DataLoader(dataset=tst_set, batch_size=batch_size,
        shuffle=False)
    Y_tst_hat_mean, Y_tst_hat_std = trainer.inference(tst_loader)

    mean_list.append(scalerY.inverse_transform(Y_tst_hat_mean.reshape(-1,1)))
    std_list.append(Y_tst_hat_std * scalerY.scale_)
    meta_list.append([scalerY.scale_, np.mean(Y_val_hat_std *
        scalerY.scale_), np.median(Y_val_hat_std * scalerY.scale_),
        np.max(Y_val_hat_std * scalerY.scale_)])
    print(val_losses)
    print(val_loss)

report_RMSE = []
report_MAE = []
Y_tst = Y_tst.flatten()


oracle selection
Y_tst_hat_oracle = np.array([mean_list[cluster_list.index(s)][i] for
    i, s in enumerate(C_tst)]).flatten()
tst_rmse_all = mean_squared_error(Y_tst, Y_tst_hat_oracle) ** 0.5
tst_MAE_all = mean_absolute_error(Y_tst, Y_tst_hat_oracle)
tst_rmse_cluster = [mean_squared_error(Y_tst[C_tst == k],
    Y_tst_hat_oracle[C_tst == k]) ** 0.5 for k in cluster_list]
```

```
report_RMSE.append(tst_rmse_all)
report_MAE.append(tst_MAE_all)

# Calculate weights incorporating validation loss (W Normalisation)
val_losses = np.array(val_losses).reshape(-1, 1)
uncertainties = 1 / np.vstack([a/meta_list[i][1] for i, a in
    enumerate(std_list)])
weights = np.transpose(weight_method_3(uncertainties * val_losses,
    alpha=0.1))

mean_array = np.transpose(np.squeeze(np.array(mean_list)))
Y_tst_hat_dyn_ensemble = np.sum(np.array(weights * mean_array), axis=1)
tst_rmse_all = mean_squared_error(Y_tst, Y_tst_hat_dyn_ensemble) ** 0.5
tst_MAE_all = mean_absolute_error(Y_tst, Y_tst_hat_dyn_ensemble)
report_MAE.append(tst_MAE_all)
report_RMSE.append(tst_rmse_all)

# Novel Proposed: Calculate weights incorporating validation loss
    (With Normalisation uncertainties)
val_losses = np.array(val_losses).reshape(-1, 1)
uncertainties = 1 / np.vstack([a/meta_list[i][1] for i, a in
    enumerate(std_list)])
weights = np.transpose(weight_method_3(uncertainties * val_losses,
    alpha=0.1))

mean_array = np.transpose(np.squeeze(np.array(mean_list)))
Y_tst_hat_dyn_ensemble = np.sum(np.array(weights * mean_array), axis=1)
tst_rmse_all = mean_squared_error(Y_tst, Y_tst_hat_dyn_ensemble) ** 0.5
tst_MAE_all = mean_absolute_error(Y_tst, Y_tst_hat_dyn_ensemble)
report_MAE.append(tst_MAE_all)
report_RMSE.append(tst_rmse_all)


 # ensemble W normalization (proposed Lee/Kang)
weight = np.transpose(weight_method_3((1 /
    np.vstack([a/meta_list[i][1] for i, a in enumerate(std_list)])),
    alpha = 10))
mean_array = np.transpose(np.squeeze(np.array(mean_list)))
Y_tst_hat_dyn_ensemble = np.sum(np.array(weight * mean_array), axis =
    1)
tst_rmse_all = mean_squared_error(Y_tst, Y_tst_hat_dyn_ensemble) ** 0.5
```

```
tst_MAE_all = mean_absolute_error(Y_tst, Y_tst_hat_dyn_ensemble)
report_MAE.append(tst_MAE_all)
report_RMSE.append(tst_rmse_all)

# ensemble W/O normalization
weight = np.transpose(weight_method_3((1 / np.array(std_list)), alpha
    = 10))
weight[weight < 10 ** (-8)] = 0
weight = np.nan_to_num(weight, nan = 1)
if np.sum(weight, axis = 0).any() > 1.1:
    raise ValueError
mean_array = np.transpose(np.squeeze(np.array(mean_list)))
Y_tst_hat_dyn_ensemble = np.sum(np.array(weight * mean_array), axis =
    1)
tst_rmse_all = mean_squared_error(Y_tst, Y_tst_hat_dyn_ensemble) ** 0.5
tst_MAE_all = mean_absolute_error(Y_tst, Y_tst_hat_dyn_ensemble)
report_MAE.append(tst_MAE_all)
report_RMSE.append(tst_rmse_all)

tst_rmse_cluster = [mean_squared_error(Y_tst[C_tst == k],
    Y_tst_hat_dyn_ensemble[C_tst == k]) ** 0.5 for k in cluster_list]
tst_mae_cluster = [mean_absolute_error(Y_tst[C_tst == k],
    Y_tst_hat_dyn_ensemble[C_tst == k]) for k in cluster_list]

## dynamic selection / var mean normalized
selection = np.argmin(np.vstack([a/meta_list[i][1] for i, a in
    enumerate(std_list)]), 0)
Y_tst_hat_dyn = np.array([mean_list[s][i] for i, s in
    enumerate(selection)]).flatten()
tst_rmse_all = mean_squared_error(Y_tst, Y_tst_hat_dyn) ** 0.5
tst_rmse_cluster = [mean_squared_error(Y_tst[C_tst == k],
    Y_tst_hat_dyn[C_tst == k]) ** 0.5 for k in cluster_list]
tst_MAE_all = mean_absolute_error(Y_tst, Y_tst_hat_dyn)
report_MAE.append(tst_MAE_all)
report_RMSE.append(tst_rmse_all)

## dynamic selection / vanila
selection = np.argmin(np.vstack(std_list), 0)
Y_tst_hat_dyn = np.array([mean_list[s][i] for i, s in
    enumerate(selection)]).flatten()
tst_rmse_all = mean_squared_error(Y_tst, Y_tst_hat_dyn) ** 0.5
```

```
tst_rmse_cluster = [mean_squared_error(Y_tst[C_tst == k],
    Y_tst_hat_dyn[C_tst == k]) ** 0.5 for k in cluster_list]
tst_MAE_all = mean_absolute_error(Y_tst, Y_tst_hat_dyn)
report_MAE.append(tst_MAE_all)
report_RMSE.append(tst_rmse_all)

## mean prediction
Y_tst_hat_mean = np.mean(mean_list, 0).flatten()
tst_rmse_all = mean_squared_error(Y_tst, Y_tst_hat_mean) ** 0.5
tst_rmse_cluster = [mean_squared_error(Y_tst[C_tst == k],
    Y_tst_hat_mean[C_tst == k]) ** 0.5 for k in cluster_list]
tst_MAE_all = mean_absolute_error(Y_tst, Y_tst_hat_mean)
report_MAE.append(tst_MAE_all)
report_RMSE.append(tst_rmse_all)

## median prediction
Y_tst_hat_median = np.median(mean_list, 0).flatten()
tst_rmse_all = mean_squared_error(Y_tst, Y_tst_hat_median) ** 0.5
tst_rmse_cluster = [mean_squared_error(Y_tst[C_tst == k],
    Y_tst_hat_median[C_tst == k]) ** 0.5 for k in cluster_list]
tst_MAE_all = mean_absolute_error(Y_tst, Y_tst_hat_median)
report_MAE.append(tst_MAE_all)
report_RMSE.append(tst_rmse_all)

with open('./{}_NEW_DE_wo_RMSE_MINE2.csv'.format(dname_list[data_id]),
    'a') as f:
    wr = csv.writer(f)
    wr.writerow(report_RMSE)

with open('./{}_NEW_DE_wo_MAE_MINE2.csv'.format(dname_list[data_id]),
    'a') as f:
    wr = csv.writer(f)
    wr.writerow(report_MAE)
```