ML models comparison for multivariate regression

Christian Esposito, *Federico Lusiani*

espo.christian@gmail.com, felusiani@gmail.com.

ML course (exam code), Academic Year 2020/2021

Date: 25/01/2021

Type of project: **B** *(MLP, KNN, SVM and ELM)*

Abstract

The project compares the regression with four different algorithms: MLP, KNN, SVM and Extreme Learning Machine. Hyperparameters are optimized with grid search. The following results are found: KNN is simple, fast, and performed better than the rest. MLP performed slightly better than SVM and ELM, but it is orders of magnitude slower. SVM is complex to optimize for hyper-parameters. ELM is relatively more simple and stable to optimize, but performed slightly worse.

1 Introduction

Given a dataset, we try to solve a regression problem exploring the application of different models. The models we have tested are the following four: Feed forward MLP, SVM, KNN, Extreme Learning Machine. For each model, we perform hyper-parameters tuning through grid-search and k-fold cross-validation. We then choose the best performing model obtained as a solution to the regression problem, and test it against new data. This gives us an approximate measure of its performance on data that has not been taken into account in the model selection process. We work on Python3 and use scikit-learn and pytorch libraries.

2 Method

We use Pytorch for the NN implementation and scikit-learn for data pre-processing, validation and other algorithms. When applying the models, we have normalized both the input and output data, in order to have a zero mean and unitary variance. However, the error is always measured in the original (unscaled) output space, both during training and evaluation. We split data in two segments: development set 90% and test set 10%. We use the development set with k-fold ($k = 5$) cross validation for model selection. Once we have selected the model, we test its performance on the test set.

The metrics chosen to evaluate models performances is the Euclidean metric. Given the training set $X = \{x^i\} : x^i \in \mathbb{R}^K : \quad x^i = (x^i_1 ... x^i_K)$,the loss function we will use is the Mean Square Error. They are defined as:

$$MEE := \frac{1}{N}\sum_{i=1}^{N} ||x^i - y^i||_2 \qquad MSE := \frac{1}{N}\sum_{i=1}^{N} ||x^i - y^i||_2^2$$

The MSE is easier to optimize and it's naturally implemented in most of the algorithms, so we choose to use this loss function for training and then report the results with the MEE.

3 Experiments

For each model, we first perform a rough grid search, in order to locate a region of optimality. We then perform a finer grid-search, where we choose appropriate ranges for the hyper-parameters to display their effect on the performance of the model. The best performing values are taken as the final hyper-parameters set for the model. Finally, the best performing model is chosen as a solution to the regression problem, and we measure its performance on the test set, after training it on the whole development set. From now on results, until otherwise specified, will refer to k-fold CV experiments and MEE.

3.1 Hardware resources and computing time

We run all experiments on a virtual machine with 8 Intel Xeon Processor CPU cores at 2GHz. In results of cross validation on the models, the time shown is the total time required to cross-validate the model.

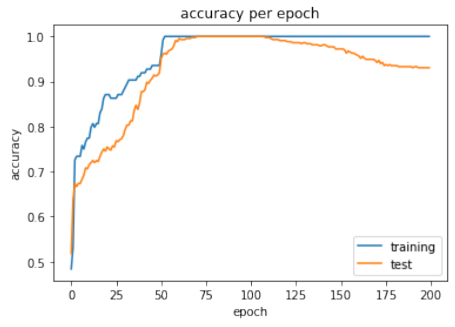
3.2 Monk Results

Pr	Model description	Score
1	'LeakyRelu', Adam optimizer, lr=0.01, β_1 =0.9, β_2 =0.99, NN= (17,4,1)	1.0
2	'LeakyRelu', Adam optimizer, lr=0.01, β_1 =0.9, β_2 =0.99, NN= (17,4,1)	1.0
3	'LeakyRelu', Adam optimizer, lr=0.01, β_1 =0.9, β_2 =0.99, weight_decay=0.003, NN= (17,4,1)	0.972

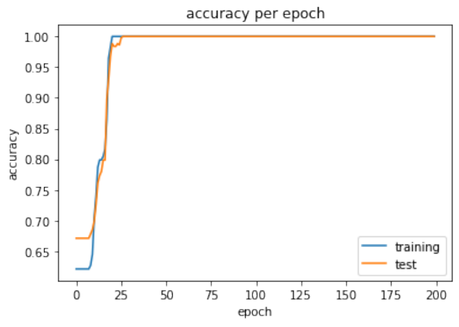
Table 1: MONK results

3.3 Linear Model

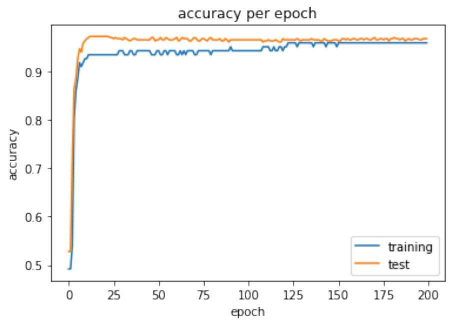
Our very first attempt to solve the regression is a linear model. This gives us a good grasp of problem complexity and an upper bound for the error. The linear regression is performed with scikit learn and is the ridge regression with the penalty term for the weights. The loss function is:



(a) Accuracy M1

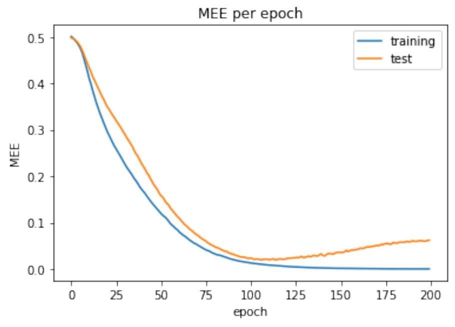


(b) Accuracy M2

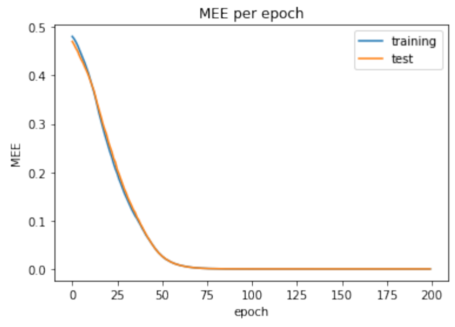


(c) Accuracy M3: Train score is lower because of 6 misclassification in training data

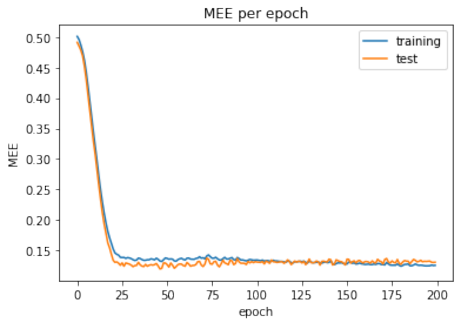
Figure 1: Train and Test accuracy for MONK problems with Pytorch NN



(a) Model loss M1



(b) Model loss M2



(c) Model loss M3

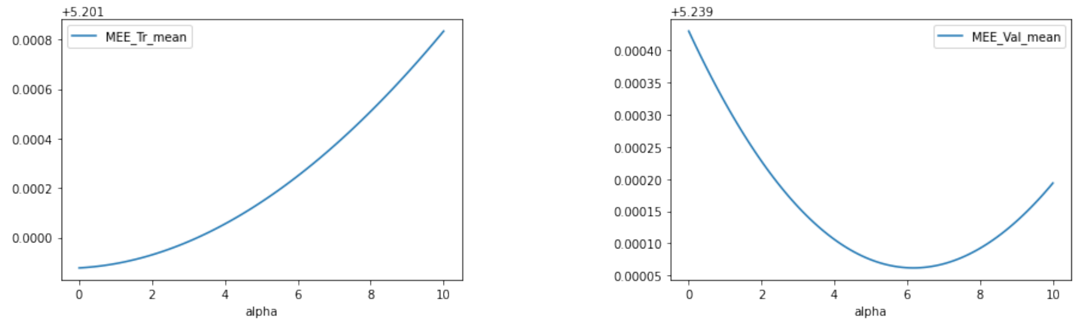
Figure 2: Model loss for MONK Problems

$$min_w ||y - Xw||_2^2 + \alpha ||w||_2^2$$

We rapidly search along α (ref. *LinearModel.ipynb*) and we find out that the best results are for $\alpha \sim 6.16$.

Alpha	**Val MEE mean**	**Val MEE std**	**Tr MEE mean**
6.165	5.239062	0.088420	5.201268

Table 2: Ridge regression results



(a) Linear validation error

(b) Linear training error

Figure 3: Training and validation error of the Ridge regression

3.4 SVM with "RBF" Kernel

We chose to use three different kernels for the SVM regression: RBF, Polynomial and Sigmoid. The last one did not yield good results, hence it will not be described in the following sections. The RBF kernel is defined as $k(x_i, x_j) = e^{\gamma|x_i - x_j|^2}$ First of all we investigate how γ, C, ξ, influence model performance. We extend the search over 8 orders of magnitude from 10^4 to 10^3 for each hp. We report the performances of the model through heatmaps, as we find it provides a clear way to visualize the best hyperparameter regions.

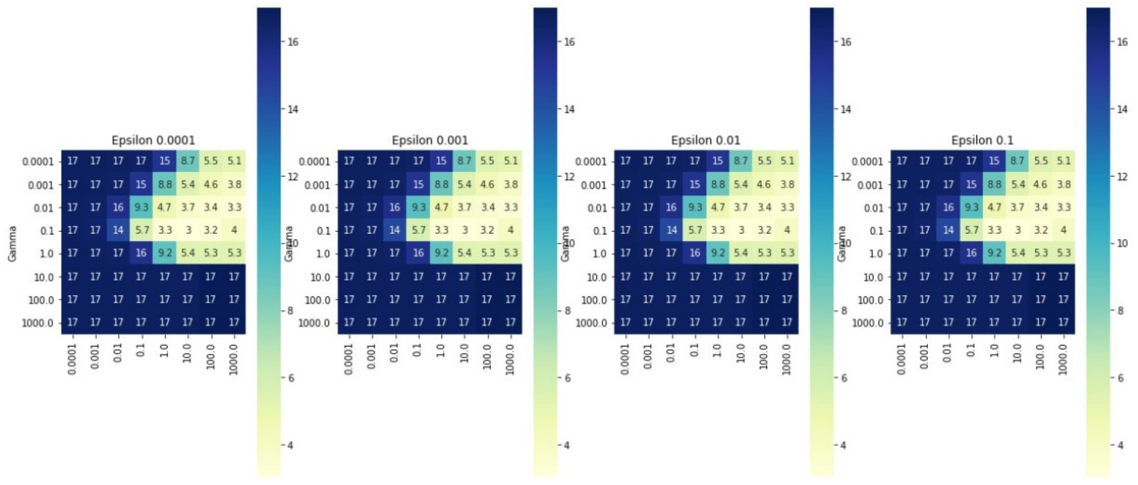


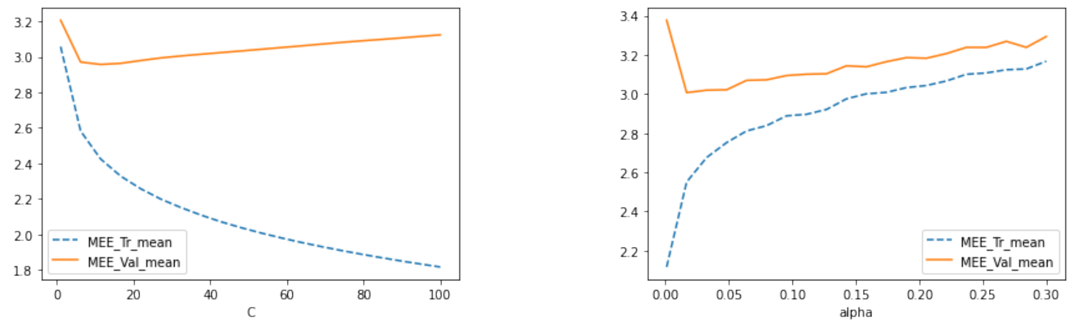
Figure 4: Heatmap for SVM kernel *rbf* and varying ξ

Through the first grid-search we notice that: Performances are badly affected by ξ values that are equal or higher than 10, but is otherwise not sensitive to variation in ξ values; the best performances are achieved around C=10 and γ=0.1. For these reasons we decide to hone the hyperparameter search selecting ξ=0.01 and concentrating on a shorter range of C and γ, centered around the values mentioned above. Table 3 shows the results obtained.

γ C	**0.0001**	**0.00100**	**0.0100**	**0.1000**	**1.0000**
0.1	16.749785	15.635549	9.648565	5.633035	15.372620
1.0	15.612316	9.095410	4.809139	3.206170	8.392088
10.0	9.053809	5.433131	3.698070	2.958995	5.008156
100.0	5.527399	4.644068	3.353219	3.123882	4.947466
1000.0	5.170011	3.849089	3.233564	3.800438	4.947466

Table 3: Refined grid search for SVM with kernel *rbf* showing values of MEE

The best result (lowest MEE, scoring ~2.95) is achieved with C=10, γ=0.1 and ξ=0.01.



(a) SVR *rbf* validation and training error

(b) ELM training and validation error

Figure 5: Training and validation error of SVM and ELM

3.5 SVM with "Poly" Kernel

We then proceed trying the polynomial kernel $(\mathbf{x}^T \cdot \mathbf{x})^p$. Here we need to account for other parameters, such as the grade and the polynomial coefficient. Based on previous results we try ξ=0.01, which was the best value for *rbf*. We proceed with the usual grid search, this time ranging over 4 degrees of the polynomial (1 to 4) and sparse values of C $\in$ [0.1,100]. We note that computing time is more than proportional to the degree, so for higher degree polynomials some kind of approximation method, such as a low-rank approximation to the original kernel matrix, could have been used to reduce the computation time. The first grid search highlights that the results downgrade if C is lower than 1 or higher than 5, so we refine our gridsearch concentrating on those values; meanwhile we keep γ=1, and we vary the polynomial coefficient from 0.5 to 1.5. We also extended the polynomial grade to 5 since it seemed like performances were incrementing with the grade. The results are summarized by the following table and heatmaps.

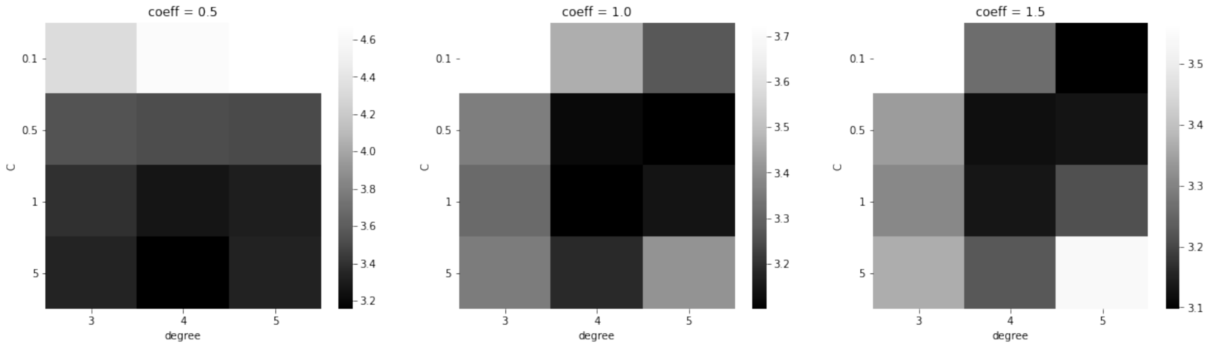


Figure 6: Heatmap for SVM kernel *poly* and varying polynomial coefficient

By analyzing the results we can observe that: there seems to be a correlation between the pc and (C,degree). Moreover we should note that time grows exponentially with C; the best result of the polynomial kernel is comparable to the best result obtained with the RBF kernel, though the RBF kernel provides still better results; the grade 5 doesn't improve the results obtained through the grade 4, hence we do not continue the grid search incrementing the value of the grade (also, it would've proved computationally challenging)

We get the best result with the following combination of hyperparameters: C=0.5, xi=1, d=5, pc=1, γ=1.

3.6 Extreme Learning Machine

The second model we choose is the Extreme Learning Machine[3] , a semi-randomness based method which lies between feed forward neural networks and feature space methods. We investigate time performance and accuracy with respect to SVM and MLP. For this algorithm we use fully connected NN, implemented with pytorch. The linear regression is performed with scikit learn and is the ridge regression, as described in section 3.1.
The reasons behind this choice is explained in [1]. We focus on three very important HPs which are:

- Network Architecture, characterized by one or two Hidden Layers maximum with an "high" number of neurons.
- Activation Function: we use Sigmoid and LeakyRelu
- α used for the Ridge penalty term

We have found that using more than one layer does not seem to decrease the error of the model, while requiring more time to train. We therefore decide to experiment using only one hidden layer. After a first grid search, we have found that the two activation functions worked best with very different ranges of values for α, and also the effect of hidden units was very different. For the LeakyRelu activation function, we have found that good values for α are in the range $\alpha \in$ [0.01, 0.05, 0.1, 0.5], with a peak of performance for the value of 0.05. The performance increases with the number of neural units. However, increasing the number of neural units also increases the computational cost of the model. For this reason, we have not tried ELM architectures with more than 8000 neural units, with the explored range being [1000, 2000, 5000, 8000]. The top 4 performing models are shown in Table 4. When using Sigmoid as an activation function, the hyper-parameters search

n units	α	**MEE_mean**	**MEE_std**	**time (s)**
8000	0.050000	3.019734	0.155371	10.162159
5000	0.050000	3.040678	0.156197	10.324484
8000	0.010000	3.050389	0.137124	10.198512
2000	0.050000	3.067192	0.167496	6.346691

Table 4: Top 4 performing ELM models with LeakyRely as activation function

becomes more complex. Firstly, increasing the number of neural units doesn't seem to bring an improvement in the performance. For this reason, we have explored a different range of neural units: [500, 1000, 2000, 4000]. The optimal number of neural units seems to be inversely correlated to the value of α (see Figure 5). Also notice that the range of values for is very different from the one that resulted optimal for LeakyRely, being $\alpha \in$ [0.000001,0.000005,0.00001,0.00005]. The top 4 performing models are shown in Table 5. Looking at Table 4, we can see that models with LeakyRely as activation function perform better.

Size	**alpha**	**MEE_mean**	**MEE_std**	**time (s)**
500	0.000050	3.303561	0.133421	2.614035
1000	0.000010	3.309781	0.134735	4.646328
2000	0.000005	3.311260	0.132773	6.366117
4000	0.000001	3.315052	0.132618	8.065913

Table 5: Top 4 performing ELM models with Sigmoid as activation function.

size	**alpha**	**activation_f**	**MEE_Val_mean**	**MEE_Val_std**	**MEE_Tr_mean**	**time**
8000	0.016737	LeakyRelu	3.007069	0.131338	2.550472	15.29

Table 6: Final ELM model.

3.7 Neural Networks

For the MLP model, we have used the pytorch python library, implementing our own training loop, in order to correctly take into account the scaling of the output (see Methods

above). We have decided to use the Adam optimizer, which is generally considered well-performing even without proper hyper-parameters tuning [2]. For this reason, we decide to tune only the learning rate η, keeping β_1=0.9 and β_2=0.99. This has the advantage to reduce noticeably the number of MLP models to train, which is fundamental, considering that MLP is much slower to train compared to the other models.
For the training we used minibatch (MB) with size 32. The activation function is LeakyRelu.
For regularization, we apply Early Stopping: the training stops either because an improvement has not been found in the last 20 epochs, or because 500 epochs have been reached (failure to converge).
We have decided to keep the number of units per hidden layer constant. Therefore, each architecture is defined as "LxN" (L number of hidden layers, and N number of units per hidden layer).
Each network architecture is tested with three values of η [2.00E-02, 1.00E-02, 1.00E-03], with 1.00E-02 proving to be optimal in almost all cases. For bigger values of L, we have chosen to explore smaller values of N. In fact, if we kept N fixed as we increase L, we would have reached a number of hidden units disproportionate to the number of training samples models, which makes the model prone to overfitting (aside from taking unfeasible long times to train). This is supported by the results we have obtained, as the optimal number for N becomes smaller as L increases. The results are shown in Table 7,8,9, 10.

LR	**4x10**	**4x15**	**4x25**	**4x50**	**5x10**	**5x15**	**5x25**	**5x50**
2,00E-02	3,016257	3,055002	3,061258	3,012552	3,048661	2,989739	3,053807	3,034202
1,00E-02	3,035271	2,958287	3,000726	3,060434	3,002359	2,937914	2,982229	3,006267
1,00E-03	3,099102	3,068318	3,053878	3,078885	3,064699	2,983944	2,984846	3,021418

Table 7: Mean MEE Val. Error for different MLP Architectures and learning rates

LR	**6x10**	**6x15**	**6x25**	**6x50**
2,00E-02	3,043029	3,104561	3,043777	3,019132
1,00E-02	2,932165	3,039037	2,957049	3,000497
1,00E-03	3,114484	3,102977	2,986966	2,985802

Table 8: Mean MEE Val. Error for different MLP Architectures and learning rates

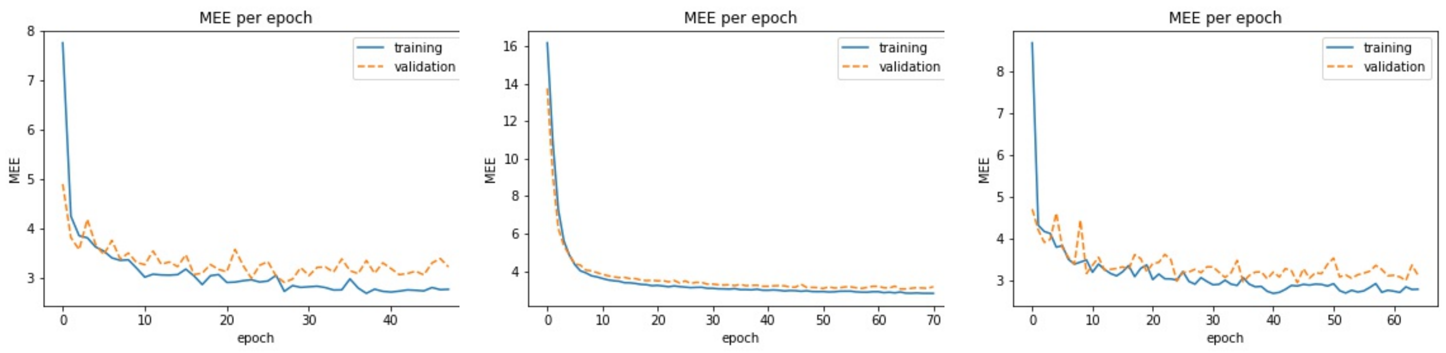
LR	**3x25**	**3x50**	**3x80**	**3x100**	**2x25**	**2x50**	**2x80**	**2x100**	**2x150**
2,00E-02	3,072714	3,050954	3,032807	3,063063	3,01568	3,072003	3,055449	3,070301	3,065065
1,00E-02	3,015933	2,998649	3,060876	3,019994	3,04891	3,012351	3,012317	3,017681	3,057486
1,00E-03	3,072927	3,061761	3,091016	3,073457	3,179805	3,139962	3,117993	3,151567	3,112522

Table 9: Mean MEE Val. Error for different MLP Architectures and learning rates

Nodes	LR	MEE_mean	MEE_std	time (s)
6x10	1.0E-02	2,932165	0,20404	24,420202
5x15	1.0E-02	2,937914	0,151128	17,642732
6x25	1.0E-02	2,957049	0,135452	17,425248
4x15	**1.0E-02**	**2,958287**	**0,146356**	**5,118133**
5x25	1.0E-02	2,982229	0,164257	16,872215
5x15	1.0E-03	2,983944	0,200041	16,90943

Table 10: Best performing NN

The top 6 performing models are shown in table 10. As we can see, they all perform very similarly (the differences in the mean validation MEE are about one-tenth of the standard deviation measured). For this reason, we decide to also consider the dimensions of the model (favouring smaller models), and take 4x15 as the best MLP model. In fig 7 we see the learning curves of the model for the three values of η.



(a) $\eta = 10^{-2}$ (b) $\eta = 10^{-3}$ (c) $\eta = 2 * 10^{-2}$

Figure 7: Learning curves of the final MLP model (4x15) for the three values of η

3.8 KNN

The final model we test is KNN. We test K over the [1,39] interval, using both the uniform and distance weighting function. Since the distance weighting function performs always better, we discard the uniform one. The results are shown in fig 8.

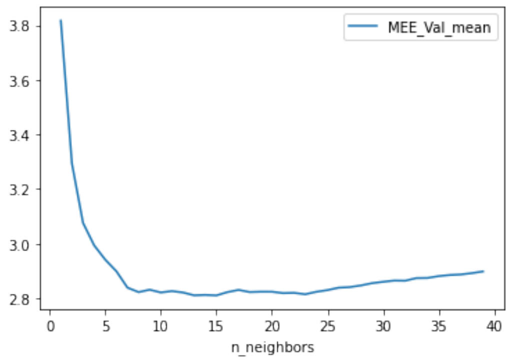


Figure 8: Learning curve KNN Model

Note that using the distance weighting function, the training error is always zero. The most performing model is for K=15, with a mean validation MEE of $\sim$ 2.81 and a std of $\sim$ 0.1, scoring better than all the previous models. For this reason, we have decided to take it as our final model. Training it over the entirety of the development set, yields a mean MEE of $\sim$ 2.95 on the unseen test set. Considering the relatively small size of our test set (10% of the total starting data), and the measured std in the cross validation, we find it a reasonable shift in the measurement of the model performance.

K	**mean VAL MEE**	**mean TR MEE**
15	2.81038	0

K	**TS MEE**	**TR TIME**
15	2.9515	0.0016

Table 11: Left: Mean validation and training error for final KNN model. Right: Error on test set and training time for final KNN model.

4 Conclusions

After having experimented many different tools, libraries and algorithms we would like to make some considerations. Firstly, writing modular code proved to be time saving, along with proper logging and storing of the experimental data. Then, there was a great difference (albeit only when not considering certain hyperparameters choices, for instance high values of C or grade in the polynomial SVM kernel) in terms of computing time between the different models. NN architectures are very popular as they usually outperform other algorithms, though SVM, ELM and KNN has been more straightforward. In the end, KNN achieved a better performance. We think that this is because the data was not so much complex as noisy, and the NN were not able to disentangle this noise. Last consideration is on how important is to visualize results to understand how the different hyper-parameters affect the model and how they relate to each other.

I/we agree to the disclosure and publication of my name, and of the results with preliminary and final ranking.

References

[1] Guang-Bin Huang. An insight into extreme learning machines: random neurons, random features and kernels. *Cognitive Computation*, 6(3):376–390, 2014.

[2] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[3] Jiexiong Tang, Chenwei Deng, and Guang-Bin Huang. Extreme learning machine for multilayer perceptron. *IEEE transactions on neural networks and learning systems*, 27(4):809–821, 2015.