**Generalized Temperature-dependent Material Models for Compressive Strength of Masonry Using Fire Tests, Statistical Methods and Artificial Intelligence**

Aditya Daware1, M.Z. Naser2,3, Ghada Karaki4

1 Graduate Student, Glenn Department of Civil Engineering, Clemson University, Clemson, SC, USA

E-mail: [adaware@g.clemson.edu](mailto:adaware@g.clemson.edu)

2Assistant Professor, Glenn Department of Civil Engineering, Clemson University, Clemson, SC, USA

3AI Research Institute for Science and Engineering (AIRISE), Clemson University, Clemson, SC, USA

E-mail: [mznaser@clemson.edu](mailto:mznaser@clemson.edu), Website: [www.mznaser.com](http://www.mznaser.com)

4Assistant Professor, Department of Civil and Environmental Engineering, Birzeit University, Birzeit, Palestine

E-mail: [gkaraki@birzeit.edu](mailto:gkaraki@birzeit.edu)

# Abstract

Masonry has superior fire resistance properties stemming from its inert characteristics, and slow degradation of mechanical properties. However, once exposed to fire conditions, masonry undergoes a series of physio-chemical changes. Such changes are often described via temperature-dependent material models. Despite calls for standardization of such models, we continue to lack such standardized models. As a result, available temperature-dependent material models vary across various fire codes and standards. In order to bridge this knowledge gap, this paper presents three methodologies, namely, regression-based, probabilistic-based, and the use of artificial neural (ANN) networks, to derive generalized temperature-dependent material models for masonry with a case study on the compressive strength property. Findings from this paper can be adopted to establish updated temperature-dependent material models of fire design and analysis of masonry structures.

*Keywords:* Masonry; Fire; Mechanical properties; Material models.

# Introduction

Masonry has been favored for use in civil and historical structures [1,2]. Such structures are exposed to a variety of hazards (such as fire), and hence it is of utmost importance for engineers to be able to trace the fire response of such structures. This can be attained by properly describing the behavior of masonry under elevated temperatures. For such descriptions to be derived, fire tests are often undertaken [3]. In these tests, small-sized specimens are prepared and examined in a steady-state, or transient-state manner to identify how fire testing influences the examined property (say compressive strength) as a function of the testing procedure (i.e., temperature rise, loading conditions, etc.). Unfortunately, little has been conducted on this front [4–7].

On a more positive note, the open literature delivers successful works that examined the fire performance of large scale masonry components [5,8,9]. In these tests, masonry components (i.e., walls) were tested under standard fire conditions (i.e., ASTM E119 [10], ISO834 [11]) to investigate the integrity, insulation, and load bearing capacity of the tested components [12,13]. On the one hand, such tests are expensive and require sophisticated equipment, and on the other, the same tests provide us with an overall view of the structural performance of the tested components. In a way, these tests implicitly describe the material response under fire. To explicitly arrive at such a description, small scale material tests are often carried out.

The material behavior of masonry and masonry derivatives was explored under fire conditions over the past years. For example, Marco et al. [14] investigated the influence of aggregate properties and mixture design on the mechanical degradation of over 100 masonry specimens in the temperature range of 25-700°C. A similar experimental campaign was also carried out by Khaliq and Bashir [15]. These researchers explored fire performance of burnt masonry units to obtain a temperature-dependent material model. Parallel tests were adopted to guide the development of the masonry material model adopted by Eurocode 6 [16] and continues to be unchanged for over 15 years. Other tests were also duly noted and can be found elsewhere [17–20].

A look into the cited works shows that we lack a standardized testing procedure that ensures reproducibility of results, and most importantly, a fair comparison between the attained models. Generally speaking, researchers devise testing procedures that fit the availability of testing equipment. Testing regimes (e.g., heating rate, holding temperature, etc.) are also primarily cherry-picked. The above complicates the fire design procedure and hinders engineers from properly accounting for fire effects in their designs [21]. The lack of a standard model cast doubt on reviewing and peer-reviewing fire-related designs and pauses collaborations across designers from various geographical backgrounds.

For example, say that a designer is tasked with developing a finite element (FE) model to predict fire response of a masonry wall. This designer will be in need to input a material model to supplement for material degradation within the FE simulation. A question arises, which material model should this designer adopt? Noting how fire response is heavily influenced by the selected material model, then the same designer may obtain a series of predicted responses belonging to different material models that are, in theory, proper. To arrive at a comfortable level of confidence, the above practice is best accomplished with a fire test. As one can see, the lack of standard material model adds further complication to an already delicate matter. Recent works by our group note that it is possible to attain 15-25% variability in fire response of identical structural members given different temperature-dependent material model input [21].

Other questions also arise, what justifies the adoption of the particular model over others? And, how can we ensure that a given model is representative of the material to be used in construction? As one can see, a range of replies can be used to answer the above questions – whether from a codal provision standpoint, or the acceptance of inherent variability within our design process. However, the intention behind these questions is to be highlight the need to have a set of models that are accepted to capture a general and holistic look into material response under fire conditions. Hence, such models can be used in our design process as a steppingstone to allow us to attain a more likely and predictive response of the structural component on hand. This is the main motivation behind this work.

The lack of guidance on this front is natural and is hoped to be overcome in the years to come. Until then, there exists a knowledge gap that merits investigation. This work hopes to narrow this knowledge gap by exploring three different methods to derive generalized material models for masonry, or other construction materials). The explore methods combine traditional and advanced data analytics, namely, regression-based, probabilistic-based, and the use of artificial neural (ANN) networks. These methods were first used to arrive at an updated material model for masonry with a case study on compressive strength. Then, the individual models were compared between all methods for completion.

# Description and Results of Previous Experimental Programs

The behavior of masonry under elevated temperatures is primarily a function of mix design constituents (type of aggregate, binder type, water content etc.) [14,15,17,18,20,22–27]. For completion, this section starts with a review of available material models covering the compressive strength of masonry, and a complete review of other properties can be found elsewhere [19,28].

The compressive strength property is traditionally obtained vis small scale tests conducted under elevated temperatures. The obtained strength is then divided by the original strength at ambient temperature to arrive at a degradation factor. For example, the reduction factor (*fc,400°C*/*fc,25°C*) reflect the adverse change in this property at a target temperature of 400°C (i.e., *fc,400°C*) to that at ambient temperature (*fc,25°C*). As mentioned earlier, there is a lack of standard testing fire procedures, and hence a variety of testing procedures were developed as described in the open literature [14,15,17,18,20,22–27]. Results from these tests are presented in Fig. 1 and results on masonry and its derivates are combined into a database. This figure shows a general trend of degradation with scatter (due to differences in mixture design, heating history, etc.) of all models beyond ambient temperature. The same figure also shows that the degradation of masonry is much slower than concrete.

Figure 1 Degradation in compressive strength of masonry under elevated temperatures

As one can see in Fig. 1, the compressive strength of masonry degrades with the rise in temperature. For example, Ayala and Bailey [17] show the degradation in lightweight (LW) masonry follows a linear trend until reaching 75% at 400°C. The degradation slows down until 600°C, and then continues to degrade at a larger pace beyond this temperature. Similarly, Khaliq and Bashir [15] tested burnt bricks under compression and noted an almost linear degradation throughout the 20oC to 800oC temperature range. The degradation was estimated at 22% at 400°C and 32% at 600°C. This reduction was reported to be related to the formation of mechanical cracks as a result of series of physio-chemical changes in microstructure arising from mineralogical transformations. While the collected data plotted in Fig. 1 shows similar trends of reduction along with the whole range of testing, Andreini et al. [14] reported a different behavior at 500°C. In this behavior, the compressive strength recovers until reaching 600oC. The same behavior can be related to the composition of the tested masonry or to the use of cylindrical specimens – see [14] for more details. For brevity, other material models were not discussed herein. Still, it is worth noting, a complete discussion on the above observations, together with those related to other masonry properties under elevated temperature, can be found in our earlier work [28].

# Description of Methodology

To echo our previously noted motivation behind this work, we aim to develop a generalized material model for the degradation of the compressive strength property of masonry. In this effort, our general procedure includes the following three steps: 1) collecting data points from the aforenoted literature review on temperature-dependent material models for common masonry materials with a special emphasis on compressive strength property *under elevated temperatures*, 2) classify all the values in terms of reduction factors corresponding to target temperatures at intervals of 100°C, and 3) apply the proposed three methodologies to analyze the data and arrive at a generalized model. It should be noted that there are limitations of fire tests on masonry, therefore, all data were combined on masonry (including brick units) into one database. The applied methodologies include arithmetic “mean” methods, Probabilistic-based method, and ANN as described in the following subsections.

## The Mean Method

The first adopted method is based on the simple arithmetic mean method. This method had been used in publications by international building committees for its simplicity. In this method, the arithmetic mean for all available compressive strength degradation factors values at each target temperature for masonry is calculated. Then, the calculated “mean” values were fitted, as one can see in Fig. 2. Figure 2 shows a linear-like and minor degradation in the strength property that reaches around 10% at 300°C. This degradation then slightly accelerates until reaching 600°C at which point the mean calculated degradation factor is 80.4%. From then, the degradation picks up and continues further until reaching 39.5% at 800°C.

Figure 2 Compressive Strength reduction factors with respect to temperature using the arithmetic mean method

## The Probabilistic Method

Limited research works derived temperature-dependent probabilistic models for the construction material properties, i.e., models for normal-strength concrete, mild steel, and insulating materials, can be found in [29–32]. The approach in this paper aims to develop temperature-dependent probabilistic models for the compressive strength of masonry. It comprises the following steps:

1) Survey and collect experimental data.

2) Select a set of distribution functions; the collected experimental data is fitted to this set of distribution functions. As the data size was small (five data points at every temperature), normal and lognormal (basic distribution functions) were chosen.

3) check the goodness of fit; the best model is the one that delivers an acceptable description of the data while using a minimum number of parameters. The most common likelihood criteria for model selection are the Akaike Information Criterion (AIC), [33]. The model with a smaller value of *AIC* estimatoris considered the one with the lowest information loss. Furthermore, *AIC*values can be transformed to conditional probabilities for each model; these probabilities refer to Akaike weights [34], equation 1.

Eq. 1

where *AIC* is the difference between the *AIC*of the ith model and the minimum value of *AIC* for all candidate models, and *K* is the number of all candidate models. Akaike weight (*wi*) represents the probability that the ith model has the lowest information loss given the data, and the other candidate models examined [34]. The models with high values of Akaike weights are considered the best candidates to represent the data sets.

4) Develop a regression model for the parameters defining the best-fit distribution function. Polynomial functions were used to derive relations for the parameters defining the selected distribution function as a continuous function of temperature. The coefficient of determination R2 was used to check the regression model quality and prevent underfitting the data.

*Implementation:* the data points for the compressive strength of masonry were fitted at every temperature point to normal and lognormal, as explained earlier. An overall *AIC* measure for the candidate model was then calculated as the sum of *AIC* estimators at the examined temperatures. Akaike weights, which present the conditional probability that the candidate model describes the data with the lowest data loss, were calculated. Table 1 shows AIC estimators and Akaike weights for data sets and examined distribution functions. Following the data analysis in Table 1, the best-fit model is the normal distribution, the regression models for the mean (**) and standard deviation (**) of normal distribution were then developed and documented in Eq. 2.

Table 1: The *AIC* estimators and Akaike weights for the fitted distribution functions of compressive strength for masonry

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Distribution | *AICi* | *wi* | Distribution | *AICi* | *wi* |
| Normal | -110.13 | **0.64** | Lognormal | -109.00 | 0.36 |

The coefficient of determination for the regression models of ** and ** are 0.98 and 0.84, respectively.

Eq 2.a

Eq 2.b

*Tstd* is the standardized temperature, i.e. , where *Ti* is the examined temperature point, *Tmin* is the minimum temperature in data set, and *Tmax* is the maximum temperature in the data set. By using the standardized temperatures’, the determination of regression coefficients was more stable. Figure 3 shows the 5%, 50%, and 95% percentiles of the samples created using the developed probabilistic model.

1. Comparison between normal and log normal distributions
2. Comparison between quantiles

Figure 3 Compressive Strength reduction factors with respect to temperature using Bayesian Method [Note: Grey and Green series in the bottom figure represent 5% and 95% confidence intervals].

## The Artificial Neural Network Method

ANNs can be used in two primary tasks: classification and regression. Classification is the way to classify the input data in a discrete set or a categorical set. On the other hand, regression is the process of fitting a function that maps the inputs to a given phenomenon which suits the motivation behind this study. Thus, the results of fire tested collected from experimental studies noted above are used as training data for the ANN. This ANN is developed to predict the generalized temperature-dependent material model for the compressive strength of masonry. For the purpose of developing an ANN, the open-source data analytics Python programming language is used.

An ANN mimics the way the human brain processes and evaluates information. Due to its own makeup (i.e., topology), an ANN can learn patterns with datasets. Each ANN has a series of layers (just like the human brain) with neuron (e.g., processing units) links connecting such layers to process information. The first layer receives the input data, and sends them to the next (hidden) layer. The hidden layer processes such data to identify patterns within data which is then delivered via the output layer for visualization. Figure 4 below shows a typical structure of a neural network. Please note that our ANN has two inputs, reduction factors and accompanying temperatures, as well as the same for the output. Given the exploratory notion behind using ANN as a method to derive a material model, we utilized default values used in the Python ANN code (which is also appended in the appendix). We invite future works to explore the influence of varying the key parameters within ANNs to seek their effect on the obtained results.

Diagram, schematic

Description automatically generated

Figure 4 Typical ANN Structure

Unlike traditionally applied ANNs, the developed ANN does not follow a training/validation/testing setup. This is due to the fact that for validation and testing, a known value of a “ground truth” output is needed to compare ANN’s predictions against the “ground truth”. This brings a complication to the user, wherein a ground truth value (or material model) is needed to be identified. However, such a model does not exist yet. Hence, we opted to use all of the available data to train the ANN. Then, the output of the ANN (after undergoing data processing and transformation) is assumed to be the generalized “holistic” material model.

For this research, all the data points available were trained into ANN. The selected ANN structure had the batch size of 60 per iteration and a maximum number of 53 neurons in three hidden layers. These layers establish a connection between the data set provided at the starting of the training phase. The ANN is programmed to run the different types of combinations up to 1000 iterations. Each iteration runs all the data through each neuron to obtain the less possible error values. After analyzing every connection between the provided data, ANN enters into the prediction phase to arrive at the generalized material model. Then, a question arises is to how to ensure the validity of such model.

The aforenoted question can be answered via two checks: 1) by comparing the created model to those derived by other researchers to allow for a “similarity” comparison (i.e., the ANN model shows the degradation of strength and not an unjustified behavior of strength recovering – which was not seen in most works (see Fig. 1), and 2) by comparing the ANN model to the generalized models obtained via statistical and Bayesian methods (which were arrived at via well-established statistical theories).

# Comparison of Generalized Models

All the derived models from the different methods were compared (see Fig. 5). Overall, the comparison of these models shows a close agreement between the derived models, as well as newly conducted fire experiments by the authors [35]. Thus, it can safely be assumed that the three used methodologies converge and do yield comparable outcomes – irrespective of their complexities. This opens up the door for designers to apply any or all of these methodologies to arrive at generalized material models. Future works are invited to continue exploring the above three methodologies, as well as others, to hope of deriving modern and generalized material models.

Fig. 5 Comparison between the derived models via the different methodologies [Note: Shaded area reflects the scatter in available reported data not undertaken by the authors]

Overall, the following two phases were identified to be common across all models: 1) gradual decrease in compressive strength till 400oC will almost no loss till 200oC, and 2) beyond 400oC, rapid reduction in compressive strength till 800oC was observed. The first trend of loss in compressive strength from ambient temperature to 400oC was observed across the different analysis methods with a maximum deviation of 6%. The reduction in strength till 400oC was slightly lower in the case of the probabilistic model.

Furthermore, the degradation in compressive strength from ambient to 200oC predicted by the ANN was 6% that of strength at ambient. The difference in compressive strength at 400oC predicted by ANN and by the experimental investigation was observed to be 8%. On the other hand, a notable reduction by 36% is predicted compared to only 16% loss as noted in experimental testing at 600oC. After 600oC, very sudden decrease of compressive strength was observed up to 800oC. Compared to 68% loss observed in the experimental study, ANN predicted 57% loss to 800oC.

A similar trend was also observed between the three methodologies for temperature ranges from 400oC to 800oC. The rate of reduction in compressive strength increases significantly as temperature increases above 600oC. Less than 50% of the original strength was retained by the CMUs when calculated by each method. This shows fair agreement between the results from various analysis methods and can be validated with the help of experimental results. A small behavior was noticed in the case of the arithmetic mean method between 500-600oC in which there seems to be a slight re-gaining of strength. Such behavior can be attributed to the fact that some of the reported data from fire tests as obtained from the open literature also showed a similar behavior 1, which helped push the arithmetic mean to lower values (see Fig. 1).

# Further Insights into Findings from this work

A closer look into Fig. 5 also shows that all generalized models (including the one derived from ANN) fit within the scatter of the reported experimental studies. This observation acts as external validation. In addition, the same figure and scatter also show that the largest scatter between the developed model occurs between the temperature range of 400-700°C (which also mirrors that observed by the reported tests wherein the largest discrepancy is noted at that particular region). Off all three models, the Bayesian model seems to nicely represent the bulk of the data – owing to its intricate procedure. On the other hand, one can see that the same can also be said to the model obtained by the much simpler mean procedure. The ANN model, while it fits within the scattered data, seems to be skewed towards a more conservative (i.e., high degradation) reduction in strength. Finally, one can also see that the scatter within the generalized models is much smaller than that in the experimental results obtained from our review. We speculate that a true representation of a standard model is likely to lie somewhere between the newly derived models.

At this point in time, and while the above results show that any of the three models can be used as a generalized model to represent the degradation in compressive strength of masonry, the authors feel that is appropriate to prioritize the Bayesian model and the statistical model. Future works that leverage ANN are invited to dig deeper into the mechanisms of ANN to better understand the effects of ANN tuning parameters on the validity of the derived material models. Once such understanding is developed, then future material models can be easily updated. Away from the numerical and analytical investigations, future works are also invited to prioritize developed standardized testing procedures that will allow engineers to overcome the lack of guidance, and by extension, to move past the “generalized” material models toward “standardized material models”.

# Conclusions

This paper showcases the application of three different methodologies to derive generalized material models with a case study on the compressive strength of masonry. Findings from this work echo the need to establish up-to-date standardized testing procedures to create a modern representation of material behavior under fire conditions. These representations (i.e., material models) will come in handy in design and modeling situations. The following list of inferences can also be drawn from this study:

* There is a persistent lack of commonly accepted testing procedures, leading to the rise of researcher-derived testing methods.
* The disparity in existing material models could be smoothened by implementing statistical or ANN methods to develop more generalized temperature-dependent property models.
* Despite differences in all three methods, namely, regression-based, probabilistic-based, and the use of artificial neural networks (ANN), used in our analysis, these methods converge and yield comparable results (i.e., generalized material models).
* Masonry tends to degrade to reach 82-91% and 70-84% of initial strength at 400°C and 600°C, respectively.

# Data Availability

Some or all data, models, or code that support the findings of this study are available from the corresponding author upon reasonable request.

# Acknowledgment

The authors would like to acknowledge and thank the support provided by the National Concrete Masonry Association (NCMA) under grant no. 2020.010.

# References

[1] Y. Praticò, J. Ochsendorf, S. Holzer, R.J. Flatt, Post-fire restoration of historic buildings and implications for Notre-Dame de Paris, Nat. Mater. 18 (2020) 817–820. https://doi.org/10.1038/s41563-020-0748-y.

[2] A. Arêde, C. Almeida, C. Costa, A. Costa, In-situ and lab tests for mechanical characterization of stone masonry historical structures, Constr. Build. Mater. 220 (2019) 503–15. https://doi.org/10.1016/j.conbuildmat.2019.06.039.

[3] V. Kodur, M.Z. Naser, Structural fire engineering, Professional, McGraw Hill, 2020.

[4] T.D. Nguyen, F. Meftah, Behavior of clay hollow-brick masonry walls during fire. Part 1: Experimental analysis, Fire Saf. J. 52 (2012) 55–64. https://doi.org/10.1016/j.firesaf.2012.06.001.

[5] A. Nadjai, M. O’Gara, F. Ali, R. Jurgen, Compartment masonry walls in fire situations, Fire Technol. (2006). https://doi.org/10.1007/s10694-006-7509-6.

[6] M.Z. Naser, Observational Analysis of Fire-Induced Spalling of Concrete through Ensemble Machine Learning and Surrogate Modeling, J. Mater. Civ. Eng. 33 (2021) 04020428. https://doi.org/10.1061/(ASCE)MT.1943-5533.0003525.

[7] M.Z. Naser, Autonomous Fire Resistance Evaluation, ASCE Jounral Struct. Eng. 146 (2020). https://doi.org/10.1061/(ASCE)ST.1943-541X.0002641.

[8] H. Foster, E. Pinkston, S. Ingberg, Fire Resistance of Walls of Gravel-Aggregate Concrete Masonry Units, 1951. https://doi.org/https://www.govinfo.gov/content/pkg/GOVPUB-C13-2e4ec5d1d7161cf03003ab0009057f1c/pdf/GOVPUB-C13-2e4ec5d1d7161cf03003ab0009057f1c.pdf.

[9] S.H. Ingberg, Fire resistance tests of building materials and constructions, Saf. Eng. (1929).

[10] ASTM, E119-16 - Standard Test Methods for Fire Tests of Building Construction and Materials, Am. Soc. Test. Mater. (2016). https://doi.org/10.1520/E0119-10B.1.2.

[11] ISO 834-1999, Fire resistance tests — Elements of building construction, ISO, Int. Org. Stand., 1999.

[12] A.J. O’Meagher, I.D. Bennetts, Modelling of concrete walls in fire, Fire Saf. J. 17 (1991) 315–335. https://doi.org/10.1016/0379-7112(91)90026-U.

[13] BSI, BSI Standards Publication Fire resistance tests Part 1 : General Requirements, (2012).

[14] M. Andreini, A. De Falco, M. Sassu, Experiences and Analysis on Masonry Materials Subjected to High Temperatures, 15th Int. Brick Block Mason. Conf. (2012).

[15] W. Khaliq, M.F. Bashir, High temperature mechanical and material properties of burnt masonry bricks, Mater. Struct. Constr. 49 (2016) 5195–5208. https://doi.org/10.1617/s11527-016-0854-0.

[16] European Committee for Standardization, EN 1996-1-2: Design of masonry structures - Part 1-2: General rules - Structural fire design, 2005.

[17] R. Ayala, F. Rene, C. Bailey, Mechanical properties and structural behaviour of masonry at elevated temperatures, 2011.

[18] M. Andreini, A. De Falco, M. Sassu, Stress-strain curves for masonry materials exposed to fire action, Fire Saf. J. 69 (2014) 43–56. https://doi.org/10.1016/j.firesaf.2014.08.005.

[19] S. Russo, F. Sciarretta, Masonry exposed to high temperatures: Mechanical behaviour and properties - An overview, Fire Saf. J. 55 (2013) 69–86. https://doi.org/10.1016/j.firesaf.2012.10.001.

[20] S. Russo, F. Sciarretta, Experimental and Theoretical Investigation on Masonry after High Temperature Exposure, Exp. Mech. 52 (2012) 341–359. https://doi.org/10.1007/s11340-011-9493-0.

[21] M.Z. Naser, Deriving temperature-dependent material models for structural steel through artificial intelligence, Constr. Build. Mater. 191 (2018) 56–68. https://doi.org/10.1016/J.CONBUILDMAT.2018.09.186.

[22] M. Andreini, M. Sassu, Mechanical behaviour of full unit masonry panels under fire action, Fire Saf. J. 46 (2011) 440–450. https://doi.org/10.1016/j.firesaf.2011.07.004.

[23] G.L. Bai, N.J. Du, Y.Z. Xu, C.G. Qin, Study on the Thermal Properties of Hollow Shale Blocks as Self-Insulating Wall Materials, Adv. Mater. Sci. Eng. (2017). https://doi.org/10.1155/2017/9432145.

[24] M. Madrid, A. Orbe, H. Carré, Y. García, Thermal performance of sawdust and lime-mud concrete masonry units, Constr. Build. Mater. 169 (2018) 113–123. https://doi.org/10.1016/j.conbuildmat.2018.02.193.

[25] T.Z. Harmathy, Experimental study on moisture and fire endurance, Fire Technol. 2 (1966) 52–59. https://doi.org/10.1007/BF02588965.

[26] Z. Xiao, T.C. Ling, C.S. Poon, S.C. Kou, Q. Wang, R. Huang, Properties of partition wall blocks prepared with high percentages of recycled clay brick after exposure to elevated temperatures, Constr. Build. Mater. 49 (2013) 56–61. https://doi.org/10.1016/j.conbuildmat.2013.08.004.

[27] J. Bošnjak, S. Gambarelli, A. Sharma, A. Mešković, Experimental and numerical studies on masonry after exposure to elevated temperatures, Constr. Build. Mater. 230 (2020). https://doi.org/10.1016/j.conbuildmat.2019.116926.

[28] A. Daware, M.Z. Naser, Fire performance of masonry under various testing methods, Constr. Build. Mater. (2021). https://doi.org/10.1016/j.conbuildmat.2021.123183.

[29] N. Elhami Khorasani, T. Gernay, C. Fang, Parametric Study for Performance-Based Fire Design of US Prototype Composite Floor Systems, J. Struct. Eng. (United States). (2019). https://doi.org/10.1061/(ASCE)ST.1943-541X.0002315.

[30] R. Qureshi, S. Ni, N. Elhami Khorasani, R. Van Coile, D. Hopkin, T. Gernay, Probabilistic Models for Temperature-Dependent Strength of Steel and Concrete, J. Struct. Eng. 146 (2020) 04020102. https://doi.org/10.1061/(asce)st.1943-541x.0002621.

[31] B. Jovanović, N.E. Khorasani, T. Thienpont, R.K. Chaudhary, R. Van Coile, Probabilistic models for thermal properties of concrete, (2020). https://doi.org/10.14264/363ff91.

[32] G. Karaki, R.A. Hawileh, V.K.R. Kodur, Probabilistic-Based Approach for Evaluating the Thermal Response of Concrete Slabs under Fire Loading, J. Struct. Eng. 147 (2021) (ASCE)ST.1943-541X.0003039. https://doi.org/10.1061/(ASCE)ST.1943-541X.0003039.

[33] P. Stoica, Y. Selen, Model-order selection, IEEE Signal Process. Mag. 21 (2004) 36–47. https://doi.org/10.1109/MSP.2004.1311138.

[34] E.J. Wagenmakers, S. Farrell, AIC model selection using Akaike weights, Psychon. Bull. Rev. 11 (2004) 192–196. https://doi.org/10.3758/BF03206482.

[35] A. Daware, Deriving Generalized Temperature-dependent Material Models for Masonry through Fire Tests and Machine Learning, Clemson University, 2021.

# Appendix

# coding: utf-8

from \_\_future\_\_ import print\_function

# In[1]:

import warnings

warnings.filterwarnings('ignore')

warnings.simplefilter("ignore")

import os

import numpy as np

import tensorflow as tf

import random as rn

os.environ['KMP\_DUPLICATE\_LIB\_OK']='True'

from keras import regularizers

from keras.datasets import mnist

from keras.layers import Dense, Dropout

from keras.models import Sequential

from keras.optimizers import RMSprop

from pandas.plotting import scatter\_matrix

from pprint import pprint

from sklearn import linear\_model

from sklearn import preprocessing

from sklearn import svm

from sklearn.cluster import KMeans

from sklearn.decomposition import PCA

from sklearn.ensemble import RandomForestRegressor

from sklearn.ensemble import RandomForestClassifier

from sklearn.feature\_selection import f\_regression

from sklearn.feature\_selection import SelectFromModel

from sklearn.feature\_selection import SelectKBest

from sklearn.gaussian\_process.kernels import WhiteKernel, ExpSineSquared

from sklearn.isotonic import IsotonicRegression

from sklearn.kernel\_ridge import KernelRidge

from sklearn.linear\_model import LinearRegression

from sklearn.linear\_model import RidgeClassifier

from sklearn.linear\_model import SGDRegressor

from sklearn.metrics import mean\_squared\_error

from sklearn.metrics import mean\_absolute\_error

from sklearn.metrics import r2\_score

from sklearn.model\_selection import cross\_val\_predict

from sklearn.model\_selection import cross\_val\_score

from sklearn.model\_selection import RandomizedSearchCV

from sklearn.neighbors import KNeighborsRegressor

from sklearn.neighbors import KNeighborsClassifier

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import MinMaxScaler

from sklearn.preprocessing import Normalizer

from sklearn.preprocessing import StandardScaler

from keras.wrappers.scikit\_learn import KerasClassifier

from sklearn.model\_selection import KFold

from sklearn.preprocessing import LabelEncoder

import configparser

import csv

import datetime

import glob

import keras

import math

import matplotlib

#matplotlib.use("TkAgg")

from matplotlib import pyplot as plt

import os

import pandas as pd

import pickle

import random

import sys

import time

import traceback

from minepy import MINE

from sklearn.linear\_model import LogisticRegression

from os import path

from pathlib import PurePath

from tensorflow.python.util import deprecation

import tensorflow as tf

import multiprocessing

deprecation.\_PRINT\_DEPRECATION\_WARNINGS = False

config = tf.ConfigProto()

#config.gpu\_options.per\_process\_gpu\_memory\_fraction = 0.5 # maximun alloc gpu50% of MEM

#config.gpu\_options.allow\_growth = True #allocate dynamically

#config = tf.ConfigProto( device\_count = {'GPU': 2 , 'CPU':4 } )

sess = tf.Session(config=config)

keras.backend.set\_session(sess)

#os.environ['TF\_CPP\_MIN\_LOG\_LEVEL'] = '3'

os.environ['TF\_CPP\_MIN\_VLOG\_LEVEL'] = '2'

def save\_test\_data(predictions, actual\_values, filename):

df\_predictions = pd.DataFrame(predictions, columns=['predictions'])

df\_actual\_values = pd.DataFrame(actual\_values, columns=['actual\_values'])

df = df\_predictions.join(df\_actual\_values)

df.to\_csv(filename)

# In[3]:

def load\_header(csv\_file, print\_out = False):

# Loading headers

headers = pd.read\_csv(csv\_file,header=None, nrows=1).values[0]

idx\_to\_key = {}

key\_to\_idx = {}

for i in range(0, len(headers)):

idx = i

key = headers[i]

key\_to\_idx[key] = idx

idx\_to\_key[idx] = key

if print\_out is True:

print(idx\_to\_key)

return headers, idx\_to\_key, key\_to\_idx

def model\_name(model\_abbr):

model\_name = None

if(model\_abbr =='RF'):

model\_name = 'Random Forest'

elif(model\_abbr =='NET'):

model\_name = 'Neural Network'

elif(model\_abbr =='LR'):

model\_name = 'Linear Regression'

elif(model\_abbr =='LRC'):

model\_name = 'Logistic Regression'

elif(model\_abbr =='RG'):

model\_name = 'Ridge'

elif(model\_abbr =='KR'):

model\_name = 'Kernel Ridge'

elif(model\_abbr =='BR'):

model\_name = 'Bayesian Ridge'

elif(model\_abbr =='SVM'):

model\_name = 'Support Vector Machine'

elif(model\_abbr =='NN'):

model\_name = 'k-Nearest Neighbor'

return model\_name

# In[4]:

def data\_load\_shuffle(csv\_file, input\_col, cols\_to\_remove, target\_col, random\_state=0, delimiter = ',', map\_all = None):

data = pd.read\_csv(csv\_file, delimiter = delimiter)

data\_df = data[data[target\_col].notnull()]

if cols\_to\_remove is not None:

for col in cols\_to\_remove:

del data\_df[col]

data\_df\_shuffle = data\_df.sample(frac=1, random\_state=random\_state)

if input\_col is not None:

data\_df\_shuffle = data\_df\_shuffle[input\_col+[target\_col]]

data\_df = data\_df[input\_col+[target\_col]]

y\_train = pd.DataFrame(data\_df\_shuffle[target\_col])

# training set is without target column

del data\_df\_shuffle[target\_col]

x\_train = data\_df\_shuffle.copy()

headers, idx\_to\_key, key\_to\_idx = load\_header(csv\_file)

cols\_to\_remove\_ = []

headers = list(headers)

header\_y = target\_col

if cols\_to\_remove is not None:

for col in cols\_to\_remove:

headers.remove(col)

headers.remove(target\_col)

# dataframe to numpy array

x\_train\_original = x\_train

y\_train = y\_train.values

x\_train = x\_train.values

# reshaping target values

y\_train = y\_train.reshape(y\_train.shape[0],1)

if input\_col is not None:

header\_x = np.array(input\_col)

else:

header\_x = np.array(headers)

key\_idx = -1

if map\_all is not None:

for key in map\_all.keys():

for i in range(0, len(header\_x)):

if header\_x[i]==key:

key\_idx = i

break

if key\_idx!=-1:

for item in x\_train:

item[key\_idx] = map\_all[key][item[key\_idx]]

if map\_all is not None:

for key in map\_all.keys():

if header\_y==key:

new\_y\_train = []

for item in y\_train:

item = [map\_all[key][item[0]]]

new\_y\_train.append(item)

y\_train = np.array(new\_y\_train)

for key in map\_all.keys():

data\_df[key] = data\_df[key].map(map\_all[key])

x\_train = x\_train.astype('float32')

y\_train = y\_train.astype('float32')

return data\_df, x\_train, y\_train, header\_x, header\_y

def correlation\_analysis\_all(data\_df, target\_col, top\_k=10, file\_to\_save = None, save\_chart = None, only\_pcc= False, feature\_selection\_file = None):

if feature\_selection\_file == None:

print("\* correlation\_analysis\_all")

pcc = data\_df.corr()[target\_col]

if(len(pcc)<top\_k):

top\_k=len(pcc)

print("Computing PCC, PCC\_SQRT ..")

pcc = pcc.sort\_values(ascending = False).dropna()

pcc = pcc.rename("PCC")

try:

del pcc[target\_col]

except:

pass

pcc\_sqrt = pcc.apply(lambda x: np.sqrt(x\* x))

pcc\_sqrt = pcc\_sqrt.sort\_values(ascending = False).dropna()

pcc\_sqrt = pcc\_sqrt.rename("PCC\_SQRT")

MICs = []

MASs = []

MEVs = []

MCNs = []

MCN\_generals = []

GMICs = []

TICs = []

print("Computing all other metrics ..")

if only\_pcc==False or only\_pcc=='False':

for col in data\_df.columns:

print(" - computing for ", col, "...")

if col!=target\_col:

x = data\_df[col].values

y = data\_df[target\_col].values

mine = MINE()

mine.compute\_score(x,y)

MICs.append((col,mine.mic()))

MASs.append((col,mine.mas()))

MEVs.append((col,mine.mev()))

MCNs.append((col,mine.mcn(0)))

MCN\_generals.append((col,mine.mcn\_general()))

GMICs.append((col,mine.gmic()))

TICs.append((col,mine.tic()))

top\_k\_pcc = list(pcc.keys())[:top\_k]

top\_k\_pcc\_sqrt = list(pcc\_sqrt.keys())[:top\_k]

top\_k\_mic = [tup[0] for tup in sorted(MICs, key=lambda tup: tup[1], reverse = True)[:top\_k]]

top\_k\_mas = [tup[0] for tup in sorted(MASs, key=lambda tup: tup[1], reverse = True)[:top\_k]]

top\_k\_mev = [tup[0] for tup in sorted(MEVs, key=lambda tup: tup[1], reverse = True)[:top\_k]]

top\_k\_mcn = [tup[0] for tup in sorted(MCNs, key=lambda tup: tup[1], reverse = True)[:top\_k]]

top\_k\_mcn\_general = [tup[0] for tup in sorted(MCN\_generals, key=lambda tup: tup[1], reverse = True)[:top\_k]]

top\_k\_gmic = [tup[0] for tup in sorted(GMICs, key=lambda tup: tup[1], reverse = True)[:top\_k]]

top\_k\_tic = [tup[0] for tup in sorted(TICs, key=lambda tup: tup[1], reverse = True)[:top\_k]]

mic\_df = pd.DataFrame([tup[1] for tup in MICs],columns=['MIC'],index=[tup[0] for tup in MICs])

mas\_df = pd.DataFrame([tup[1] for tup in MASs],columns=['MAS'],index=[tup[0] for tup in MASs])

mev\_df = pd.DataFrame([tup[1] for tup in MEVs],columns=['MEV'],index=[tup[0] for tup in MEVs])

mcn\_df = pd.DataFrame([tup[1] for tup in MCNs],columns=['MCN'],index=[tup[0] for tup in MCNs])

mcn\_general\_df = pd.DataFrame([tup[1] for tup in MCN\_generals],columns=['MCN\_general'],index=[tup[0] for tup in MCN\_generals])

gmic\_df = pd.DataFrame([tup[1] for tup in GMICs],columns=['GMIC'],index=[tup[0] for tup in GMICs])

tic\_df = pd.DataFrame([tup[1] for tup in TICs],columns=['TIC'],index=[tup[0] for tup in TICs])

if only\_pcc==False or only\_pcc=='False':

final\_report = mic\_df.join(mas\_df).join(mev\_df).join(mcn\_df).join(mcn\_general\_df).join(gmic\_df).join(tic\_df).sort\_index().join(pcc\_sqrt).join(pcc)

else:

pcc\_sqrt = pd.DataFrame(pcc\_sqrt)

final\_report = pcc\_sqrt.join(pcc)

if file\_to\_save is not None:

# save to correlation report

final\_report.to\_csv(file\_to\_save)

if save\_chart is not None:

for col in final\_report.keys():

ax = final\_report[col].sort\_values(ascending=False).plot(kind='bar',alpha=0.8)

ax.set\_ylabel(col+" (target\_col = '"+target\_col+"')", fontsize=12)

plt.axhline(0, color='k')

plt.savefig(save\_chart)

plt.close()

fs\_dict = {'PCC':top\_k\_pcc,'PCC\_SQRT':top\_k\_pcc\_sqrt,'MIC':top\_k\_mic,'MAS':top\_k\_mas,'MEV':top\_k\_mev,'MCN':top\_k\_mcn,'MCN\_general':top\_k\_mcn\_general,'GMIC':top\_k\_gmic,'TIC':top\_k\_tic}

else:

final\_report = pd.read\_csv(feature\_selection\_file)

top\_k\_pcc = list(final\_report.sort\_values(by=['PCC'], ascending=False).T.values[0])[:top\_k]

top\_k\_pcc\_sqrt = list(final\_report.sort\_values(by=['PCC\_SQRT'], ascending=False).T.values[0])[:top\_k]

top\_k\_mic = list(final\_report.sort\_values(by=['MIC'], ascending=False).T.values[0])[:top\_k]

top\_k\_mas = list(final\_report.sort\_values(by=['MAS'], ascending=False).T.values[0])[:top\_k]

top\_k\_mev = list(final\_report.sort\_values(by=['MEV'], ascending=False).T.values[0])[:top\_k]

top\_k\_mcn = list(final\_report.sort\_values(by=['MCN'], ascending=False).T.values[0])[:top\_k]

top\_k\_mcn\_general = list(final\_report.sort\_values(by=['MCN\_general'], ascending=False).T.values[0])[:top\_k]

top\_k\_gmic = list(final\_report.sort\_values(by=['GMIC'], ascending=False).T.values[0])[:top\_k]

top\_k\_tic = list(final\_report.sort\_values(by=['TIC'], ascending=False).T.values[0])[:top\_k]

fs\_dict = {'PCC':top\_k\_pcc,'PCC\_SQRT':top\_k\_pcc\_sqrt,'MIC':top\_k\_mic,'MAS':top\_k\_mas,'MEV':top\_k\_mev,'MCN':top\_k\_mcn,'MCN\_general':top\_k\_mcn\_general,'GMIC':top\_k\_gmic,'TIC':top\_k\_tic}

if file\_to\_save is not None:

# save to correlation report

final\_report.to\_csv(file\_to\_save)

if save\_chart is not None:

for col in final\_report.keys()[1:]:

ax = final\_report[col].sort\_values(ascending=False).plot(kind='bar',alpha=0.8)

ax.set\_ylabel(col+" (target\_col = '"+target\_col+"')", fontsize=12)

plt.axhline(0, color='k')

plt.savefig(save\_chart)

plt.close()

return fs\_dict, final\_report

# In[6]:

def default\_model\_parameters\_classifier():

model\_parameters = {

'scaler\_option':'StandardScaler', \

'rf\_n\_estimators': '100', 'rf\_max\_features': 'auto', 'rf\_max\_depth': 'None', \

'rf\_min\_samples\_split': '2', 'rf\_min\_samples\_leaf': '1', 'rf\_bootstrap': 'True', \

'rf\_criterion':'gini','rf\_min\_weight\_fraction\_leaf':'0.','rf\_max\_leaf\_nodes':'None',\

'rf\_min\_impurity\_decrease':'0.',\

'nn\_n\_neighbors': '5', 'nn\_weights': 'uniform', 'nn\_algorithm': 'auto', 'nn\_leaf\_size': '30', 'nn\_p': '2',\

'nn\_metric':'minkowski','nn\_metric\_params':'None',

'rg\_alpha':'1','rg\_fit\_intercept':'True','rg\_normalize':'False','rg\_max\_iter':'None','rg\_tol':'0.001','rg\_class\_weight':'None','rg\_solver':'auto','svm\_kernel': 'rbf', \

'svm\_degree': '3', 'svm\_coef0': '0.0', 'svm\_tol': '1e-3', 'svm\_c': '1.0', \

'svm\_gamma': 'auto', \

'svm\_decision\_function\_shape':'ovr', \

'net\_structure':'16 16 16',\

'net\_layer\_n':'3',\

'net\_dropout': '0.0',\

'net\_l\_2': '0.01',\

'net\_learning\_rate': '0.01',\

'net\_epochs': '100',\

'net\_batch\_size': '2',\

}

return model\_parameters

def default\_model\_parameters():

model\_parameters = {

'scaler\_option':'StandardScaler', \

'rf\_n\_estimators': '100', 'rf\_max\_features': 'auto', 'rf\_max\_depth': 'None', \

'rf\_min\_samples\_split': '2', 'rf\_min\_samples\_leaf': '1', 'rf\_bootstrap': 'True', \

'rf\_criterion':'mse','rf\_min\_weight\_fraction\_leaf':'0.','rf\_max\_leaf\_nodes':'None',\

'rf\_min\_impurity\_decrease':'0.',\

'nn\_n\_neighbors': '5', 'nn\_weights': 'uniform', 'nn\_algorithm': 'auto', 'nn\_leaf\_size': '30', 'nn\_p': '2',\

'nn\_metric':'minkowski','nn\_metric\_params':'None',\

'kr\_alpha': '1', 'kr\_kernel': 'linear', 'kr\_gamma': 'None', 'kr\_degree': '3', 'kr\_coef0': '1', \

'br\_n\_iter': '300', 'br\_alpha\_1': '1.2e-6', 'br\_alpha\_2': '1.e-6', 'br\_tol': '1.e-3', \

'br\_lambda\_1': '1.e-6', 'br\_lambda\_2': '1.e-6', 'br\_compute\_score': 'False', 'br\_fit\_intercept': 'True', \

'br\_normalize':'False',\

'svm\_kernel': 'rbf', \

'svm\_degree': '3', 'svm\_coef0': '0.0', 'svm\_tol': '1e-3', 'svm\_c': '1.0', \

'svm\_epsilon': '0.1', 'svm\_shrinking': 'True', 'svm\_gamma': 'auto', \

'net\_structure':'16 16 16',\

'net\_layer\_n':'3',\

'net\_dropout': '0.0',\

'net\_l\_2': '0.01',\

'net\_learning\_rate': '0.01',\

'net\_epochs': '100',\

'net\_batch\_size': '2',\

}

return model\_parameters

def load\_model\_parameter\_from\_file(filename):

config = configparser.RawConfigParser()

config.read(filename)

model\_parameters = {}

for key in config['HYPERPARAMETERS']:

model\_parameters[key] = config["HYPERPARAMETERS"][key]

return model\_parameters

# In[7]:

def fix\_value(val, val\_type):

if val is None or val=='None':

return None

elif val=='auto':

return val

else:

if(val\_type=='float'):

return float(val)

elif(val\_type=='str'):

return str(val)

elif(val\_type=='int'):

return int(val)

elif(val\_type=='bool'):

return str2bool(val)

elif val\_type=='PurePath':

return PurePath(val)

else:

return val

def define\_model\_classifier(model\_type, model\_parameters, x\_header\_size, random\_state = None):

if model\_type == "LRC":

model = Pipeline([

('classification', LogisticRegression())

])

elif model\_type == "RF":

model = Pipeline([

('classification', RandomForestClassifier(n\_estimators = int(model\_parameters['rf\_n\_estimators']),

max\_features = fix\_value(model\_parameters['rf\_max\_features'],'int'),

max\_depth = fix\_value(model\_parameters['rf\_max\_depth'],'int'),

min\_samples\_split = int(model\_parameters['rf\_min\_samples\_split']),

min\_samples\_leaf = int(model\_parameters['rf\_min\_samples\_leaf']),

bootstrap = str2bool(model\_parameters['rf\_bootstrap']),

criterion = model\_parameters['rf\_criterion'],

random\_state = random\_state,

min\_weight\_fraction\_leaf = float(model\_parameters['rf\_min\_weight\_fraction\_leaf']),

max\_leaf\_nodes = fix\_value(model\_parameters['rf\_max\_leaf\_nodes'],'int'),

min\_impurity\_decrease = float(model\_parameters['rf\_min\_impurity\_decrease']),

))])

elif model\_type == "NN":

model = Pipeline([

('classification', KNeighborsClassifier(n\_neighbors = int(model\_parameters['nn\_n\_neighbors']),

weights = model\_parameters['nn\_weights'],

algorithm = model\_parameters['nn\_algorithm'],

leaf\_size = int(model\_parameters['nn\_leaf\_size']),

metric = model\_parameters['nn\_metric'],

metric\_params = fix\_value(model\_parameters['nn\_metric\_params'],'str'),

p = int(model\_parameters['nn\_p'])))

])

elif model\_type == "RG":

model = Pipeline([

('classification', RidgeClassifier(alpha = float(model\_parameters['rg\_alpha']),

fit\_intercept = fix\_value(model\_parameters['rg\_fit\_intercept'],'bool'),

normalize = fix\_value(model\_parameters['rg\_normalize'],'bool'),

max\_iter = fix\_value(model\_parameters['rg\_max\_iter'],'int'),

tol = float(model\_parameters['rg\_tol']),

class\_weight = fix\_value(model\_parameters['rg\_class\_weight'],'str'),

solver = fix\_value(model\_parameters['rg\_solver'],'str')))

])

elif model\_type == "SVM":

model = Pipeline([

('classification', svm.SVC(kernel = model\_parameters['svm\_kernel'],

degree = int(model\_parameters['svm\_degree']),

coef0 = float(model\_parameters['svm\_coef0']),

tol = float(model\_parameters['svm\_tol']),

C = float(model\_parameters['svm\_c']),

gamma = fix\_value(model\_parameters['svm\_gamma'],'float'),

decision\_function\_shape = model\_parameters['svm\_decision\_function\_shape']

))

])

return model

def define\_model\_regression(model\_type, model\_parameters, x\_header\_size, random\_state = None):

if model\_type == "LR":

model = Pipeline([

('regression', LinearRegression())

])

elif model\_type == "RF":

model = Pipeline([

('regression', RandomForestRegressor(n\_estimators = int(model\_parameters['rf\_n\_estimators']),

max\_features = fix\_value(model\_parameters['rf\_max\_features'],'int'),

max\_depth = fix\_value(model\_parameters['rf\_max\_depth'],'int'),

min\_samples\_split = int(model\_parameters['rf\_min\_samples\_split']),

min\_samples\_leaf = int(model\_parameters['rf\_min\_samples\_leaf']),

bootstrap = str2bool(model\_parameters['rf\_bootstrap']),

criterion = model\_parameters['rf\_criterion'], random\_state = random\_state,

min\_weight\_fraction\_leaf = float(model\_parameters['rf\_min\_weight\_fraction\_leaf']),

max\_leaf\_nodes = fix\_value(model\_parameters['rf\_max\_leaf\_nodes'],'int'),

min\_impurity\_decrease = float(model\_parameters['rf\_min\_impurity\_decrease']),

))])

elif model\_type == "NN":

model = Pipeline([

('regression', KNeighborsRegressor(n\_neighbors = int(model\_parameters['nn\_n\_neighbors']),

weights = model\_parameters['nn\_weights'],

algorithm = model\_parameters['nn\_algorithm'],

leaf\_size = int(model\_parameters['nn\_leaf\_size']),

metric = model\_parameters['nn\_metric'],

metric\_params = fix\_value(model\_parameters['nn\_metric\_params'],'str'),

p = int(model\_parameters['nn\_p'])))

])

elif model\_type == "BR":

model = Pipeline([

('regression', linear\_model.BayesianRidge(n\_iter = int(model\_parameters['br\_n\_iter']),

alpha\_1 = float(model\_parameters['br\_alpha\_1']),

alpha\_2 = float(model\_parameters['br\_alpha\_2']),

tol = float(model\_parameters['br\_tol']),

lambda\_1 = float(model\_parameters['br\_lambda\_1']),

lambda\_2 = float(model\_parameters['br\_lambda\_2']),

compute\_score = fix\_value(model\_parameters['br\_compute\_score'],'bool'),

normalize = fix\_value(model\_parameters['br\_normalize'],'bool'),

fit\_intercept = fix\_value(model\_parameters['br\_fit\_intercept'],'bool')))

])

elif model\_type == "SVM":

model = Pipeline([

('regression', svm.SVR(kernel = model\_parameters['svm\_kernel'],

degree = int(model\_parameters['svm\_degree']),

coef0 = float(model\_parameters['svm\_coef0']),

tol = float(model\_parameters['svm\_tol']),

C = float(model\_parameters['svm\_c']),

gamma = fix\_value(model\_parameters['svm\_gamma'],'float'),

epsilon = float(model\_parameters['svm\_epsilon']),

))

])

# 'kr\_alpha': '1', 'kr\_kernel': 'linear', 'kr\_gamma': 'None', 'kr\_degree': '3', 'kr\_coef0': '1', \

elif model\_type == "KR":

model = Pipeline([

('regression', KernelRidge(alpha = int(model\_parameters['kr\_alpha']),

kernel = fix\_value(model\_parameters['kr\_kernel'],'str'),

gamma = fix\_value(model\_parameters['kr\_gamma'],'str'),

degree = int(model\_parameters['kr\_degree']),

coef0 = int(model\_parameters['kr\_coef0']),))

])

return model

# In[8]:

def rescale\_x(scaler\_option, x\_train):

scale = None

if scaler\_option=='False':

x\_train\_ = x\_train

elif scaler\_option == "MinMaxScaler":

scale = preprocessing.MinMaxScaler()

x\_train\_ = scale.fit\_transform(x\_train)

elif scaler\_option == "MaxAbsScaler":

scale = preprocessing.MaxAbsScaler()

x\_train\_ = scale.fit\_transform(x\_train)

elif scaler\_option == "RobustScaler":

scale = preprocessing.RobustScaler()

x\_train\_ = scale.fit\_transform(x\_train)

elif scaler\_option == "QuantileTransformer":

scale = preprocessing.QuantileTransformer()

x\_train\_ = scale.fit\_transform(x\_train)

elif scaler\_option == "Normalizer":

scale = preprocessing.Normalizer()

x\_train\_ = scale.fit\_transform(x\_train)

else:

scale = preprocessing.StandardScaler()

x\_train\_ = scale.fit\_transform(x\_train)

return x\_train\_, scale

# In[9]:

def str2bool(v):

return v.lower() in ("yes", "true", "t", "1")

def cross\_val\_predict\_net\_classifier(model, x\_train, y\_train, epochs=1000, batch\_size=8, verbose = 0, scaler\_option='StandardScaler', num\_of\_folds = 5, num\_of\_class = 2, force\_to\_proceed = False, accuracy\_threshold = 0.5, fast\_tune = True):

x\_trains, y\_trains, x\_tests, y\_tests = split\_data(x\_train, y\_train, num\_of\_folds=num\_of\_folds)

predictions\_total = []

actual\_values\_total = []

for j in range(0, num\_of\_folds):

print(" Evaluating fold(%d) ..."%(j))

start\_time = time.time()

x\_train\_, scale = rescale\_x(scaler\_option, x\_trains[j])

# This is the change

if scale is not None:

x\_test\_ = scale.transform(x\_tests[j])

else:

x\_test\_ = x\_tests[j]

dummy\_y = keras.utils.to\_categorical(y\_trains[j], num\_classes=num\_of\_class, dtype='float32')

history = model.fit(x\_train\_, dummy\_y,

batch\_size=batch\_size,

epochs=epochs,

verbose=verbose)

predictions = model.predict\_classes(x\_test\_)

actual\_values = y\_tests[j]

actual\_values = actual\_values.reshape(actual\_values.shape[0],)

accuracy= evaluate\_classifier(predictions, actual\_values)

if force\_to\_proceed == False:

if accuracy<accuracy\_threshold:

return [],[]

print(" accuracy = %8.3f "%(accuracy))

predictions\_total+=list(predictions)

actual\_values\_total+=list(actual\_values)

if fast\_tune==True:

print("\* Fast tuning enabled. so we only test 1 fold, and move on ..")

break

return np.array(predictions\_total), np.array(actual\_values\_total)

def cross\_val\_predict\_net(model, x\_train, y\_train, epochs=1000, batch\_size=8, verbose = 0, scaler\_option='StandardScaler', num\_of\_folds = 5, force\_to\_proceed= False, fast\_tune=True):

x\_trains, y\_trains, x\_tests, y\_tests = split\_data(x\_train, y\_train, num\_of\_folds=num\_of\_folds)

predictions\_total = []

actual\_values\_total = []

for j in range(0, num\_of\_folds):

print(" Evaluating fold(%d) ..."%(j))

start\_time = time.time()

x\_train\_, scale = rescale\_x(scaler\_option, x\_trains[j])

# This is the change

if scale is not None:

x\_test\_ = scale.transform(x\_tests[j])

else:

x\_test\_ = x\_tests[j]

history = model.fit(x\_train\_, y\_trains[j],

batch\_size=batch\_size,

epochs=epochs,

verbose=verbose)

predictions = model.predict(x\_test\_)

actual\_values = y\_tests[j]

MAE, R2 = evaluate(predictions, actual\_values)

print(" MAE = %8.3f R2 = %8.3f ..."%(MAE, R2))

if force\_to\_proceed == False:

if R2<0:

return [],[]

predictions\_total+=list(predictions)

actual\_values\_total+=list(actual\_values)

if fast\_tune==True:

print("\* Fast tuning enabled. so we only test 1 fold, and move on ..")

break

return np.array(predictions\_total), np.array(actual\_values\_total)

def save\_parameters(model\_parameters, filename):

f = open(filename,'w')

f.write("[HYPERPARAMETERS]\n\n")

for key in model\_parameters.keys():

f.write(str(key)+" = "+str(model\_parameters[key])+"\n")

f.close()

def save\_args(model\_args,filename):

f = open(filename,'w')

f.write("[ARGUMENTS]\n\n")

for key in model\_args.keys():

f.write(str(key)+" = "+str(model\_args[key])+"\n")

f.close()

def save\_metadata(model\_args, model\_stats,filepath):

if path.exists(filepath):

meta=pd.read\_csv(filepath,index\_col=0)

else:

meta=pd.DataFrame(columns=["session","model\_args","model\_stats"])

size=len(meta.index)+1

meta=meta.append(pd.DataFrame(data=[[size,model\_args,model\_stats]],columns=["session","model\_args","model\_stats"]))

meta.to\_csv(filepath)

return size

def train\_and\_predict(model, x\_train, y\_train, scaler\_option, num\_of\_folds=5):

x\_train\_, scale = rescale\_x(scaler\_option, x\_train)

y\_train\_ = y\_train.reshape(y\_train.shape[0],)

predictions = cross\_val\_predict(model, x\_train\_, y\_train\_, cv=num\_of\_folds)

actual\_values = y\_train\_

return predictions, actual\_values

def get\_session(project\_file):

if path.exists(project\_file / "metadata.csv"):

meta=pd.read\_csv(project\_file / "metadata.csv")

else:

return 1

return len(meta.index)+1

# In[10]:

def train\_and\_save\_net\_classifier(model, tag, input\_cols, target\_col, x\_train, y\_train, scaler\_option, accuracy=None, path\_to\_save = '.', num\_of\_folds=5, epochs=100, batch\_size=2, num\_of\_class = 2):

if accuracy is None:

print('\* Model has not been evaluated. Evaluation initiated via %d-fold cross validation'%(num\_of\_folds))

predictions, actual\_values = cross\_val\_predict\_net\_classifier(model, epochs=epochs, batch\_size=batch\_size, x\_train = x\_train, y\_train = y\_train, verbose = 0, scaler\_option = scaler\_option, num\_of\_folds = num\_of\_folds, num\_of\_class = num\_of\_class, fast\_tune = False)

accuracy = evaluate\_classifier(predictions, actual\_values)

x\_train\_, scale = rescale\_x(scaler\_option, x\_train)

dummy\_y = keras.utils.to\_categorical(y\_train, num\_classes=num\_of\_class, dtype='float32')

print('\* Training initiated ..')

model.fit(x\_train\_, dummy\_y, epochs=epochs, batch\_size=batch\_size)

print('\* Training done.')

model\_dict = {}

model\_dict['tag'] = tag

model\_dict['model'] = model

model\_dict['model\_abbr'] = 'NET'

model\_dict['input\_cols'] = input\_cols

model\_dict['target\_col'] = target\_col

model\_dict['accuracy'] = accuracy

model\_dict['fitted\_scaler\_x'] = scale

output\_file = PurePath(path\_to\_save) / (tag)

#print(model\_dict)

print("\* Trained model saved to file:", str(output\_file))

output = open(output\_file, 'wb')

pickle.dump(model\_dict, output)

def train\_and\_save\_net(model, tag, input\_cols, target\_col, x\_train, y\_train, scaler\_option, MAE=None, R2=None, path\_to\_save = '.', num\_of\_folds=5, epochs=100, batch\_size=2):

if MAE is None or R2 is None:

print('\* Model has not been evaluated. Evaluation initiated via %d-fold cross validation'%(num\_of\_folds))

predictions, actual\_values = cross\_val\_predict\_net(model, epochs=epochs, batch\_size=batch\_size, x\_train = x\_train, y\_train = y\_train, verbose = 0, scaler\_option = scaler\_option, fast\_tune = False)

MAE, R2 = evaluate(predictions, actual\_values)

x\_train\_, scale = rescale\_x(scaler\_option, x\_train)

print('\* Training initiated ..')

model.fit(x\_train\_, y\_train, epochs=epochs, batch\_size=batch\_size)

print('\* Training done.')

model\_dict = {}

model\_dict['tag'] = tag

model\_dict['model'] = model

model\_dict['model\_abbr'] = 'NET'

model\_dict['input\_cols'] = input\_cols

model\_dict['target\_col'] = target\_col

model\_dict['MAE'] = MAE

model\_dict['R2'] = R2

model\_dict['fitted\_scaler\_x'] = scale

output\_file = PurePath(path\_to\_save) / (tag)

#print(model\_dict)

print("\* Trained model saved to file:", str(output\_file))

output = open(output\_file, 'wb')

pickle.dump(model\_dict, output)

def train\_and\_save\_classifier(model, tag, model\_abbr, input\_cols, target\_col, x\_train, y\_train, scaler\_option, accuracy=None, path\_to\_save = '.', num\_of\_folds=5):

x\_train\_, scale = rescale\_x(scaler\_option, x\_train)

y\_train\_ = y\_train.reshape(y\_train.shape[0],)

if accuracy is None:

print('\* Model has not been evaluated. Evaluation initiated via %d-fold cross validation'%(num\_of\_folds))

predictions = cross\_val\_predict(model, x\_train\_, y\_train\_, cv=num\_of\_folds)

actual\_values = y\_train\_

accuracy = evaluate\_classifier(predictions, actual\_values)

print('\* Training initiated ..')

model.fit(x\_train\_, y\_train\_)

print('\* Training done.')

actual\_values = y\_train\_

model\_dict = {}

model\_dict['tag'] = tag

model\_dict['model'] = model

model\_dict['model\_abbr'] = model\_abbr

model\_dict['input\_cols'] = input\_cols

model\_dict['target\_col'] = target\_col

model\_dict['accuracy'] = accuracy

model\_dict['fitted\_scaler\_x'] = scale

output\_file = PurePath(path\_to\_save) / (tag)

#print(model\_dict)

print("\* Trained model saved to file:", str(output\_file))

output = open(output\_file, 'wb')

pickle.dump(model\_dict, output)

def train\_and\_save(model, tag, model\_abbr, input\_cols, target\_col, x\_train, y\_train, scaler\_option, MAE=None, R2=None, path\_to\_save = '.', num\_of\_folds=5):

x\_train\_, scale = rescale\_x(scaler\_option, x\_train)

y\_train\_ = y\_train.reshape(y\_train.shape[0],)

if MAE is None or R2 is None:

print('\* Model has not been evaluated. Evaluation initiated via %d-fold cross validation'%(num\_of\_folds))

predictions = cross\_val\_predict(model, x\_train\_, y\_train\_, cv=num\_of\_folds)

actual\_values = y\_train\_

MAE, R2 = evaluate(predictions, actual\_values)

print('\* Training initiated ..')

model.fit(x\_train\_, y\_train\_)

print('\* Training done.')

actual\_values = y\_train\_

model\_dict = {}

model\_dict['tag'] = tag

model\_dict['model'] = model

model\_dict['model\_abbr'] = model\_abbr

model\_dict['input\_cols'] = input\_cols

model\_dict['target\_col'] = target\_col

model\_dict['MAE'] = MAE

model\_dict['R2'] = R2

model\_dict['fitted\_scaler\_x'] = scale

output\_file = PurePath(path\_to\_save) / ((tag))

#print(model\_dict)

print("\* Trained model saved to file:", str(output\_file))

output = open(output\_file, 'wb')

pickle.dump(model\_dict, output)

def evaluate\_classifier(predictions, actual\_values):

correct = 0

wrong = 0

for i in range(0,len(predictions)):

if predictions[i]==actual\_values[i]:

correct+=1

else:

wrong+=1

accuracy = float(correct)/(float(correct)+float(wrong))

return accuracy

def evaluate(predictions, actual\_values):

try:

MAE = mean\_absolute\_error(predictions,actual\_values)

R2 = r2\_score(actual\_values, predictions, multioutput='variance\_weighted')

except Exception as e:

MAE = -1

R2 = -1

return MAE, R2

def save\_comparison\_chart(predictions, actual\_values, filename):

plt.close()

min\_val = min(predictions)

max\_val = max(actual\_values)

plt.xlabel('Predicted Value')

plt.ylabel('Actual Value')

plt.ylim([min\_val, max\_val])

plt.xlim([min\_val, max\_val])

plt.grid(True)

plt.scatter(predictions,actual\_values)

t = np.arange(min\_val, max\_val, 0.01)

line, = plt.plot(t, t, lw=1)

if type(filename)==str:

filename = PurePath(filename)

if not os.path.exists(filename.parent): os.makedirs(filename.parent)

plt.savefig(filename)

plt.close()

# In[13]:

def add\_key\_to\_params(tag, params):

tag = tag.lower()

parameters = {}

for key in params.keys():

parameters[(tag+'\_'+key).lower()] = params[key]

return parameters

# In[14]:

def hyperparameter\_tuning\_classifier(tag, x\_train, y\_train, num\_of\_folds, scaler\_option, n\_iter=100, random\_state=0, verbose=1):

rf\_n\_estimators = [int(x) for x in np.linspace(start = 10, stop = 1000, num = 20)]

rf\_max\_features = list(range(1,x\_train.shape[1]))

rf\_max\_depth = [int(x) for x in np.linspace(1, 32, 32)]

rf\_max\_depth.append(None)

rf\_min\_samples\_split = [int(x) for x in np.linspace(start = 2, stop = 15, num = 20)]

rf\_min\_samples\_leaf = [int(x) for x in np.linspace(start = 2, stop = 15, num = 20)]

rf\_bootstrap = ['True', 'False']

rf\_criterion = ['gini']

rf\_min\_weight\_fraction\_leaf = [float(x) for x in np.linspace(start = 0, stop = 1.e-5, num = 10)]

rf\_max\_leaf\_nodes = [2, 5, 10, 50, 100]

rf\_min\_impurity\_decrease = [float(x) for x in np.linspace(start = 0, stop = 1.e-5, num = 10)]

rf\_random\_grid = {'n\_estimators': rf\_n\_estimators,

'max\_features': rf\_max\_features,

'max\_depth': rf\_max\_depth,

'min\_samples\_split': rf\_min\_samples\_split,

'min\_samples\_leaf': rf\_min\_samples\_leaf,

'bootstrap': rf\_bootstrap,

'criterion': rf\_criterion,

'min\_weight\_fraction\_leaf': rf\_min\_weight\_fraction\_leaf,

'max\_leaf\_nodes': rf\_max\_leaf\_nodes,

'min\_impurity\_decrease': rf\_min\_impurity\_decrease}

nn\_n\_neighbors = [int(x) for x in np.linspace(start = 2, stop = 15, num = 10)]

nn\_weights = ['uniform', 'distance']

nn\_algorithm = ['auto','ball\_tree','kd\_tree','brute']

nn\_leaf\_size = [1,2,3,4,5]

nn\_p = [int(x) for x in np.linspace(start = 1, stop = 5, num = 5)]

nn\_metric = ['minkowski']

nn\_metric\_params = [None]

nn\_random\_grid = {'n\_neighbors': nn\_n\_neighbors,

'weights': nn\_weights,

'algorithm': nn\_algorithm,

'leaf\_size': nn\_leaf\_size,

'metric': nn\_metric,

'metric\_params': nn\_metric\_params,

'p': nn\_p}

rg\_alpha = [float(x) for x in np.linspace(start = 0, stop = 10, num = 10)]

rg\_fit\_intercept = ['True', 'False']

rg\_max\_iter = [100, 500, 1000, None]

rg\_tol = [float(x) for x in np.linspace(start = 1.e-5, stop = 1.e-2, num = 20)]

rg\_class\_weight = [None,'balanced']

rg\_normalize = ['True', 'False']

rg\_solver = ['auto', 'svd', 'cholesky', 'lsqr', 'sparse\_cg', 'sag', 'saga']

rg\_random\_grid = {'max\_iter': rg\_max\_iter,

'alpha': rg\_alpha,

'fit\_intercept': rg\_fit\_intercept,

'tol': rg\_tol,

'class\_weight': rg\_class\_weight,

'normalize': rg\_normalize,

'solver': rg\_solver

}

svm\_kernel = ['rbf', 'poly','linear','sigmoid']

svm\_gamma = ['auto', 0.001, 0.01, 0.1, 1]

svm\_degree = [1, 2, 3]

svm\_coef0 = [0, 1, 2, 3]

svm\_tol = [float(x) for x in np.linspace(start = 1.e-4, stop = 1.e-2, num = 20)]

#svm\_C = [float(x) for x in np.linspace(start = 0.001, stop = 3000, num = 100)]

svm\_C = [0.001, 0.01, 0.1, 1, 10]

svm\_decision\_function\_shape = ['ovr','ovo']

svm\_random\_grid = {'kernel': svm\_kernel,

'degree': svm\_degree,

'gamma' : svm\_gamma,

'coef0': svm\_coef0,

'tol': svm\_tol,

'C' : svm\_C,

'decision\_function\_shape':svm\_decision\_function\_shape}

if tag=='RF':

estimator = RandomForestClassifier()

random\_grid = rf\_random\_grid

elif tag=='NN':

estimator = KNeighborsClassifier()

random\_grid = nn\_random\_grid

elif tag=='RG':

estimator = RidgeClassifier()

random\_grid = rg\_random\_grid

elif tag=='SVM':

estimator = svm.SVC()

random\_grid = svm\_random\_grid

else:

estimator = None

tuned\_parameters = None

if estimator is not None:

model = RandomizedSearchCV(

estimator = estimator,

param\_distributions = random\_grid,

n\_iter = n\_iter,

cv = num\_of\_folds,

verbose=verbose,

random\_state=random\_state,

n\_jobs = multiprocessing.cpu\_count())

x\_train\_, scale = rescale\_x(scaler\_option, x\_train)

y\_train\_ = y\_train.reshape(y\_train.shape[0],)

model.fit(x\_train\_, y\_train\_)

tuned\_parameters = add\_key\_to\_params(tag, model.best\_params\_)

tuned\_parameters['scaler\_option'] = scaler\_option

return tuned\_parameters

def hyperparameter\_tuning(tag, x\_train, y\_train, num\_of\_folds, scaler\_option, n\_iter=100, random\_state=0, verbose=1):

rf\_n\_estimators = [int(x) for x in np.linspace(start = 10, stop = 1000, num = 20)]

rf\_max\_features = list(range(1,x\_train.shape[1]))

rf\_max\_depth = [int(x) for x in np.linspace(1, 32, 32)]

rf\_max\_depth.append(None)

rf\_min\_samples\_split = [int(x) for x in np.linspace(start = 2, stop = 15, num = 20)]

rf\_min\_samples\_leaf = [int(x) for x in np.linspace(start = 2, stop = 15, num = 20)]

rf\_bootstrap = ['True', 'False']

rf\_criterion = ['mse']

rf\_min\_weight\_fraction\_leaf = [float(x) for x in np.linspace(start = 0, stop = 1.e-5, num = 10)]

rf\_max\_leaf\_nodes = [2, 5, 10, 50, 100]

rf\_min\_impurity\_decrease = [float(x) for x in np.linspace(start = 0, stop = 1.e-5, num = 10)]

rf\_random\_grid = {'n\_estimators': rf\_n\_estimators,

'max\_features': rf\_max\_features,

'max\_depth': rf\_max\_depth,

'min\_samples\_split': rf\_min\_samples\_split,

'min\_samples\_leaf': rf\_min\_samples\_leaf,

'bootstrap': rf\_bootstrap,

'criterion': rf\_criterion,

'min\_weight\_fraction\_leaf': rf\_min\_weight\_fraction\_leaf,

'max\_leaf\_nodes': rf\_max\_leaf\_nodes,

'min\_impurity\_decrease': rf\_min\_impurity\_decrease}

nn\_n\_neighbors = [int(x) for x in np.linspace(start = 2, stop = 15, num = 10)]

nn\_weights = ['uniform', 'distance']

nn\_algorithm = ['auto','ball\_tree','kd\_tree','brute']

nn\_leaf\_size = [1,2,3,4,5]

nn\_p = [int(x) for x in np.linspace(start = 1, stop = 5, num = 5)]

nn\_metric = ['minkowski']

nn\_metric\_params = [None]

nn\_random\_grid = {'n\_neighbors': nn\_n\_neighbors,

'weights': nn\_weights,

'algorithm': nn\_algorithm,

'leaf\_size': nn\_leaf\_size,

'metric': nn\_metric,

'metric\_params': nn\_metric\_params,

'p': nn\_p}

kr\_alpha = [float(x) for x in np.linspace(start = 0, stop = 10, num = 50)]

kr\_gamma = [None, 'RBF', 'laplacian','polynomial','chi2','exponential','sigmoid']

kr\_degree = [1,2,3]

kr\_coef0 = [0,1,2,3,4,5]

kr\_kernel = ['linear']

kr\_random\_grid = {'alpha': kr\_alpha,

'kernel': kr\_kernel,

'gamma': kr\_gamma,

'degree': kr\_degree,

'coef0': kr\_coef0}

br\_n\_iter = [int(x) for x in np.linspace(start = 100, stop = 1000, num = 50)]

br\_alpha\_1 = [float(x) for x in np.linspace(start = 1.e-7, stop = 1.e-4, num = 50)]

br\_alpha\_2 = [float(x) for x in np.linspace(start = 1.e-7, stop = 1.e-4, num = 50)]

br\_tol = [float(x) for x in np.linspace(start = 1.e-2, stop = 1.e-4, num = 50)]

br\_lambda\_1 = [float(x) for x in np.linspace(start = 1.e-7, stop = 1.e-4, num = 50)]

br\_lambda\_2 = [float(x) for x in np.linspace(start = 1.e-7, stop = 1.e-4, num = 50)]

br\_compute\_score = ['True', 'False']

br\_fit\_intercept = ['True', 'False']

br\_normalize = ['True', 'False']

br\_random\_grid = {'n\_iter': br\_n\_iter,

'alpha\_1': br\_alpha\_1,

'alpha\_2': br\_alpha\_2,

'tol': br\_tol,

'lambda\_1': br\_lambda\_1,

'lambda\_2': br\_lambda\_2,

'compute\_score': br\_compute\_score,

'fit\_intercept': br\_fit\_intercept,

'normalize':br\_normalize}

svm\_kernel = ['rbf', 'poly','linear','sigmoid']

svm\_gamma = ['auto', 0.001, 0.01, 0.1, 1]

svm\_degree = [1, 2, 3]

svm\_coef0 = [0, 1, 2, 3]

svm\_tol = [float(x) for x in np.linspace(start = 1.e-4, stop = 1.e-2, num = 50)]

#svm\_C = [float(x) for x in np.linspace(start = 0.001, stop = 3000, num = 100)]

#svm\_C+=[0.001, 0.01, 0.1, 1]

svm\_C = [0.001, 0.01, 0.1, 1, 10]

svm\_epsilon = [0.01, 0.1, 0.2, 0.3]

svm\_random\_grid = {'kernel': svm\_kernel,

'degree': svm\_degree,

'gamma' : svm\_gamma,

'coef0': svm\_coef0,

'tol': svm\_tol,

'C': svm\_C,

'epsilon': svm\_epsilon}

if tag=='RF':

estimator = RandomForestRegressor()

random\_grid = rf\_random\_grid

elif tag=='NN':

estimator = KNeighborsRegressor()

random\_grid = nn\_random\_grid

elif tag=='KR':

estimator = KernelRidge()

random\_grid = kr\_random\_grid

elif tag=='BR':

estimator = linear\_model.BayesianRidge()

random\_grid = br\_random\_grid

elif tag=='SVM':

estimator = svm.SVR()

random\_grid = svm\_random\_grid

else:

estimator = None

tuned\_parameters = None

if estimator is not None:

model = RandomizedSearchCV(

estimator = estimator,

param\_distributions = random\_grid,

n\_iter = n\_iter,

cv = num\_of\_folds,

verbose=verbose,

random\_state=random\_state,

n\_jobs = multiprocessing.cpu\_count())

x\_train\_, scale = rescale\_x(scaler\_option, x\_train)

y\_train\_ = y\_train.reshape(y\_train.shape[0],)

model.fit(x\_train\_, y\_train\_)

tuned\_parameters = add\_key\_to\_params(tag, model.best\_params\_)

tuned\_parameters['scaler\_option'] = scaler\_option

return tuned\_parameters

def split\_data(x\_train, y\_train, num\_of\_folds=5):

num = x\_train.shape[0]/int(num\_of\_folds)

x\_train\_parts = []

y\_train\_parts = []

for i in range(0,int(num\_of\_folds)):

start\_idx = int(i\*num+1) -1

end\_idx = int(num\*(i+1))

if i==int(num\_of\_folds)-1:

x\_train\_parts.append(x\_train[start\_idx:])

y\_train\_parts.append(y\_train[start\_idx:])

else:

x\_train\_parts.append(x\_train[start\_idx:end\_idx])

y\_train\_parts.append(y\_train[start\_idx:end\_idx])

x\_trains = []

y\_trains = []

x\_tests = []

y\_tests = []

for i in range(0, num\_of\_folds):

x\_test = x\_train\_parts[i]

y\_test = y\_train\_parts[i]

x\_train = []

y\_train = []

for j in range(0, num\_of\_folds):

if j!=i:

x\_train+=list(x\_train\_parts[j])

y\_train+=list(y\_train\_parts[j])

x\_trains.append(np.array(x\_train))

y\_trains.append(np.array(y\_train))

x\_tests.append(np.array(x\_test))

y\_tests.append(np.array(y\_test))

return x\_trains, y\_trains, x\_tests, y\_tests

def net\_define(params = [8, 8, 8], layer\_n = 3, input\_size = 16, dropout=0, l\_2=0.01, optimizer='adam', random\_state = 0):

if len(params)!=layer\_n or layer\_n<1:

return None

model = Sequential()

model.add(Dense(params[0], kernel\_initializer='normal', activation='relu', input\_dim=input\_size, kernel\_regularizer=regularizers.l2(l\_2)))

for i in range(1, layer\_n):

if dropout!=0:

model.add(Dropout(dropout))

model.add(Dense(params[i], kernel\_initializer='normal', activation='relu', kernel\_regularizer=regularizers.l2(l\_2)))

model.add(Dense(1, activation = 'linear'))

model.compile(loss='mse',

optimizer=optimizer, metrics=['mape'])

print(params, layer\_n, dropout, l\_2, optimizer)

return model

def net\_define\_classifier(params = [8, 8, 8], layer\_n = 3, num\_of\_class = 2, input\_size = 16, dropout=0, l\_2=0.01, optimizer='adam', random\_state = 0):

if len(params)!=layer\_n or layer\_n<1:

return None

model = Sequential()

model.add(Dense(params[0], kernel\_initializer='normal', activation='relu', input\_dim=input\_size, kernel\_regularizer=regularizers.l2(l\_2)))

for i in range(1, layer\_n):

if dropout!=0:

model.add(Dropout(dropout))

model.add(Dense(params[i], kernel\_initializer='normal', activation='relu', kernel\_regularizer=regularizers.l2(l\_2)))

model.add(Dense(num\_of\_class, activation = 'softmax'))

model.compile(loss='categorical\_crossentropy',

optimizer=optimizer, metrics=['accuracy'])

return model

def evaluate\_net(model, x\_train, y\_train, x\_test, y\_test, epochs=1000, batch\_size=8, verbose = 0):

if optimizer is None:

optimizer = 'adam'

history = model.fit(x\_train, y\_train,

batch\_size=batch\_size,

epochs=epochs,

verbose=verbose,

validation\_data=(x\_test, y\_test))

score = model.evaluate(x\_test, y\_test, verbose=0)

return score, history

def net\_tuning\_classifier(x\_train, y\_train, num\_of\_class = 2, tries = 10, lr = None, layer = None, params=None, epochs=None, batch\_size=None, dropout=None, l\_2 = None, neuron\_max=[64, 64, 64], batch\_size\_max=32, layer\_min=1, layer\_max=3, dropout\_max=0.2, scaler\_option='StandardScaler', default\_neuron\_max=32, checkpoint = None, num\_of\_folds=5, fast\_tune = True, random\_state = 0):

tuned\_parameters = {}

if layer is not None:

if layer<1:

print("Error: layer must be >=1")

sys.exit()

# Trying to tune hyperparameters

best\_score = 0

best\_params = None

\_layer = layer

\_params = params

\_epochs = epochs

\_batch\_size = batch\_size

\_dropout = dropout

\_l\_2 = l\_2

\_neuron\_max = neuron\_max

\_batch\_size\_max = batch\_size\_max

\_dropout\_max = dropout\_max

\_lr = lr

for i in range(0, tries):

optimizer = 'adam'

if lr is None:

lr = 10\*\*np.random.uniform(-4,-3)

optimizer = keras.optimizers.Adam(lr=lr)

else:

optimizer = keras.optimizers.Adam(lr=lr)

if layer is None:

layer = random.sample(range(layer\_min, layer\_max+1), 1)[0]

if params is None:

params = []

for j in range(0, layer):

try:

params.append(random.sample(range(1, neuron\_max[j]), 1)[0])

except:

params.append(random.sample(range(1, default\_neuron\_max), 1)[0])

if epochs is None:

epochs = int(10\*\*np.random.uniform(2,3)) # 100 - 1000

if batch\_size is None:

batch\_size = random.sample(range(1, batch\_size\_max), 1)[0]

if dropout is None:

dropout = 0

elif dropout is True:

dropout = random.uniform(0,dropout\_max)

else:

dropout = dropout

if l\_2 is None:

l\_2 = 10\*\*np.random.uniform(-3,-1)

model = net\_define\_classifier(params=params, layer\_n = layer, input\_size = x\_train.shape[1], dropout=dropout, l\_2=l\_2, optimizer=optimizer, num\_of\_class = num\_of\_class, random\_state = random\_state)

print("\n Cross-validation (iteration=%d): [layer=%d, structure=[%s], epochs=%d, dropout=%8.4f, l\_2=%8.7f, batch\_size=%d, lr=%8.7f]"%(i, layer, params, epochs, dropout, l\_2, batch\_size, lr))

start\_time = time.time()

predictions, actual\_values = cross\_val\_predict\_net\_classifier(model, epochs=epochs, batch\_size=batch\_size, x\_train = x\_train, y\_train = y\_train, verbose = 0, scaler\_option = scaler\_option, num\_of\_folds = num\_of\_folds, num\_of\_class = num\_of\_class, fast\_tune = fast\_tune)

if predictions == []:

print(" Validation stopped early with the setting:","[layer=%d, structure=[%s], epochs=%d, dropout=%8.4f, l\_2=%8.7f, batch\_size=%d, lr=%8.7f]"%(layer, params, epochs, dropout, l\_2, batch\_size, lr))

print(' Keep trying to find best settings .., took %ss'%(time.time()-start\_time))

accuracy= -1

else:

accuracy = evaluate\_classifier(predictions, actual\_values)

print("Cross validation result - accuracy = %8.3f, took %ss "%(accuracy, time.time()-start\_time))

if(accuracy>best\_score):

best\_score = accuracy

best\_params = (layer, params, epochs, dropout, l\_2, batch\_size, lr)

tuned\_parameters = {"net\_layer\_n":best\_params[0], \

"net\_structure":str(best\_params[1])[1:-1].replace(",",""), \

"net\_epochs":best\_params[2], \

"net\_dropout":best\_params[3], \

"net\_l\_2":best\_params[4], \

"net\_batch\_size":best\_params[5], \

"net\_learning\_rate": best\_params[6]}

if checkpoint is not None:

print("Best so far parameters stored :", str(checkpoint)+",Model=NET,Scaler="+str(scaler\_option)+",accuracy="+str(accuracy)+".tuned.checkpoint.prop")

save\_parameters(tuned\_parameters, str(checkpoint)+",Model=NET,Scaler="+str(scaler\_option)+",accuracy="+str(accuracy)+".tuned.checkpoint.prop")

if(best\_score!=0):

print("Best accuracy = %8.3f"%(best\_score),"[layer=%d, structure=[%s], epochs=%d, dropout=%8.4f, l\_2=%8.7f, batch\_size=%d, lr=%8.7f]"%best\_params)

# set to original values

layer = \_layer

params = \_params

epochs = \_epochs

batch\_size = \_batch\_size

dropout = \_dropout

l\_2 = \_l\_2

neuron\_max = \_neuron\_max

batch\_size\_max = \_batch\_size\_max

dropout\_max = \_dropout\_max

lr = \_lr

tuned\_parameters['scaler\_option'] = scaler\_option

return tuned\_parameters

print(" -- DONE --")

def net\_tuning(x\_train, y\_train, tries = 10, lr = None, layer = None, params=None, epochs=None, batch\_size=None, dropout=None, l\_2 = None, neuron\_max=[64, 64, 64], batch\_size\_max=32, layer\_min=1, layer\_max=3, dropout\_max=0.2, scaler\_option='StandardScaler', default\_neuron\_max=32, checkpoint = None, num\_of\_folds=5, fast\_tune = True, random\_state = 0):

tuned\_parameters = {}

if layer is not None:

if layer<1:

print("Error: layer must be >=1")

sys.exit()

# Trying to tune hyperparameters

best\_score = 0

best\_params = None

best\_mae = 0

\_layer = layer

\_params = params

\_epochs = epochs

\_batch\_size = batch\_size

\_dropout = dropout

\_l\_2 = l\_2

\_neuron\_max = neuron\_max

\_batch\_size\_max = batch\_size\_max

\_dropout\_max = dropout\_max

\_lr = lr

for i in range(0, tries):

optimizer = 'adam'

if lr is None:

lr = 10\*\*np.random.uniform(-4,-3)

optimizer = keras.optimizers.Adam(lr=lr)

else:

optimizer = keras.optimizers.Adam(lr=lr)

if layer is None:

layer = random.sample(range(layer\_min, layer\_max+1), 1)[0]

params = \_params

if params is None:

params = []

for j in range(0, layer):

try:

params.append(random.sample(range(1, neuron\_max[j]), 1)[0])

except:

params.append(random.sample(range(1, default\_neuron\_max), 1)[0])

if epochs is None:

epochs = int(10\*\*np.random.uniform(2,3)) # 100 - 1000

if batch\_size is None:

batch\_size = random.sample(range(1, batch\_size\_max), 1)[0]

if dropout is None:

dropout = 0

elif dropout is True:

dropout = random.uniform(0,dropout\_max)

else:

dropout = dropout

if l\_2 is None:

l\_2 = 10\*\*np.random.uniform(-3,-1)

model = net\_define(params=params, layer\_n = layer, input\_size = x\_train.shape[1], dropout=dropout, l\_2=l\_2, optimizer=optimizer, random\_state = random\_state)

print("\n Cross-validation (iteration=%d): [layer=%d, structure=[%s], epochs=%d, dropout=%8.4f, l\_2=%8.7f, batch\_size=%d, lr=%8.7f]"%(i, layer, params, epochs, dropout, l\_2, batch\_size, lr))

start\_time = time.time()

predictions, actual\_values = cross\_val\_predict\_net(model, epochs=epochs, batch\_size=batch\_size, x\_train = x\_train, y\_train = y\_train, verbose = 0, scaler\_option = scaler\_option, num\_of\_folds = num\_of\_folds, fast\_tune = fast\_tune)

if predictions == []:

print(" Validation stopped early with the setting:","[layer=%d, structure=[%s], epochs=%d, dropout=%8.4f, l\_2=%8.7f, batch\_size=%d, lr=%8.7f]"%(layer, params, epochs, dropout, l\_2, batch\_size, lr))

print(' Keep trying to find best settings .., took %ss'%(time.time()-start\_time))

MAE= -1

R2 = -1

else:

MAE, R2 = evaluate(predictions, actual\_values)

print("Cross validation result - MAE = %8.3f R2 = %8.3f, took %ss "%(MAE, R2, time.time()-start\_time))

if(R2>best\_score and R2!=-1) or (best\_score==0 and R2!=-1):

best\_score = R2

best\_mae = MAE

best\_params = (layer, params, epochs, dropout, l\_2, batch\_size, lr)

tuned\_parameters = {"net\_layer\_n":best\_params[0], \

"net\_structure":str(best\_params[1])[1:-1].replace(",",""), \

"net\_epochs":best\_params[2], \

"net\_dropout":best\_params[3], \

"net\_l\_2":best\_params[4], \

"net\_batch\_size":best\_params[5], \

"net\_learning\_rate": best\_params[6]}

if checkpoint is not None:

print("Best so far parameters stored :", str(checkpoint)+",Model=NET,Scaler="+str(scaler\_option)+",MAE="+str(MAE)+",R2="+str(R2)+".tuned.checkpoint.prop")

save\_parameters(tuned\_parameters, str(checkpoint)+",Model=NET,Scaler="+str(scaler\_option)+",MAE="+str(MAE)+",R2="+str(R2)+".tuned.checkpoint.prop")

print(" Saving test charts to : ", str(checkpoint)+",Model=NET,Scaler="+str(scaler\_option)+",MAE="+str(MAE)+",R2="+str(R2)+".tuned.checkpoint.png")

#try:

save\_comparison\_chart(predictions, actual\_values, str(checkpoint)+",Model=NET,Scaler="+str(scaler\_option)+",MAE="+str(MAE)+",R2="+str(R2)+".tuned.checkpoint.png")

#except:

# print(" \* Warning: couldn't generate a chart - please make sure the model is properly trained .. ")

if(best\_score!=0):

print("Best R2 = %8.3f"%(best\_score),"MAE=",best\_mae, "[layer=%d, structure=[%s], epochs=%d, dropout=%8.4f, l\_2=%8.7f, batch\_size=%d, lr=%8.7f]"%best\_params)

# set to original values

layer = \_layer

params = \_params

epochs = \_epochs

batch\_size = \_batch\_size

dropout = \_dropout

l\_2 = \_l\_2

neuron\_max = \_neuron\_max

batch\_size\_max = \_batch\_size\_max

dropout\_max = \_dropout\_max

lr = \_lr

tuned\_parameters['scaler\_option'] = scaler\_option

return tuned\_parameters

print(" -- DONE --")