Supplementary Information (SI) for ChemComm.
This journal is © The Royal Society of Chemistry 2024

Supporting information

Predict the mechanical and flame-retardant properties of MOF-loaded polymer composites

Junchen Xiao[a,b], Maciej Haranczyk[a*], De-Yi Wang[a*]

[a] *IMDEA Materials Institute, C/Eric Kandel, 2, 28906 Getafe, Madrid, Spain.*

b Universidad Politécnica de Madrid, E.T.S. de Ingenieros de Caminos, 28040 Madrid, Spain.

c Advanced Chemical Energy Research Center, Institute of Advanced Sciences, Yokohama National University, Yokohama 240-8501, Japan.

**Email:* deyi.wang@imdea.org, Maciej.haranczyk@imdea.org

S1 Data collection

[1–59]The data used to build machine learning models are mainly collected from 59 scientific publications, and some experimental data from our group. All the research works are related to the usage of MOF in polymer composites, individually or together with other flame retardants. In the collection of data, values are obtained from the tables or diagrams. If the values are not shown in numbers directly in the articles, two strategies would be applied: using the Digitizer in Origin to read the values from the graphs or deduced from the results part indirectly. All the literature used is listed at the end.

S2 Dataset pre-processing and feature selection

Each record in the dataset, much like the formulation of the polymer composite, is marked as unique through the combination of features containing the information about the type of polymer matrix, loading of the additives (referring to MOFs and other FR chemicals), and operation parameters in sample preparation and characterization. In total, 268 different sample points were collected to build the machine learning models separately.

The input features describe the processes and materials used to produce the corresponding polymer composites and are listed below in detail:

1) Properties of polymer matrix ("Polymer_Matrix"): basic physical properties of neat polymer such as density, thermal conductivity, and decomposition temperatures; storage modulus and CCT characteristics are also included.
2) Metal-Organic Framework ("MOF_Materials"): a combination of metal ions and organic ligands, represented mainly by their atomic numbers and

molecular weights; its thermal decomposition properties, micro-structures and type of the building units are collected.

3) FR additives (“Main_FR”): other FR additives added into polymer composites besides MOF; typically, the chemical compositions in element contents and loading amount by weight percent.
4) Other parameters (“Other_Parameters”): the parameters concerning the sample preparation and characterization methods; for example, processing temperatures of materials, heat flux of CCT, sample sizes and testing modes are significant.

All the related values were collected to make sure we didn’t lose any significant information about the polymer composites. In general, for those features containing vacancies below 20% in the dataset, we have searched the samples with similar formulations and inserted the mathematical mean values into the missing blanks, such the missing values of polymer matrix. If the features have too many vacancies, we must drop them to keep the dataset clean and trustable. However, we will add some indirect descriptors to compensate for the information loss. Target features are fire and mechanical properties, more specifically, the storage modulus at room temperature (“*Modulus*”) from DMA, time to ignition (“*TTI*”), peak heat release rate (“*pHRR*”) and total heat release (“*THR*”) from CCT.

Instead of numeric values, these four target properties have been converted to categorical features. But firstly, the properties of polymer composites were divided by the corresponding values of neat polymer (e.g. from “*TTI*” to “*TTI_d*”). then these ratios were classified to different categories (e.g. from “*TTI_d*” to “*TTI_dc*”) as shown in Table S2.

$$TTI_d = \frac{TTI_{polymer\ composite}}{TTI_{neat\ polymer}} \quad (1)$$

S3 Model selection and assessment

Pioneer researchers have demonstrated abundant machine learning algorithms that establish high-performance models oriented toward different situations and requirements in material science. Supervised learning, in which a model is trained with a large amount of data acting much like a researcher, has been adopted in most investigations. We have constructed 3 machine learning models for all target features in our work separately. The first supervised learning technique is the widely used Random Forest (RF) developed by Leo Breiman and Adele Cutler, and then a Support Vector Machine (SVM) with totally different mathematical fundamentals. The former is a tree-based ensemble method combining intended number of simple estimators as shown in Figure S1. This basic unit is a decision tree, in which the prediction is made by walking through all necessary conditional control statements. RF collects the results of such sub-classifiers and averages the data to obtain the final prediction with high accuracy and low overfitting. Besides, this algorithm benefits greatly from an intrinsic function of ranking the input features by their influence on the prediction of RF model over the dataset, which is called the feature importance. The interpretability allows researchers to gain insight into the relevance of a specific feature and adjust the process of feature engineering timely. This is the reason for choosing the popular RF as main algorithm besides their universality and reliability.

The second algorithm SVM is a powerful tool for classification with complex dataset and has sometimes better performance than other techniques due to many features like various kernel functions and capacity control brought by margin optimization etc. It constructs hyperplane with certain margin as boundary between different classes. This algorithm and the variants have been widely applied in image classification like biological and medical studies, military application, hydrological science, agriculture sector, remote sensing etc. The third model, which is actually built upon the former two models, is the linear combination of classifiers. In general, the performance of prediction is strongly related to the characteristics of dataset, complexity of research purpose and iterative process of machine learning model. In

some cases, a single model fails to live up to expectations due to certain limitations. In view of such facts, combining machine learning models is an effective strategy to improve predictive accuracy.

The whole dataset was split into training and test sets in a ratio of 80 : 20. The training set, 80% of the collected data, was used to confirm the models' structure and train the (hyper-)parameters with k-fold cross validation, followed by the immediate evaluation of predictive performance with the test set. The train-test split was repeated 100 times randomly to avoid the occasionality, which is usually reflected by the strong up-and-down fluctuations in the determination coefficients. Meanwhile, mean errors of the predictions were calculated at the same time. In addition, we plotted the precision-recall curves (receiver operating characteristic curve, short as ROC) for the classifiers and calculated the area under curve (AUC) to evaluate the predictive accuracy of classification.

As mentioned above, analyzing the relevance of input features was performed on the basis of the dataset and machine learning models. Moldes of tree-based Random Forest Classifier (RFC) provided integrated feature importance while training the models. For kernel models based on SVM, we used SHapley Additive exPlanations (SHAP) to interpret the outputs. This tool from cooperative game theory attributes the output to the contribution (i.e. the shapely value) of each feature, representing its influence on prediction of target classes.

S4 Feature "MOF_Class"

Due to the lack of information about MOF's structure and morphology, the feature "MOF_Class" was intended to contain such information for modelling. In this work, MOF consisting of simple metal cation and imidazole was considered as ZEO-type. If the organic linker contains -COOH groups to form the MOF structure, it was MIL-type. The UIO-type comes from UIO-66, which is made up of a cluster with acid. Other MOFs that cannot be distinguished into the above-mentioned types were

divided into IRM-type. These types are merely the labels to distinguish different MOFS.

S5 Validation experiments

For the validation part, we prepared 4 different EP-based polymer composites. EP resin (Epoxydhedraz C) was provided by R&G Faserverbundwerkstoffe GmbH, Germany. Aluminum tri-hydroxide (ATH), diamino diphenyl methane (DDM, ≥97.0%) and pentaerythritol (PER) were purchased from Sigma-Aldrich Química SL. Ammonium polyphosphate (APP) was provided by Clariant AG, marked with Exolit AP 750. The intumescent flame retardant (IFR) was obtained by mixing APP with PER in a ratio of 3:1. The MOF was the typical Fe-BTC (CAS number 1195763-37-1) purchased from MOF Technologies Ltd, United Kingdom.

All EP samples were prepared by triple roll milling at 40 °C and stepwise curing. After 30 min milling with the miller (EXAKT 80E), the curing agent DDM was added, and the mixer was degassed at 90 °C for 10 min. The curing was done in a thermostatic oven at 110 °C and 150 °C for 2 h respectively. The cured samples were molded into squares (100 × 100 mm^2) with a thickness varied from 3 to 5 mm for fire testing, and bars (size of DMA test) for mechanical test. According to ISO5660, we used the mass loss cone calorimeter (short as CCT) from Fire Testing technology to characterize the fire performance of EP samples. Heat flux was fixed with 50 kW/m^2 for all measurements. The dynamic mechanical analysis (DMA) was done with DMA Q800 in three-point bend mode.

Figures

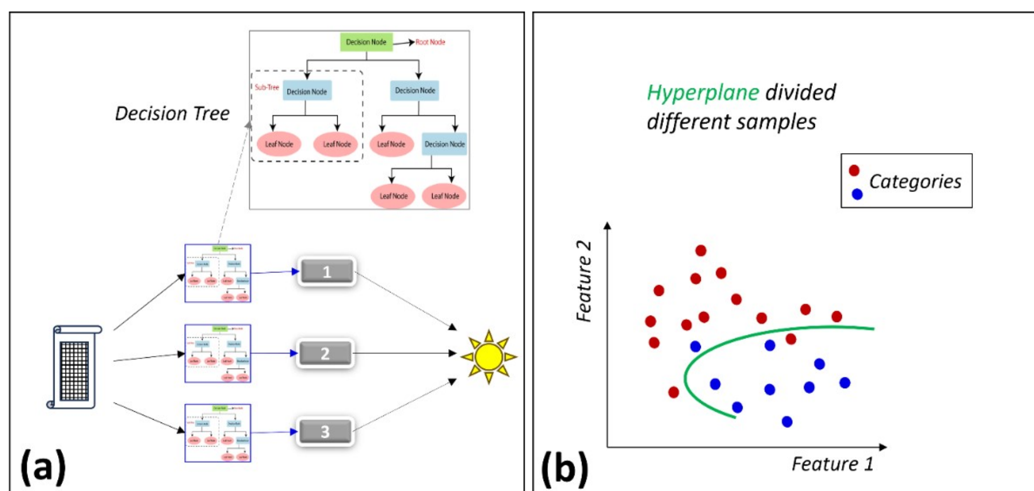

Figure S1 *supervised learning algorithms: left - Random Forest (RF) and right - Support Vector Machine (SVM)*

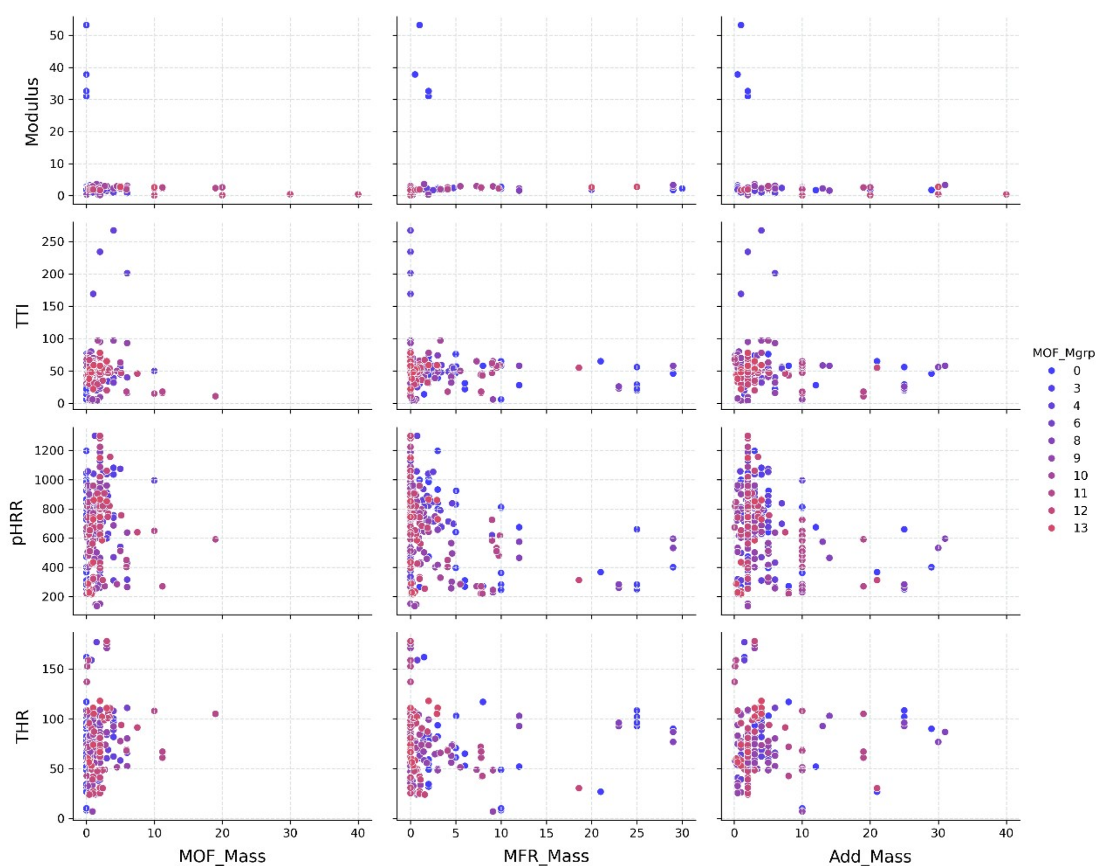

Figure S2 *Pairwise plotting of significant features against the target properties of polymer composite: Mass fraction of MOFs or/and other flame-retardant additives.*

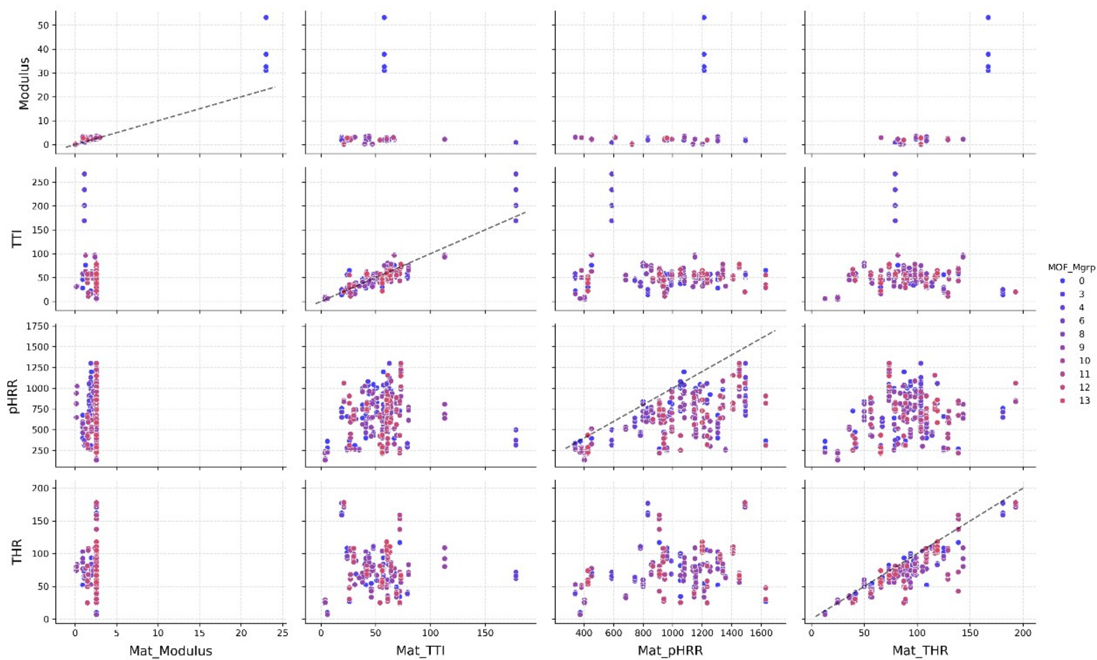

Figure S3 *Pairwise plotting of significant features against the target properties of polymer composite: relationships between the properties of polymer composites and neat polymers; with colored dots representing the group of metal in MOF*

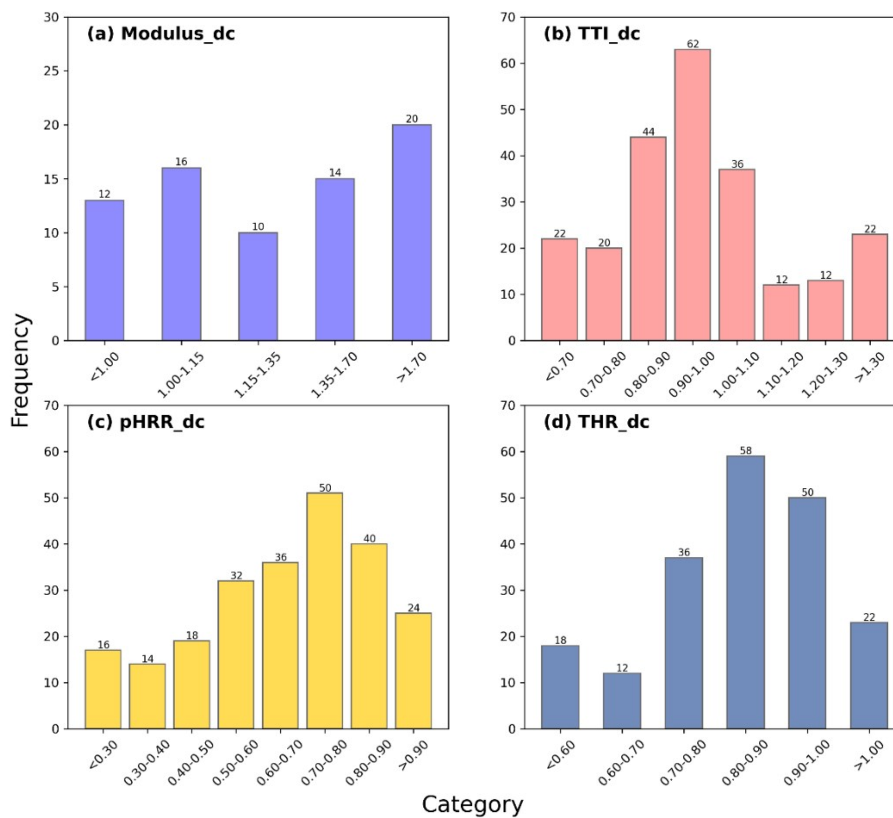

Figure S4 *Distribution of all categories of four target properties in our dataset*

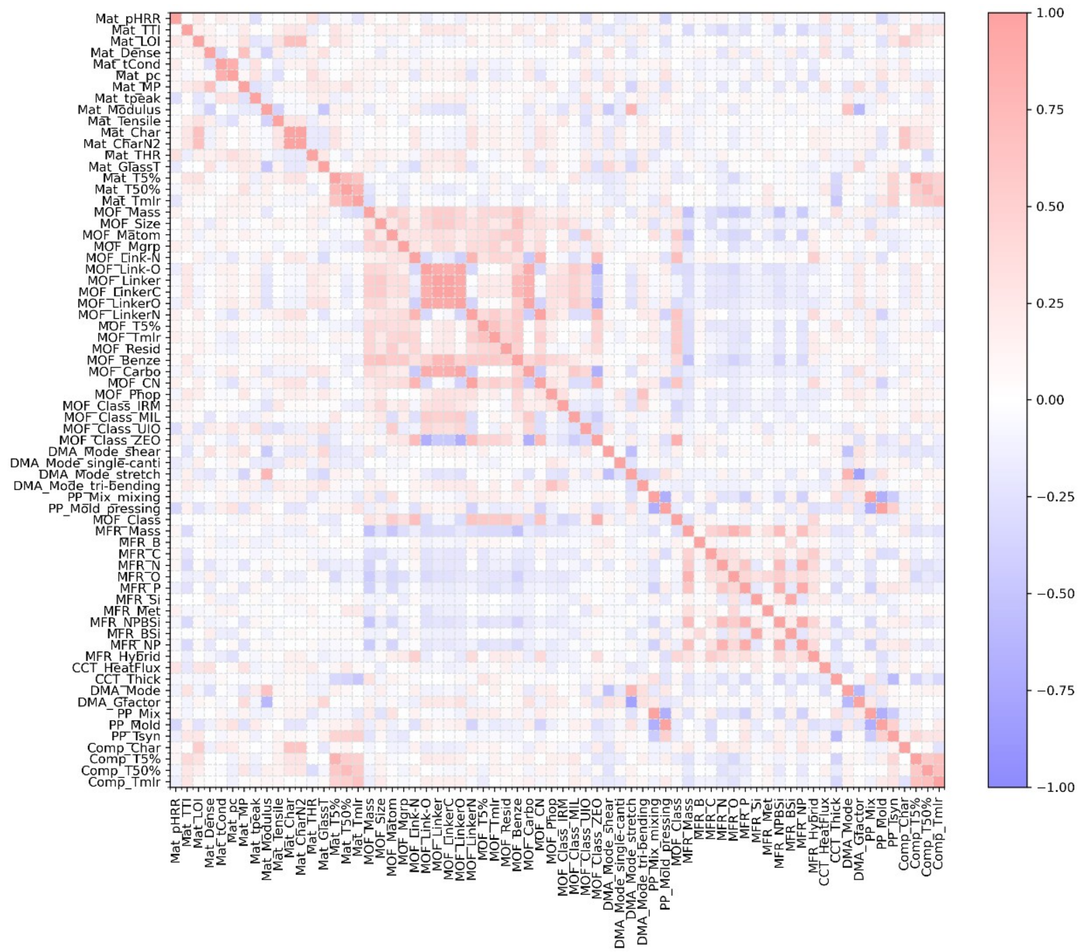

Figure S5 *Spearman's correlation coefficients between input features, the lighter the color is, the weaker the correlation is*

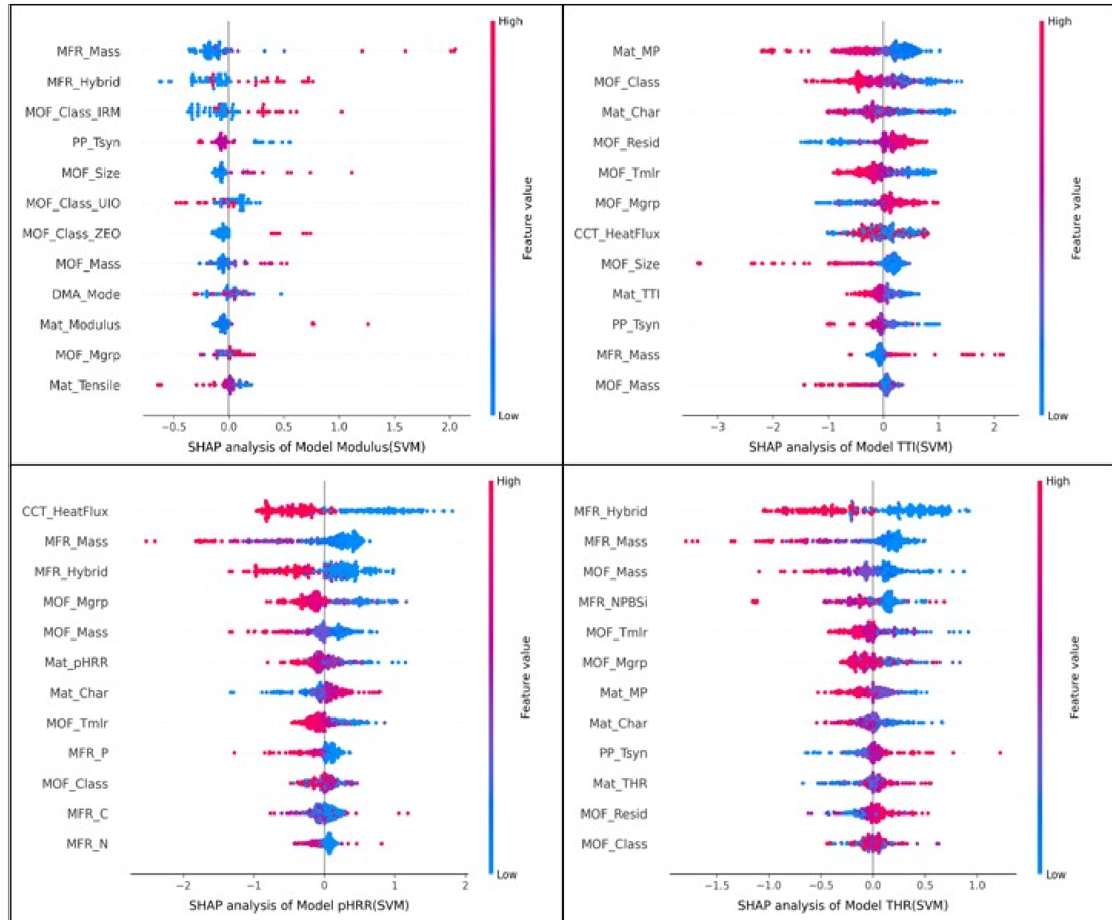

Figure S6 *SHAP interpretation of SVM models predicting "Modulus", "TTI", "pHRR" and "THR". Color stands for the values of features in each row*

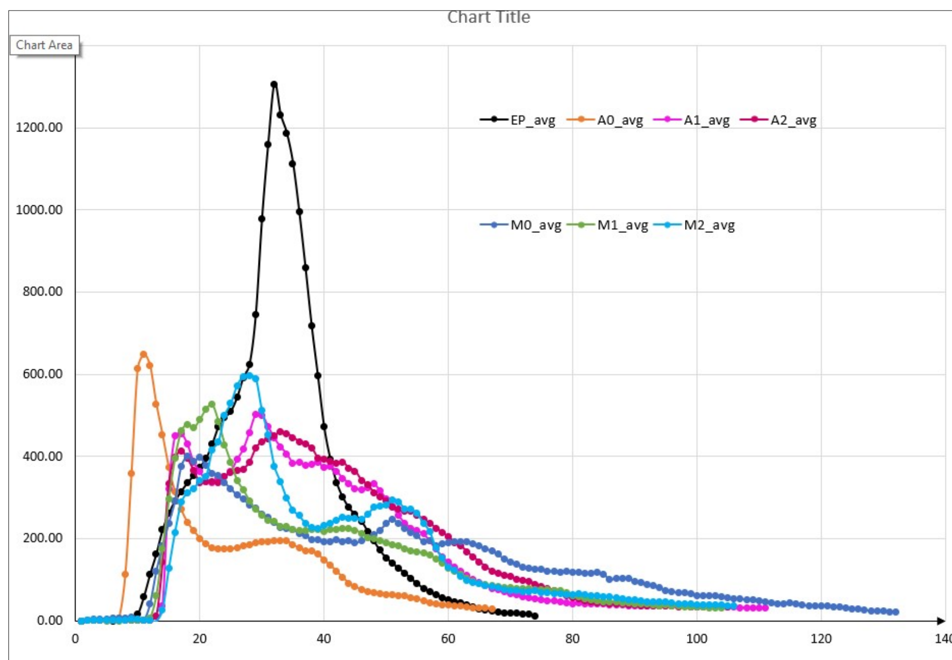

Figure S7 *Heat release rate of EP samples measured with heat flux=35kW/m²*

Tables

Table S1: Some research works about MOF-loading in polymer composites

Polymer matrix	MOF type	Synergist	year	Mechanical performance	Cone results
TPU	Co-DdmSa	APP	2019	Tensile strength decreased from 24.3 MPa to 17.3 MPa	TTI slightly lower, pHRR reduction of 80% (257 kW/m2), THR decreased by 20%
UP	HKUST-1	DMMP	2019	Storage modulus @25C raised to 2.3 GPa	TTI lowered by 40%, pHRR reduction of 70%, THR halved compared to neat polymer
PC	UIO-66		2019	Both tensile strength and storage modulus increased by 10%	TTI was prolong, pHRR halved to 316 kW/m2 and THR decreased
EP	UIO-66	SiO_2	2019	Storage modulus slightly increased to 2.1 GPa	pHRR and THR decreased by 30%, TTI stayed the same
PLA	Ni-SaTr	APP	2020	Storage modulus significantly increased to 2 GPa	TTI was prolonged to 97 s, pHRR reduction of 28%, and THR decreased by 22%
PS	Co-BDC	PCT	2020	Storage modulus increased from 2.6 to 3.6 GPa	TTI stayed the same, but pHRR reduction of 40%
EP	UIO-66	PA + β-CD	2021	Tensile strength slightly decreased, but storage modulus was improved to 1.9 GPa	TTI was the same to neat EP of 60ss, pHRR reduction of 50% (675 kW/m2)
EP	ZIF-67	PA	2022	storage modulus increased by 50% to 0.33 GPa	TTI was shorter but pHRR reduced to 645 kW/m2

Table S2: Categories of target properties, suffix "_dc" means this target feature was divided and categorized

Target	Categories							
Modulus_dc	< 1.00	1.00-1.15	1.15-1.35	1.35-1.70	>1.70			
TTI_dc	< 0.70	0.70-0.80	0.80-0.90	0.90-1.00	1.00-1.10	1..10-1.20	1.20-1.30	>1.30
pHRR_dc	<0.30	0.30-0.40	0.40-0.50	0.50-0.60	0.60-0.70	0.70-0.80	0.80-0.90	>0.90
THR_dc	<0.60	0.60-0.70	0.70-0.80	0.80-0.90	0.90-1.00	>1.00		

Table S3: Epoxy composites with mass fraction of MOF, ATH and IFR in wt.-%

Label	EP	MOF	ATH	IFR
M0	71	0	29	-
M1	70	1	29	-
M2	69	2	29	-
A0	88	0	-	12
A1	87	1	-	12
A2	86	2	-	12

Table S4: Model indices of all 12 models

Indices of machine learning models	R2		MAE		RMSE	
	train	test	train	test	train	test
Modulus(RF)	0.87	0.75	0.18	0.42	0.54	0.96
Modulus(SVM)	0.99	0.75	0.01	0.33	0.01	0.5
Modulus(SVG)	0.96	0.77	0.09	0.33	0.3	0.71
TTI(RF)	1.0	0.84	0	0.2	0	0.53
TTI(SVM)	0.96	0.78	0.07	0.32	0.15	0.62
TTI(SVG)	1.0	0.78	0.02	0.51	0.14	1.04
pHRR(RF)	1.0	0.67	0	0.4	0	0.73
pHRR (SVM)	0.97	0.5	0.04	0.93	0.05	2.07
pHRR (SVG)	1.0	0.77	0.01	0.62	0.11	1.07
THR(RF)	1.0	0.76	0	0.37	0	0.85
THR (SVM)	0.99	0.76	0.01	0.33	0.01	0.57
THR (SVG)	1.0	0.84	0	0.37	0	0.7

References

1 X. G. Wang, P. Qi, S. J. Zhang, S. L. Jiang, Y. C. Li, J. Sun, B. Fei, X. Y. Gu and S. Zhang, A novel flame-retardant modification strategy for UiO66-NH2 by encapsulating triethyl phosphate: preparation, characterization, and multifunctional application in poly (lactic acid), *Materials Today Chemistry*, 2023, **30**, 101550.

2 R. Wang, Y. Chen, Y. Liu, M. Ma, Z. Tong, X. Chen, Y. Bi, W. Huang, Z. Liao, S. Chen, X. Zhang and Q. Li, Metal-organic frameworks derived ZnO @ MOF @ PZS flame retardant for reducing fire hazards of polyurea nanocomposites, *Polymers for Advanced Techs*, 2021, **32**, 4700–4709.

3 J. Wang, Y. Liu, X. Guo, H. Qu, R. Chang and J. Ma, Efficient Adsorption of Dyes Using Polyethyleneimine-Modified NH 2 -MIL-101(Al) and its Sustainable Application as a Flame Retardant for an Epoxy Resin, *ACS Omega*, 2020, **5**, 32286–32294.

4 X. Wang, S. Wang, W. Wang, H. Li, X. Liu, X. Gu, S. Bourbigot, Z. Wang, J. Sun and S. Zhang, The flammability and mechanical properties of poly (lactic acid) composites containing Ni-MOF nanosheets with polyhydroxy groups, *Composites Part B: Engineering*, 2020, **183**, 107568.

5 W. Xu, L. Fan, Z. Qin, Y. Liu and M. Li, Silica-coated metal-organic framework-β-FeOOH hybrid for improving the flame retardant and smoke suppressive properties of epoxy resin, *Plastics, Rubber and Composites*, 2021, **50**, 396–405.

6 W. Xu, Z. Cheng, Di Zhong, Z. Qin, N. Zhou and W. Li, Effect of two-dimensional zeolitic imidazolate frameworks-L on flame retardant property of thermoplastic polyurethane elastomers, *Polymers for Advanced Techs*, 2021, **32**, 2072–2081.

7 B. Xu, W. Xu, G. Wang, L. Liu and J. Xu, Zeolitic imidazolate frameworks-8 modified graphene as a green flame retardant for reducing the fire risk of epoxy resin, *Polymers for Advanced Techs*, 2018, **29**, 1733–1743.

8 K. Song, X. Li, Y.-T. Pan, B. Hou, Z. U. Rehman, J. He and R. Yang, The

influence on flame retardant epoxy composites by a bird's nest-like structure of Co-based isomers evolved from zeolitic imidazolate framework-67, *Polymer Degradation and Stability*, 2023, **211**, 110318.

9 K. Song, B. Hou, Z. Ur Rehman, Y.-T. Pan, J. He, D.-Y. Wang and R. Yang, “Sloughing” of metal-organic framework retaining nanodots via step-by-step carving and its flame-retardant effect in epoxy resin, *Chemical Engineering Journal*, 2022, **448**, 137666.

10 R. Shen, T.-H. Yan, R. Ma, E. Joseph, Y. Quan, H.-C. Zhou and Q. Wang, Flammability and Thermal Kinetic Analysis of UiO-66-Based PMMA Polymer Composites, *Polymers*, 2021, **13**, 4113.

11 K. Song, Y. Wang, F. Ruan, W. Yang, Z. Fang, D. Zheng, X. Li, N. Li, M. Qiao and J. Liu, Synthesis of a Reactive Template-Induced Core–Shell PZS@ZIF-67 Composite Microspheres and Its Application in Epoxy Composites, *Polymers*, 2021, **13**, 2646.

12 H. Wang, H. Qiao, J. Guo, J. Sun, H. Li, S. Zhang and X. Gu, Preparation of cobalt-based metal organic framework and its application as synergistic flame retardant in thermoplastic polyurethane (TPU), *Composites Part B: Engineering*, 2020, **182**, 107498.

13 M. Wan, C. Shi, X. Qian, Y. Qin, J. Jing and H. Che, Metal-organic Framework ZIF-67 Functionalized MXene for Enhancing the Fire Safety of Thermoplastic Polyurethanes, *Nanomaterials (Basel, Switzerland)*, 2022, **12**. DOI: 10.3390/nano12071142.

14 V. Unnikrishnan, O. Zabihi, Q. Li, M. Ahmadi, M. R. G. Ferdowsi, T. Kannangara, P. Blanchard, A. Kiziltas, P. Joseph and M. Naebe, Multifunctional PA6 composites using waste glass fiber and green metal organic framework/graphene hybrids, *Polymer Composites*, 2022, **43**, 5877–5893.

15 W. Xu, G. Wang, Y. Liu, R. Chen and W. Li, Zeolitic imidazolate framework-8 was coated with silica and investigated as a flame retardant to improve the flame retardancy and smoke suppression of epoxy resin, *RSC Adv.*, 2018, **8**, 2575–2585.

16 J. Zhang, Z. Li, G.-Z. Yin and D.-Y. Wang, Construction of a novel three-in-one

biomass based intumescent fire retardant through phosphorus functionalized metal-organic framework and β-cyclodextrin hybrids in achieving fire safe epoxy, *Composites Communications*, 2021, **23**, 100594.

17 J. Zhang, Z. Li, Z.-B. Shao, L. Zhang and D.-Y. Wang, Hierarchically tailored hybrids via interfacial-engineering of self-assembled UiO-66 and prussian blue analogue: Novel strategy to impart epoxy high-efficient fire retardancy and smoke suppression, *Chemical Engineering Journal*, 2020, **400**, 125942.

18 G. Zhang, W. Wu, M. Yao, Z. Wu, Y. Jiao and H. Qu, Novel triazine-based metal-organic frameworks: Synthesis and mulifunctional application of flame retardant, smoke suppression and toxic attenuation on EP, *Materials & Design*, 2023, **226**, 111664.

19 J. Zhang, Z. Li, X. Qi, W. Zhang and D.-Y. Wang, Size tailored bimetallic metal-organic framework (MOF) on graphene oxide with sandwich-like structure as functional nano-hybrids for improving fire safety of epoxy, *Composites Part B: Engineering*, 2020, **188**, 107881.

20 Y. Zheng, Y. Lu and K. Zhou, A novel exploration of metal–organic frameworks in flame-retardant epoxy composites, *J Therm Anal Calorim*, 2019, **138**, 905–914.

21 J. Zhang, Z. Li, L. Zhang, Y. Yang and D.-Y. Wang, Green Synthesis of Biomass Phytic Acid-Functionalized UiO-66-NH 2 Hierarchical Hybrids toward Fire Safety of Epoxy Resin, *ACS Sustainable Chem. Eng.*, 2020, **8**, 994–1003.

22 J. Zhang, Z. Li, L. Zhang, J. García Molleja and D.-Y. Wang, Bimetallic metal-organic frameworks and graphene oxide nano-hybrids for enhanced fire retardant epoxy composites: A novel carbonization mechanism, *Carbon*, 2019, **153**, 407–416.

23 J. Yang, A. Zhang, Y. Chen, L. Wang, M. Li, H. Yang and Y. Hou, Surface modification of core–shell structured ZIF-67@Cobalt coordination compound to improve the fire safety of biomass aerogel insulation materials, *Chemical Engineering Journal*, 2022, **430**, 132809.

24 Z. Xu, W. Xing, Y. Hou, B. Zou, L. Han, W. Hu and Y. Hu, The combustion and pyrolysis process of flame-retardant polystyrene/cobalt-based metal organic

frameworks (MOF) nanocomposite, *Combustion and Flame*, 2021, **226**, 108–116.

25 W. Xu, X. Wang, Y. Wu, W. Li and C. Chen, Functionalized graphene with Co-ZIF adsorbed borate ions as an effective flame retardant and smoke suppression agent for epoxy resin, *Journal of Hazardous Materials*, 2019, **363**, 138–151.

26 X. Yang, B. L. Bonnett, G. A. Spiering, H. D. Cornell, B. J. Gibbons, R. B. Moore, E. J. Foster and A. J. Morris, Understanding the Mechanical Reinforcement of Metal-Organic Framework-Polymer Composites: The Effect of Aspect Ratio, *ACS applied materials & interfaces*, 2021, **13**, 51894–51905.

27 F. Zhang, X. Li, L. Yang, Y. Zhang and M. Zhang, A Mo-based metal-organic framework toward improving flame retardancy and smoke suppression of epoxy resin, *Polymers for Advanced Techs*, 2021, **32**, 3266–3277.

28 S. Yu, C. Cheng, K. Li, J. Wang, Z. Wang, H. Zhou, W. Wang, Y. Zhang and Y. Quan, Fire-safe epoxy composite realized by MXenes based nanostructure with vertically arrayed MOFs derived from interfacial assembly strategy, *Chemical Engineering Journal*, 2023, **465**, 143039.

29 X. Yang, L. Zhao, F. Peng, Y. Zhu and G. Wang, Co-based metal-organic framework with phosphonate and triazole structures for enhancing fire retardancy of epoxy resin, *Polymer Degradation and Stability*, 2021, **193**, 109721.

30 Z.-B. Shao, J. Zhang, R.-K. Jian, C.-C. Sun, X.-L. Li and D.-Y. Wang, A strategy to construct multifunctional ammonium polyphosphate for epoxy resin with simultaneously high fire safety and mechanical properties, *Composites Part A: Applied Science and Manufacturing*, 2021, **149**, 106529.

31 Y. Hou, S. Qiu, Z. Xu, F. Chu, C. Liao, Z. Gui, L. Song, Y. Hu and W. Hu, Which part of metal-organic frameworks affects polymers' heat release, smoke emission and CO production behaviors more significantly, metallic component or organic ligand?, *Composites Part B: Engineering*, 2021, **223**, 109131.

32 Y. Hou, L. Liu, S. Qiu, X. Zhou, Z. Gui and Y. Hu, DOPO-Modified Two-Dimensional Co-Based Metal-Organic Framework: Preparation and Application for Enhancing Fire Safety of Poly(lactic acid), *ACS applied materials & interfaces*, 2018, **10**, 8274–8286.

33 Y. Hou, W. Hu, Z. Gui and Y. Hu, Preparation of Metal–Organic Frameworks and Their Application as Flame Retardants for Polystyrene, *Ind. Eng. Chem. Res.*, 2017, **56**, 2036–2045.

34 J. Huang, W. Guo, X. Wang, H. Niu, L. Song and Y. Hu, Combination of cardanol-derived flame retardant with SiO2@MOF particles for simultaneously enhancing the toughness, anti-flammability and smoke suppression of epoxy thermosets, *Composites Communications*, 2021, **27**, 100904.

35 X. Li, F. Zhang, M. Zhang, X. Zhou and H. Zhang, Comparative Study on the Flame Retardancy and Retarding Mechanism of Rare Earth (La, Ce, and Y)-Based Organic Frameworks on Epoxy Resin, *ACS Omega*, 2021, **6**, 35548–35558.

36 A. Kathuria, M. G. Abiad and R. Auras, Toughening of poly(l-lactic acid) with Cu3BTC2 metal organic framework crystals, *Polymer*, 2013, **54**, 6979–6986.

37 R. Huang, X. Guo, S. Ma, J. Xie, J. Xu and J. Ma, Novel Phosphorus-Nitrogen-Containing Ionic Liquid Modified Metal-Organic Framework as an Effective Flame Retardant for Epoxy Resin, *Polymers*, 2020, **12**. DOI: 10.3390/polym12010108.

38 R. Duan, H. Wu, J. Li, Z. Zhou, W. Meng, L. Liu, H. Qu and J. Xu, Phosphor nitrile functionalized UiO-66-NH2/graphene hybrid flame retardants for fire safety of epoxy, *Colloids and Surfaces A: Physicochemical and Engineering Aspects*, 2022, **635**, 128093.

39 X. Chen, X. Chen, S. Li and C. Jiao, Copper metal-organic framework toward flame-retardant enhancement of thermoplastic polyurethane elastomer composites based on ammonium polyphosphate, *Polymers for Advanced Techs*, 2021, **32**, 2829–2842.

40 W. Chen, Y. Jiang, R. Qiu, W. Xu and Y. Hou, Investigation of UiO-66 as Flame Retardant and Its Application in Improving Fire Safety of Polystyrene, *Macromol. Res.*, 2020, **28**, 42–50.

41 X. Fan, F. Xin, W. Zhang and H. Liu, Effect of phosphorus-containing modified UiO-66-NH2 on flame retardant and mechanical properties of unsaturated polyester, *Reactive and Functional Polymers*, 2022, **174**, 105260.

42 Y. Hou, W. Hu, Z. Gui and Y. Hu, A novel Co(II)–based metal-organic framework with phosphorus-containing structure: Build for enhancing fire safety of epoxy, *Composites Science and Technology*, 2017, **152**, 231–242.

43 W. Guo, S. Nie, E. N. Kalali, X. Wang, W. Wang, W. Cai, L. Song and Y. Hu, Construction of SiO2@UiO-66 core–shell microarchitectures through covalent linkage as flame retardant and smoke suppressant for epoxy resins, *Composites Part B: Engineering*, 2019, **176**, 107261.

44 H. Guo, Y. Wang, C. Li and K. Zhou, Construction of sandwich-structured CoAl-layered double hydroxide@zeolitic imidazolate framework-67 (CoAl-LDH@ZIF-67) hybrids: towards enhancing the fire safety of epoxy resins, *RSC advances*, 2018, **8**, 36114–36122.

45 A. Li, W. Xu, R. Chen, Y. Liu and W. Li, Fabrication of zeolitic imidazolate frameworks on layered double hydroxide nanosheets to improve the fire safety of epoxy resin, *Composites Part A: Applied Science and Manufacturing*, 2018, **112**, 558–571.

46 X.-L. Qi, D.-D. Zhou, J. Zhang, S. Hu, M. Haranczyk and D.-Y. Wang, Simultaneous Improvement of Mechanical and Fire-Safety Properties of Polymer Composites with Phosphonate-Loaded MOF Additives, *ACS Appl. Mater. Interfaces*, 2019, **11**, 20325–20332.

47 H. Nabipour, X. Wang, L. Song and Y. Hu, Organic-inorganic hybridization of isoreticular metal-organic framework-3 with melamine for efficiently reducing the fire risk of epoxy resin, *Composites Part B: Engineering*, 2021, **211**, 108606.

48 W. Meng, H. Wu, X. Bi, Z. Huo, J. Wu, Y. Jiao, J. Xu, M. Wang and H. Qu, Synthesis of ZIF-8 with encapsulated hexachlorocyclotriphosphazene and its quenching mechanism for flame-retardant epoxy resin, *Microporous and Mesoporous Materials*, 2021, **314**, 110885.

49 Z. Qian, B. Zou, Y. Xiao, S. Qiu, Z. Xu, Y. Yang, G. Jiang, Z. Zhang, L. Song and Y. Hu, Targeted modification of black phosphorus by MIL-53(Al) inspired by “Cannikin's Law” to achieve high thermal stability of flame retardant polycarbonate at ultra-low additions, *Composites Part B: Engineering*, 2022, **238**,

109943.

50 L. Sang, Y. Cheng, R. Yang, J. Li, Q. Kong and J. Zhang, Polyphosphazene-wrapped Fe–MOF for improving flame retardancy and smoke suppression of epoxy resins, *J Therm Anal Calorim*, 2021, **144**, 51–59.

51 T. Sai, S. Ran, Z. Guo and Z. Fang, A Zr-based metal organic frameworks towards improving fire safety and thermal stability of polycarbonate, *Composites Part B: Engineering*, 2019, **176**, 107198.

52 S. Qiu, Y. Zhou, X. Zhou, T. Zhang, C. Wang, R. K. K. Yuen, W. Hu and Y. Hu, Air-Stable Polyphosphazene-Functionalized Few-Layer Black Phosphorene for Flame Retardancy of Epoxy Resins, *Small*, 2019, **15**. DOI: 10.1002/smll.201805175.

53 X. Lu, A. F. Lee and X. Gu, Improving the flame retardancy of sustainable lignin-based epoxy resins using phosphorus/nitrogen treated cobalt metal-organic frameworks, *Materials Today Chemistry*, 2022, **26**, 101184.

54 X. Liu, P. Guo, B. Zhang and J. Mu, A novel ternary inorganic–organic hybrid flame retardant containing biomass and MOFs for high-performance rigid polyurethane foam, *Colloids and Surfaces A: Physicochemical and Engineering Aspects*, 2023, **671**, 131625.

55 Y. Li, X. Li, Y.-T. Pan, X. Xu, Y. Song and R. Yang, Mitigation the release of toxic PH3 and the fire hazard of PA6/AHP composite by MOFs, *Journal of Hazardous Materials*, 2020, **395**, 122604.

56 Lu Zhang, Siqi Chen, Ye-Tang Pan, Shuidong Zhang, Shibin Nie, Ping Wei, Xiuqin Zhang, Rui Wang and and De-Yi Wang, Nickel Metal-Organic Framework derived Hierarchically Mesoporous Nickel Phosphate towards Smoke Suppression and Mechanical Enhancement of Intumescent Flame Retardant Wood Fiber/Poly(lactic acid) Composites.

57 T. Ma, W. Wang and R. Wang, Thermal Degradation and Carbonization Mechanism of Fe-Based Metal-Organic Frameworks onto Flame-Retardant Polyethylene Terephthalate, *Polymers*, 2023, **15**. DOI: 10.3390/polym15010224.

58 X. Lv, W. Zeng, Z. Yang, Y. Yang, Y. Wang, Z. Lei, J. Liu and D. Chen,

Fabrication of ZIF-8@Polyphosphazene core-shell structure and its efficient synergism with ammonium polyphosphate in flame-retarding epoxy resin, *Polymers for Advanced Techs*, 2020, **31**, 997–1006.

59 Lu Zhang, Siqi Chen, Ye-Tang Pan, Shuidong Zhang, Shibin Nie, Ping Wei, Xiuqin Zhang, Rui Wang and and De-Yi Wang, Nickel Metal-Organic Framework derived Hierarchically Mesoporous Nickel Phosphate towards Smoke Suppression and Mechanical Enhancement of Intumescent Flame Retardant Wood Fiber/Poly(lactic acid) Composites.

Python Codes

```
# -*- coding: utf-8 -*-
"""
Created on Fri Oct 13 11:38:23 2023

@author: junchen.xiao



offline use


Machine learning models for looped predicting:
    CCT properties of TTI, pHRR, THR & Mecha. prop of StorageModulus at RT
based on RF, SVM
"""


#%% 1 Preparation of Dataset
import os
import time
import copy
import pickle
# import math
import numpy as np
import pandas as pd
import matplotlib as mpl
import shap
from matplotlib.ticker import AutoMinorLocator
import matplotlib.pyplot as plt
# import matplotlib.gridspec as gridspec
import seaborn as sns
```

```
# from adjustText import adjust_text # adjust text in plots
# from scipy import stats
# import matplotlib.ticker as ticker
from matplotlib.colors import ListedColormap
from imblearn.over_sampling import SMOTE
# from sklearn.cluster import KMeans
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
# from sklearn.preprocessing import LabelBinarizer
from sklearn import preprocessing
# from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error
from sklearn.metrics import confusion_matrix
# import py file as module
import sys
sys.path.append('C:\\LocalML\\Scikit_Algorithms')
import warnings
warnings.filterwarnings('ignore') # ignore warnings
# import MOD_GBC as est_gbc
import MOD_RFC as est_rfc
import MOD_RFR as est_rfr
import MOD_SVC as est_svc
import MOD_SVR as est_svr
import MOD_MLC as est_mlc
import MOD_MLR as est_mlr
import MOD_LGBC as est_lgbc
import MOD_LGBR as est_lgbr
import Curve_drawing as hrr_cd
```

```
#%%% 1.1 Import Excel
os.chdir('C:\\LocalML')
mof_df = pd.read_excel('Systems Composites Dataset.xlsx', sheet_name='MOF pHRR TTI')
df_mainFR = pd.read_excel('Systems Composites Dataset.xlsx', sheet_name='Add_Elements')
df_moftype = pd.read_excel('Systems Composites Dataset.xlsx', sheet_name='MOF_Types')
df_mattype = pd.read_excel('Systems Composites Dataset.xlsx', sheet_name='Matrix info')
#%%% 1.2 Graphic setting
# set global rcParameters
mpl.rcParams['axes.grid'] = True
mpl.rcParams['grid.color'] = '#d9e2e2'
mpl.rcParams['grid.linestyle'] = '--'
mpl.rcParams['axes.labelpad'] = 8
mpl.rcParams['axes.labelsize'] = 14
markers = 'abcdefghijklmnopqrstuvwxyz' # mark the subplots with a to i
clr_face = ['#8e8bfe', '#fea3a2', '#ffdc54', '#728bba', 'limegreen']
# (198, 151, 208)    (254, 163, 162 )
clr_edge = ['#443ffe', '#f74d4d', '#ffa510', '#23426b']
# (68, 63, 254) (247, 77, 77)
# set the subplots and figsize fixed at 2 rows with subplot size 6*6
subplots_RC = {1:(1,1), 2:(1,2), 3:(1,3), 4:(1,4),
               5:(2,3), 6:(2,3), 7:(2,4), 8:(2,4),
               9:(3,3), 10:(3,4), 11:(3,4), 12:(3,4),
               16:(4,4)}
def MakeFig(cnt, size=6):
    row, col = subplots_RC[cnt][0], subplots_RC[cnt][1]
    fig = plt.figure(figsize=(col*size, row*size), layout='constrained')
```

```
    if row == col:
        row_grid, col_grid = 60, 60
    else:
        row_grid, col_grid = 60, 120
    grid = fig.add_gridspec(row_grid, col_grid) # 60 rows and 120 cols
    axe = [] # save ax into list
    full_row = int(cnt/col) # how many rows are full of axes
    size = int(col_grid/col) # size of single axe
    g_no = int(row_grid/row) # how many girds does a axe take
    # append normal axes
    for i in range(full_row*col):
        ax = fig.add_subplot(grid[g_no*int(i/col):g_no*int(i/col)+g_no,
                                  size*(i%col):size*(i%col+1)])
        axe.append(ax)
    # append abnormal axes if type(cnt/col) not int
    t = cnt-full_row*col
    start_grid = int((col_grid-t*size)/2)
    if row != full_row:
        for i in range(t):
            ax                                                         =
fig.add_subplot(grid[g_no*(int(i/col)+full_row):g_no*(int(i/col)+full_row)+g_no,

size*(i%col)+start_grid:size*(i%col+1)+start_grid])
            axe.append(ax)
    return fig, axe
# Image folder for saving TIFs
folder_name = 'images MOF pub (no Comp, no lgbm)'
folder_path = f'C:\\LocalML\\{folder_name}'
if os.path.exists(folder_path):
    os.chdir(folder_path)
```

```
else:
    os.mkdir(folder_path)
    os.chdir(folder_path)
# validation experimets in excel
val_list = [20, 79, 122, 142, 263, 264, 266, 267]
def RemoveList(main_list, part): # remove part from main_list
    new_list = main_list.copy()
    for i in part:
        if i in new_list:
            new_list.remove(i)
    return new_list
#%%% 1.2 Advance preparation
# func to time the process with space
def TimeGet(process, space_num=0): # add space_num spaces before process string
    t0 = time.time()
    t1 = time.strftime("%m-%d %H:%M", time.localtime(t0))
    process_marker = space_num*' ' + f'{process} {t1}'
    print(process_marker)
    return t0
# func to calculate the time consumed
def TimeCalcu(t0, t1, space_num=4):
    d_t = (t1 - t0)/60
    t0_1 = time.strftime("%m-%d %H:%M", time.localtime(t0))
    t1_1 = time.strftime("%m-%d %H:%M", time.localtime(t1))
    process_marker = space_num*' ' + f'{t0_1} to {t1_1} lasts {d_t:.1f}min'
    print(process_marker)
    return d_t
# func to obtain the ranking order of [t] in [arr]
def IndRenew(arr, t, if_log=0):
    if np.isnan(t):
```

```
            return np.nan, 'Nan_value'
        else:
            t *= 1.01 # rank after same-value-position
            if if_log: # check if log10
                rt = np.log10(t)
                text_list = [f'{10**arr[k]:.2f}-{10**arr[k+1]:.2f}' for k in range(len(arr)-
1)]
                text_list.insert(0, f'<{10**arr[0]:.2f}')
                text_list.append(f'>{10**arr[-1]:.2f}')
            else:
                rt= t
                text_list = [f'{arr[k]:.2f}-{arr[k+1]:.2f}' for k in range(len(arr)-1)]
                text_list.insert(0, f'<{arr[0]:.2f}')
                text_list.append(f'>{arr[-1]:.2f}')
            res_list = sorted(np.append(arr, rt))
            res = res_list.index(rt)
            res_text = text_list[res]
            return res, res_text
# func to insert [values] into [locations] of [given_list]
def InsertIntoList(l0, vals, locs):
        if len(vals) != len(locs):
            print('value-list should have same length as location-list!')
        cnt_ins = len(vals)
        t = 0
        for i in range(cnt_ins):
            l_front = l0[0:locs[i]+t]
            l_front.append(vals[i])
            l_front.extend(l0[locs[i]+t:len(l0)])
            t += 1
            l0 = l_front
```

```
    return l0
# func to fill [ser_lvl] in [df] according to [ser_orig] leveled by [lvl_list]
levels_dict = {}
def LevelTarget(df, ser_orig, lvl_list, if_log=0): #
    list_lvl, ind_text = [], {} # dict with {class x: value range}
    for ki in df[ser_orig]:
        lvl_num, lvl_text = IndRenew(lvl_list, ki, if_log)
        ind_text[lvl_num] = lvl_text
        list_lvl.append(lvl_num)
    lvl_ser = pd.Series(list_lvl) # Series of list_lvl
    counts = lvl_ser.value_counts()
    counts.sort_index(inplace=True)
    cnt_df = pd.DataFrame({'class': counts.index, 'frequency': counts.values})
    ran_df = pd.DataFrame({'class':ind_text.keys(), 'range':ind_text.values()})
    levels = cnt_df.merge(ran_df, on='class') # merge to whole dataframe
    levels_dict[ser_orig] = levels
    df[ser_orig] = list_lvl
# func to reverse the levle to range according to [lvl_list]
def LevelReverse(cat_lvl, lvl_list, if_log, if_range):
    len_lvl = len(lvl_list)
    if if_log:
        lvl_new = list(map(lambda x:round(pow(10,x),1),lvl_list))
    else:
        lvl_new = lvl_list.copy()
    res_txt, res = 0, 0
    if cat_lvl == 0:
        res_txt = f'< {lvl_new[0]}'
        res = int(lvl_new[0]*0.85)
    elif cat_lvl == len_lvl:
        res_txt = f'> {lvl_new[len_lvl-1]}'
```

```
            res = int(lvl_new[len_lvl-1]*1.15)
        else:
            res_txt = f'{lvl_new[cat_lvl-1]} - {lvl_new[cat_lvl]}'
            res = round(((lvl_new[cat_lvl-1]+lvl_new[cat_lvl])/2),2)
        if if_range:
            return res, res_txt
        else:
            return res
# func to get row indices when v-count < cv_cnt in lvl-dataset (for SMOTE)
def IndNotSMOTE(df_y, cv_cnt):
        v_count = df_y.value_counts()
        del_lvl = v_count.apply(lambda x:x<cv_cnt)
        v_del = v_count[del_lvl].index.tolist()
        ind_del=[]
        for ki in v_del:
            bb = df_y.apply(lambda x: x==ki)
            ab = df_y[bb].index.tolist()
            ind_del.extend(ab)
        return ind_del
# func to fill missing data of polymer matrix (specific values)
def FillDict(df, fill_fea, fill_dict, mark='Mat_Prim'):
        for ki in df.index.tolist():
            marker = df.loc[ki, mark]
            if np.isnan(df.loc[ki,fill_fea]):
                df.loc[ki,fill_fea] = fill_dict[marker]
# func to insert new fea and apply func (new col with specific values)
def NewFeature(df, add_fea, loc_fea, apply_fea_list, apply_func):
        loc_add = df.columns.get_loc(loc_fea) + 1
        df.insert(loc_add , add_fea, 0)
        df[add_fea] = df[apply_fea_list].apply(apply_func, axis=1)
```

```
# func to insert multiple COLs in [ins_list] into [df] after [fea_str] with 0
def InsertList(df,fea_str,ins_list):
    len_ins = len(ins_list)
    loc_0 = df.columns.get_loc(fea_str) + 1
    for ki in range(len_ins):
        df.insert(loc_0+ki, ins_list[ki], 0)
# func to divide a/b with only 1 input
def SFdivide(ser):
    a, b = ser.iloc[0], ser.iloc[1]
    res = a / b
    return res
# func to multiply a*b with only 1 input
def SFmultiply(ser):
    a, b = ser.iloc[0], ser.iloc[1]
    res = a * b
    return res
# func to get alternative values
def SFoptionFill(ser):
    a, b = ser.iloc[0], ser.iloc[1]
    if b > 0:
        res = b
    else:
        res = a
    return res
# func to fill t_trans and hrr_trans
def FillTransition(df):
    for i in df.index.tolist():
        if np.isnan(df.loc[i,'t_trans']):
            df.loc[i,'t_trans'] = np.mean([df.loc[i,'t_devlp'], df.loc[i,'t_decay']])
            df.loc[i,'HRR_trans']            =            np.mean([df.loc[i,'HRR_devlp'],
```

```
df.loc[i,'HRR_decay']])
# func to fill with mean values
def FillMeanValue(df, cols=0):
    if not isinstance(cols, list):
        cols = df.columns.tolist()
    for i in range(len(cols)):
        if type(df.iloc[0,i]) != str:
            fea = cols[i]
            mask = df[fea].isnull()
            ind = df[mask].index.tolist()
            v_mean = np.nanmean(df[fea])
            df.loc[ind, fea] = v_mean
#%% 2 Data Pre-processing
TimeGet('Start running whole file')
#%%% 2.1 Subsets Features
# fill supporting DFs
df_mainFR.fillna(0, inplace=True) # 0-fill in df_mainFR
FillTransition(df_mattype) # fill trnas-values
# FillMeanValue(df_mattype)
df_moftype.fillna({'class':'IRM'}, inplace=True)
df_moftype.fillna(0, inplace=True)
# fill THR with either calculated or averaged values
def FillTHR(df):
    hrr_prop = ['t_ignit','t_devlp','t_trans','t_decay','t_after',
                'HRR_devlp','HRR_trans','HRR_decay']
    # fill THR with calculated values from hrr_prop
    mask = (~ df['t_decay'].isnull()) & (df['THR'].isnull())
    ind_mask = df[mask].index.tolist()
    for i in ind_mask:
        _, _, integral = hrr_cd.xy_hrr(df[hrr_prop].loc[i])
```

```
            df.loc[i, 'THR'] = integral
        # fill THR with mean values of same polymer
        mask = df['THR'].isnull()
        ind_mask = df[mask].index.tolist()
        polymer_mat = df['Polymer'].value_counts().index.tolist()
        for i in ind_mask:
            if df.loc[i,'Polymer'] in polymer_mat:
                df.loc[i, 'THR'] = np.mean(df[df['Polymer'] == df.loc[i, 
'Polymer']]['THR'])
FillTHR(df_mattype)
# add mof, mfr and total mass in mof_df
NewFeature(mof_df, 'MOF_Mass', 'MOF_Mass1', ['MOF_Mass0', 'MOF_Mass1'], 
np.sum)
NewFeature(mof_df, 'MFR_Mass', 'MFR_Mass1', ['MFR_Mass0', 'MFR_Mass1'], 
np.sum)
NewFeature(mof_df, 'Add_Mass', 'MFR_Hybrid', ['MOF_Mass', 'MFR_Mass'], 
np.sum)
# preparation for additional properties from sub-dataframes
# additional MATRIX features
mat_cols = ['Mat_Dense', 'Mat_tCond', 'Mat_MP', 'Mat_tpeak', 'Mat_Modulus', 
'Mat_Tensile',
            'Mat_CharAir', 'Mat_CharN2', 'Mat_tig', 'Mat_tfu', 'Mat_ttr', 'Mat_tde', 
'Mat_taf',
            'Mat_hfu', 'Mat_htr', 'Mat_hde', 'Mat_THR', 'Mat_GlassT', 'Mat_T5%', 
'Mat_T50%',
            'Mat_Tmlr']
InsertList(mof_df, 'Mat_LOI', mat_cols)
sub_key_mat = ['Mat_Prim', 'Lit.', 'Polymer','literature']
sub_mat = ['Density', 'TherCond', 'Melting', 't_peak', 'StorageM', 'Tensile', 'Char_Air',
           'Char_N2', 't_ignit', 't_devlp', 't_trans', 't_decay', 't_after', 'HRR_devlp',
```

```
                 'HRR_trans', 'HRR_decay', 'THR', 'GlassTrans', 'T5%', 'T50%', 'Tmlr']
# additional MOF features
mof_cols = ['MOF_Matom','MOF_Mgrp','MOF_Link-N','MOF_Link-O','MOF_Linker', 'MOF_LinkerC',
                 'MOF_LinkerO', 'MOF_LinkerN', 'MOF_T5%', 'MOF_Tmlr', 'MOF_Resid',
                 'MOF_Benze', 'MOF_Carbo', 'MOF_CN', 'MOF_Phop', 'MOF_Class', 'MOF_Metal']
InsertList(mof_df, 'MOF_Size', mof_cols)
sub_key_mof = ['MOF_Name', 'MOF_Mass', 'mof_name']
sub_mof = ['Matom', 'Mgroup', 'link_N', 'link_O', 'linker_mol', 'linker_C',
                 'linker_O', 'linker_N', 'T5%', 'Tmlr', 'Char',
                 'linker_Benz', 'linker_Carboxy', 'linker_CN', 'linker_PO', 'class', 'metal']
# additional MFR features
MFR_cols = ['MFR_B', 'MFR_C', 'MFR_N', 'MFR_O', 'MFR_P', 'MFR_Si', 'MFR_Met']
InsertList(mof_df, 'MFR_Mass', MFR_cols)
sub_key_mfr = ['MFR', 'MFR_Mass', 'Add']
sub_mfr = ['B', 'C', 'N', 'O', 'P', 'Si', 'Metal']
# func to add additional features from MATRIX sub-DF to main DF
def MatFeat(df, df_sub, sub_key, sub_list, list_main):
    name_df, lit_df = sub_key[0], sub_key[1]
    name_sub, lit_sub = sub_key[2], sub_key[3]
    # get the index according to the name and lit
    for ki in range(df.shape[0]):
        mask0 = (df_sub[name_sub]==df.loc[ki, name_df]) & (df_sub[lit_sub]==df.loc[ki, lit_df])
        df_ind = df_sub[mask0].index.tolist()
        if len(df_ind):
            ind = df_ind[0]
```

```
                value_list = [df_sub.loc[ind, t] for t in sub_list]
            else:
                mask1 = (df_sub['Polymer']==name_df)
                value_list = [np.mean(df_sub[mask1][t]) for t in sub_list]
            # assign values
            df.loc[ki, list_main] = value_list
# func to deal with multiple sub_types with sum
def AdditionalSum(lists):
    a0 = np.array(lists).T
    res = np.sum(a0, axis=1).tolist()
    return res
# func to deal with multiple sub_types with sum/compare/divide
def AdditionalMof(lists):
    a0 = pd.DataFrame(lists).T
    # not all rows are to make SUM, some of them are taking Minimum or Str
    res = []
    res_add = res.append
    for i in range(a0.shape[0]):
        if i in [8, 9]:    # 8 and 9 for 'T5%' and 'Tmlr', taking minimum
            res_add(min(a0.iloc[i, :]))
        elif i in [15, 16]: # 15 and 16 for 'class' and 'metal' strings
            if a0.iloc[i, 0]==a0.iloc[i, 1]: # check if they hace same 'class' or 'metal'
                res_add(a0.iloc[i, 0])
            else:
                res_add('+'.join(a0.iloc[i, :]))
        else:
            res_add(sum(a0.iloc[i, :]))
    return res
# func to add additional features from ADDITIVES sub_DFs to main DF
def AddsFeat(df, df_sub, sub_key, sub_list, sub_func, list_main, if_str=0):
```

```
    type_df, mass_df, name_sub = sub_key[0], sub_key[1], sub_key[2]
    # get the corresponding row index having type_df
    df_ind = df[df[type_df].str.len() >0].index.tolist()
    # no_sub = df[~(df[type_df].str.len() >0)].index.tolist() # no sub type
    # get values for each sub-faetures
    for ki in df_ind:
        add_list = df.loc[ki, type_df].split('+')
        sum_mass = df.loc[ki, mass_df]
        sub_res = [] # save value_list in list
        # loop (multiple) sub_types (i.e. 'ZIF8+ZIF67','APP+PER')
        for kj in range(len(add_list)):
            add_ratio = (df.loc[ki, mass_df+str(kj)])/sum_mass # ratio
            # add_ratio = df.iloc[i,mass_df+str(kj)] # amount
            ind_add = df_sub[df_sub[name_sub] == add_list[kj]]
            if len(ind_add)==0:
                print(f'No record of {add_list[kj]} at {ki+2}')
            if not if_str:
                value_list = [add_ratio*df_sub.loc[ind_add.index[0], t] for t in 
sub_list]
            else:
                value_list = [add_ratio*df_sub.loc[ind_add.index[0], sub_list[t]]
                              for t in range(len(sub_list)-2)]
                value_list.append(df_sub.loc[ind_add.index[0], sub_list[-2]])
                value_list.append(df_sub.loc[ind_add.index[0], sub_list[-1]])
            # ind_add = df_sub[df_sub[name_sub] == add_list[kj]].index[0]
            # value_list = [add_ratio*df_sub.loc[ind_add,t] for t in sub_list]
            sub_res.append(value_list)
        # combine values from multiple sub_types (len(add_list)>1)
        if len(add_list)>1:
            res_list = sub_func(sub_res)
```

```
        else:
            res_list = sub_res[0]
        df.loc[ki, list_main] = res_list
# add additional features
MatFeat(mof_df, df_mattype, sub_key_mat, sub_mat, mat_cols)
AddsFeat(mof_df, df_moftype, sub_key_mof, sub_mof, AdditionalMof, mof_cols, 1)
AddsFeat(mof_df, df_mainFR, sub_key_mfr, sub_mfr, AdditionalSum, MFR_cols)
# fill missing cells
FillTransition(mof_df) # pHRR and time in transition phase
# convert Com_CharAir to Comp_CharN2
mat_charAN = df_mattype.groupby('Polymer')[['Char_Air','Char_N2']].agg('mean')
for i in range(mof_df.shape[0]):
    if (not np.isnan(mof_df.loc[i,'Comp_CharAir'])) and np.isnan(mof_df.loc[i,'Comp_CharN2']):
        char_conv0 = mat_charAN.loc[mof_df.loc[i,'Mat_Prim'],'Char_N2'] # matrix char in N2
        char_conv1 = mat_charAN.loc[mof_df.loc[i,'Mat_Prim'],'Char_Air'] # matrix char in Air
        char_conv2 = (100 - mof_df.loc[i,'Add_Mass'])/100 # matrix percent
        char_conv3 = mof_df.loc[i,'Comp_CharAir'] - char_conv1 # FR contributed char
        mof_df.loc[i,'Comp_CharN2'] = char_conv0 * char_conv2 + char_conv3
# fill others with mean
def FillMiss(df): # fill mean values of Matrix- and Comp-related features
    mats, comps = [], []
    for i in df.columns.tolist():
        if i[0:3]=='Mat' and isinstance(df[i][0], float):
            mats.append(i)
        if i[0:3]=='Com' and isinstance(df[i][0], float):
            comps.append(i)
```

```
    mats_dict, comps_dict = {}, {}
    for i in mats:
        mats_dict[i] = df.groupby('Mat_Prim')[i].agg('mean').to_dict()
        FillDict(df, i, mats_dict[i])
    for i in comps:
        comps_dict[i] = df.groupby('Lit.')[i].agg('mean').to_dict()
        FillDict(df, i, comps_dict[i], 'Lit.')
    # fill res with mean of all
    FillMeanValue(df, mats)
    FillMeanValue(df, comps)
FillMiss(mof_df)
# func to check NAN cells
def CheckNan(df):
    for i in df.columns.tolist():
        t = 0
        if len(df[i].isnull().value_counts()) > 1 :
            nan_ratio = df[i].isnull().value_counts()[1]/df.shape[0]
            print(f'{i}: {nan_ratio:.2f}')
            t += 1
    if t == 0:
        print(f'{str(df)} is Full DataFrame')
# CheckNan(mof_df)
#%%% 2.2 Additional Features
# additional features for further use
NewFeature(mof_df, 'Mat_pc', 'Mat_tCond', ['Mat_Dense', 'Mat_tCond'], SFmultiply)
NewFeature(mof_df, 'Mat_Char', 'Mat_Tensile', ['Mat_CharAir', 'Mat_CharN2'], SFoptionFill)
NewFeature(mof_df, 'Comp_Char', 'PP_Tmold', ['Comp_CharAir', 'Comp_CharN2'], SFoptionFill)
NewFeature(mof_df, 'PP_Tsyn','PP_Tmold', ['PP_Tmix', 'PP_Tmold'], np.max)
```

```
NewFeature(mof_df, 'MFR_NP', 'MFR_Met', ['MFR_N', 'MFR_P'], np.sum)
NewFeature(mof_df, 'MFR_BSi', 'MFR_Met', ['MFR_B', 'MFR_Si'], np.sum)
NewFeature(mof_df, 'MFR_NPBSi', 'MFR_Met', ['MFR_N', 'MFR_P', 'MFR_B',
'MFR_Si'], np.sum)
# fill specific features with specific values
fill_values = {'MOF_Name': 'No MOF', 'MOF_Size': 1000, 'MOF_Mass0': 0,
'MOF_Mass1': 0,
               'MFR_Mass0': 0, 'MFR_Mass1': 0, 'MFR': 'No MFR',
'MFR_Hybrid': 0,
               'CCT_HeatFlux': 35, 'CCT_Thick': 4, 'PP_Mix': 'extrusion',
'PP_Tmix': 25,
               'PP_Mold': 'pressing', 'PP_Tmold': 25, 'PP_Tsyn': 25,
               'DMA_Mode': 'stretch', 'DMA_Gfactor': 0.006}
mof_df.fillna(value=fill_values, inplace=True)
# Type metal by the GROUP, func to group metals
def MetalGroup(a):
    if a>0:
        grp = round(a)
    else:
        grp = 0
    return grp
mof_df['MOF_Mgrp'] = mof_df['MOF_Mgrp'].apply(MetalGroup)
# set MOF-related marks to 0 if 'No MOF' in 'MOF_Name'
msk_nomof = mof_df['MOF_Name']=='No MOF'
mof_df.loc[msk_nomof,'MOF_Size'] = 0
mof_df.loc[msk_nomof,'MOF_Class'] = '0_No_Class'
mof_df.loc[msk_nomof,'MOF_Metal'] = '0_No_Metal'
# split metal in MOF
def SplitMetal0(a):
    aa = a.split('+')
```

```
    return aa[0]
def SplitMetal1(a):
    aa = a.split('+')
    if len(aa)>1:
        return aa[1]
    else:
        return np.nan
mof_df['MOF_Metal2'] = mof_df['MOF_Metal'].apply(SplitMetal1)
mof_df['MOF_Metal'] = mof_df['MOF_Metal'].apply(SplitMetal0) # keep only first
#%%% 2.3 Encoding and conversion
# Label Encoding of categorical features in X
def OHEncode(feats, df0): # func to make one-hot encoding
    cats, encs = {}, {}
    x_oh = df0[feats].to_numpy() #convert all OH features to numpy 2D array
    enc = OneHotEncoder(drop='first') # create Encoder
    enc_fit = enc.fit(x_oh)
    df_trans = pd.DataFrame(enc_fit.transform(x_oh).toarray(),

columns=enc_fit.get_feature_names_out(input_features=feats),
                              index=df0.index)
    # split dataframe to insert transformation before first feature
    loc = df0.columns.get_loc(feats[0])
    df_1 = df0.iloc[:, 0:loc]
    df_2 = df0.iloc[:, loc:]
    df = pd.concat([df_1, df_trans, df_2], axis=1)
    for i in range(len(feats)):
        cats[feats[i]] = enc_fit.categories_[i]
        encs[feats[i]] = enc_fit
    return df, cats, encs
def LabEncode(feats, df0): # func to make label encoding
```

```
        df = copy.deepcopy(df0)
        cats, encs = {}, {}
        for fea in feats:
                enc = LabelEncoder()
                res = enc.fit_transform(df[fea])
                cats[fea] = enc.classes_
                encs[fea] = enc
                df[fea] = res
        return df, cats, encs
def Encode(fea_cat, df0): # OH encode first and then Labeled
        cats, encs = {}, {}
        if len(fea_cat)>0:
                df, cats_oh, encs_oh = OHEncode(fea_cat, df0)
                df2, cats_lab, encs_lab = LabEncode(fea_cat, df)
                cats = {'OneHot': cats_oh, 'Label': cats_lab}
                encs = {'OneHot': encs_oh, 'Label': encs_lab}
                return df2, cats, encs
        else:
                print('No features reqiured to be encoded')
# start encoding
enc_cates, enc_dict = {}, {}
enc_feats = ['MOF_Class', 'DMA_Mode', 'PP_Mix', 'PP_Mold']
mof_df_enc, enc_cates, enc_dict = Encode(enc_feats, mof_df)
# convert original TARs to d(divided by matrix)/c(categorized) TARs
def ConvCate(df, list_a, arr_list, if_log): # categorical leveling
        for i in range(len(list_a)):
                d1 = f'{list_a[i]}_c'
                df[d1] = df[list_a[i]]
                LevelTarget(df, d1, arr_list[i], if_log[i])
def ConvDivd(df, list_a): # divided by matrix values
```

```
    for i in list_a:
        d1 = f'{i}_d'
        divd = f'Mat_{i}'
        df[d1] = df[[i, divd]].apply(SFdivide, axis=1)
def ConvDC(df, list_a, arr_list, if_log): # divided first and then leveled
    for i in range(len(list_a)):
        d1, d2 = f'{list_a[i]}_d', f'{list_a[i]}_dc'
        divd = f'Mat_{list_a[i]}'
        df[d1] = df[[list_a[i], divd]].apply(SFdivide, axis=1)
        df[d2] = df[d1] # add column named d2 for LevelTarget
        LevelTarget(df, d2, arr_list[i], if_log[i])
# start converting
tar_feat = ['Tensile_dc', 'Modulus_dc', 'TTI_dc', 'pHRR_dc', 'THR_dc'] # new names
for modelling
cnt_tar = len(tar_feat)
ConvDC(mof_df_enc, ['Tensile', 'Modulus', 'TTI', 'pHRR', 'THR'], # TARGET_dc
       [np.arange(0.5, 1.9, 0.2),
        [1.0, 1.15, 1.35, 1.7], np.arange(0.70, 1.30, 0.1),
        np.arange(0.30, 1.00, 0.1),
        [0.60, 0.70, 0.80, 0.90, 1.00]], [0 for i in range(5)])
# mof_show
mof_show = mof_df_enc.copy() # containing all features and targets for presentation
mof_show['MOF_Class'][mof_show['MOF_Class']=='0_No_Class'] = 'No_MOF'
# unwated ROWs
index_drop = {i:mof_df_enc[mof_df_enc[i].isnull()].index.tolist() for i in tar_feat}
index_drop['TTI_dc'].extend([208, 209, 210, 211, 212, 221, 222, 223]) # thickness =
10
#%%% 2.4 Individual Dataframes (splitting and scaling)
# split mof_df into individual dataframes for each target features and drop unwanted
COLs/ROWs
```

```
def DFsegment(df, ref_list, ax=0, seg_meth=0): # func to segment [df] with [ref_list]
    a = copy.deepcopy(df)
    if not seg_meth:
        a.drop(ref_list, axis=ax, inplace=True)
        return a
    else:
        b = a[ref_list]
        return b
# unwanted COLs
col_drop = ['Lit.', 'Mat_Prim', 'Mat_comments', 'Mat_CharAir',
            'Mat_tig', 'Mat_tfu', 'Mat_ttr', 'Mat_tde', 'Mat_taf', 'Mat_hfu', 'Mat_htr',
'Mat_hde',
            'MOF_Name', 'MOF_Mass0', 'MOF_Mass1', 'MOF_Metal',
            'MFR', 'MFR_Mass0', 'MFR_Mass1', 'Add_Mass',
            'PP_Tmix',    'PP_Tmold',    'Comp_CharAir',    'Comp_CharN2',
'DMA_Ampli']
# drop and split
mof_df_dropCOL = mof_df_enc.drop(col_drop, axis=1)
loc_xy = mof_df_dropCOL.columns.get_loc('Tensile')
mof_x_full, mof_y_full = [], []
for i in range(cnt_tar):
    mof_x_full.append(DFsegment(mof_df_dropCOL.iloc[:,          0:loc_xy],
index_drop[tar_feat[i]]))
    mof_y_full.append(DFsegment(mof_df_dropCOL[tar_feat[i]],
index_drop[tar_feat[i]]))
# delete columns with over 10% vacancies
for i in range(cnt_tar):
    for j in mof_x_full[i].columns.tolist():
        vacancy_col = mof_x_full[i][j].isnull().value_counts()
        if vacancy_col.shape[0] > 1:
```

```
            if vacancy_col[1]/(mof_x_full[i].shape[0]) > 0.15:
                mof_x_full[i].drop(j, axis=1, inplace=True)
    FillMeanValue(mof_x_full[i])
#%%% 2.4 Validation set
val_dic, model_dic = {}, {}
for i in range(cnt_tar):
    val_dic[tar_feat[i]] = val_list
    model_dic[tar_feat[i]] = RemoveList(mof_x_full[i].index.tolist(),
                                         val_dic[tar_feat[i]])
#%% 3 Final Preparation
#%%% 3.1 Correlation analysis
clr_new0 = np.transpose(np.array([np.linspace(0.556862, 1, 100),
                                  np.linspace(0.545098, 1, 100),
                                  np.linspace(0.996078, 1, 100),
                                  np.ones(100)]))
clr_new1 = np.transpose(np.array([np.linspace(1, 0.996078, 100),
                                  np.linspace(1, 0.639216, 100),
                                  np.linspace(1, 0.635294, 100),
                                  np.ones(100)]))
clr_new = np.concatenate((clr_new0, clr_new1))
clr_new_map = ListedColormap(clr_new)
corr_method = {0: 'pearson', 1: 'spearman', 2: 'kendall'}
def CorrelationAnal(df, sprm_pear=1, img_close=1, clr=clr_new_map):
    # f_corr, ax_corr = plt.subplots(1,1,figsize=(12, 12))
    f_corr, ax_corr = MakeFig(1, 12)
    # plt.subplots_adjust()
    corr_df = df.copy()
    corr_df = df.corr(method=corr_method[sprm_pear])
    feat_range = df.columns.tolist()
    img = ax_corr[0].imshow(corr_df, interpolation="nearest", cmap=clr, vmin=-1,
```

```
vmax=1)
        # set tick lables to faeture names alone x and y axis
        ax_corr[0].set_xticks(np.arange(len(feat_range)), feat_range, rotation=90)
        ax_corr[0].set_yticks(np.arange(len(feat_range)), feat_range)
        # set minor-ticks, show minor grid and hide major grid
        ax_corr[0].minorticks_on() # turn on minor ticks before setting
        ax_corr[0].xaxis.set_minor_locator(AutoMinorLocator(2)) # locate minors at 1/2,
default 4 or 5
        ax_corr[0].xaxis.grid(False, which='major') # turn off major grid of x
        ax_corr[0].xaxis.grid(True, which='minor') # turn on minor grid of x
        ax_corr[0].yaxis.set_minor_locator(AutoMinorLocator(2))
        ax_corr[0].yaxis.grid(False, which='major') # turn off major grid of y
        ax_corr[0].yaxis.grid(True, which='minor') # turn on minor grid of y
        f_corr.colorbar(img, ax=ax_corr) # colobar setting
        if img_close:
                plt.close()
        return f_corr
#%%% 3.2 Final Datasets for ML
# func to SMOTE DFs for classification
def SMOTExy(x, y): # output y is str type in dataframe
        k0 = min(y.value_counts().tolist()[-1], 4)
        k = max(k0, 2) # k should be between 2 and min_value_counts
        drop_smote = IndNotSMOTE(y, k+1)
        x.drop(drop_smote, inplace=True)
        y.drop(drop_smote, inplace=True)
        sm_est = SMOTE(k_neighbors=k, random_state=42)
        a, b = sm_est.fit_resample(x, y)
        c = b.apply(lambda x:int(x)) # convert output y to int type in dataframe
        return a, c, k
# select features
```

```
# Storage Modulus
e0 = ['Mat_Modulus', 'Mat_Dense', 'Mat_GlassT',
        'MOF_Mass', 'MOF_Size', 'MOF_Class', 'MOF_Mgrp', 'MOF_Linker',
        'MFR_Mass', 'MFR_NPBSi', 'MFR_Hybrid',
        'PP_Tsyn', 'DMA_Mode', 'DMA_Gfactor']
e1 = ['Mat_Modulus', 'Mat_Dense', 'Mat_GlassT',
        'MOF_Mass', 'MOF_Size', 'MOF_Class', 'MOF_Mgrp', 'MOF_Linker',
        'MFR_Mass', 'MFR_NPBSi', 'MFR_Hybrid',
        'PP_Tsyn', 'DMA_Mode', 'DMA_Gfactor']
e2 = ['Mat_Modulus', 'Mat_Dense', 'Mat_GlassT',
        'MOF_Mass',     'MOF_Size','MOF_Class_UIO',     'MOF_Class_ZEO',
'MOF_Class_MIL','MOF_Class_IRM',
        'MOF_Mgrp','MOF_Linker',
        'MFR_Mass', 'MFR_NPBSi', 'MFR_Hybrid',
        'PP_Tsyn', 'DMA_Mode', 'DMA_Gfactor']
# TTI
t0 = [ 'Mat_TTI', 'Mat_T5%', 'Mat_Tmlr', 'Mat_MP', 'Mat_Char', 'Mat_Dense',
        'MOF_Mass', 'MOF_Size', 'MOF_Mgrp', 'MOF_Linker',
        'MOF_T5%', 'MOF_Tmlr', 'MOF_Resid',
        'MFR_Mass', 'MFR_NPBSi', 'PP_Tsyn', 'CCT_HeatFlux', 'CCT_Thick']
t1 = [ 'Mat_TTI', 'Mat_T5%', 'Mat_Tmlr', 'Mat_MP', 'Mat_Char', 'Mat_Dense',
        'MOF_Mass', 'MOF_Size', 'MOF_Mgrp', 'MOF_Linker',
        'MOF_T5%', 'MOF_Tmlr', 'MOF_Resid',
        'MFR_Mass', 'MFR_NPBSi','PP_Tsyn', 'CCT_HeatFlux', 'CCT_Thick']
t2 = [ 'Mat_TTI', 'Mat_T5%', 'Mat_Tmlr', 'Mat_MP', 'Mat_Char', 'Mat_Dense',
        'MOF_Mass', 'MOF_Size', 'MOF_Class', 'MOF_Mgrp', 'MOF_Linker',
        'MOF_T5%', 'MOF_Tmlr', 'MOF_Resid',
        'MFR_Mass', 'MFR_NPBSi','PP_Tsyn', 'CCT_HeatFlux', 'CCT_Thick']
# pHRR
hr0 = ['Mat_pHRR', 'Mat_Char', 'Mat_Tmlr',
```

```
            'MOF_Mass', 'MOF_Size', 'MOF_Linker', 'MOF_T5%', 'MOF_Tmlr',
'MOF_Resid',
            'MOF_Mgrp', 'MFR_Mass', 'MFR_C', 'MFR_NPBSi',
            'MFR_Hybrid', 'CCT_Thick', 'PP_Tsyn']
hr1 = ['Mat_pHRR', 'Mat_Char', 'Mat_Tmlr',
            'MOF_Mass', 'MOF_Size', 'MOF_T5%', 'MOF_Tmlr', 'MOF_Resid',
'MOF_Mgrp',
            'MFR_Mass', 'MFR_C', 'MFR_NPBSi', 'MFR_Hybrid',
            'CCT_Thick', 'PP_Tsyn']
hr2 = ['Mat_pHRR','Mat_TTI', 'Mat_LOI', 'Mat_Char', 'Mat_T5%', 'Mat_Tmlr',
            'MOF_Mass', 'MOF_Size', 'MOF_T5%', 'MOF_Tmlr', 'MOF_Resid',
'MOF_Class', 'MOF_Mgrp',
            'MFR_Mass', 'MFR_B', 'MFR_C', 'MFR_N', 'MFR_P', 'MFR_Si',
            'MFR_Hybrid', 'CCT_HeatFlux', 'CCT_Thick', 'PP_Tsyn']
# THR
th0 = ['Mat_THR', 'Mat_tCond', 'Mat_MP',
            'Mat_Char', 'Mat_Tmlr', 'MOF_Mass', 'MOF_Size',
            'MOF_Linker', 'MOF_T5%', 'MOF_Tmlr', 'MOF_Resid',
            'MOF_Mgrp', 'MFR_Mass', 'MFR_NPBSi','MFR_Hybrid', 'CCT_Thick',
            'PP_Tsyn']
th1 = ['Mat_THR', 'Mat_tCond', 'Mat_MP',
            'Mat_Char', 'Mat_Tmlr', 'MOF_Mass', 'MOF_Size',
            'MOF_T5%', 'MOF_Tmlr', 'MOF_Resid', 'MOF_Mgrp',
            'MFR_Mass', 'MFR_NPBSi', 'MFR_Hybrid',
            'CCT_Thick', 'PP_Tsyn']
th2 = ['Mat_THR', 'Mat_tCond', 'Mat_MP', 'Mat_Char',
            'Mat_Tmlr', 'MOF_Mass', 'MOF_Size', 'MOF_Linker',
            'MOF_T5%', 'MOF_Tmlr', 'MOF_Resid',
            'MOF_Class', 'MOF_Mgrp',
            'MFR_Mass', 'MFR_NPBSi', 'MFR_Hybrid', 'CCT_Thick', 'PP_Tsyn']
```

```
feat_x = [e0, e2, t0, t2, hr0, hr2, th0, th2] # initial feat_x
tar_spec = ([f'Modulus({i})' for i in ['RF','SVM']] +
            [f'TTI({i})' for i in ['RF','SVM']] +
            [f'pHRR({i})' for i in ['RF','SVM']] +
            [f'THR({i})' for i in ['RF','SVM']])
cnt_tar = len(tar_spec)
# Final DataFrames
# func to get final modelling DFs
def MakeModelDF(x, y):
    df_x, df_y = [], []
    for i in range(cnt_tar):
        ind_xy = model_dic[tar_feat[int(i/2)+1]]
        # x_spec = x[i]
        # y_spec = y[i]
        df_x.append(x[i].loc[ind_xy])
        df_y.append(y[int(i/2)+1].loc[ind_xy])
    return df_x, df_y
# func to get final validation DFs
def MakeValidatelDF(x, y):
    df_x, df_y = [], []
    for i in range(cnt_tar):
        ind_xy = val_dic[tar_feat[int(i/2)+1]]
        df_x.append(x[i].loc[ind_xy])
        df_y.append(y[int(i/2)+1].loc[ind_xy])
    return df_x, df_y
# scale datasets
def Scaling():
    sca, dfs = [], []
    for i in range(cnt_tar):
```

```
        scaler = preprocessing.MinMaxScaler()
        df = mof_x_full[int(i/2)+1].loc[:,feat_x[i]]
        dfs.append(pd.DataFrame(scaler.fit_transform(df),
                                columns = df.columns,
                                index = df.index))
        sca.append(scaler)
    return sca, dfs
scalers, mof_x_scal = Scaling()
mof_x, mof_y = MakeModelDF(mof_x_scal, mof_y_full)
mof_x_val, mof_y_val = MakeValidatelDF(mof_x_scal, mof_y_full)
# expand the mof_x nad mof_y
x_dfs = [mof_x[i] for i in range(cnt_tar)]
y_dfs = [mof_y[i].copy() for i in range(cnt_tar)]
x_dfs_val = [mof_x_val[i] for i in range(cnt_tar)]
y_dfs_val = [mof_y_val[i].copy() for i in range(cnt_tar)]
#%% 4 Dataset Presentation
TimeGet('Dataset presentation:')
# func to create palette from array
def ColorPalette(arr, pal_name='ML01', pal_cnt=12):
    mpl.colormaps.unregister(pal_name) # avoid repeatd register, won't raise error if
no pal_name
    l_cmap = ListedColormap(arr, pal_name)
    mpl.colormaps.register(l_cmap)
    res = sns.color_palette(pal_name, n_colors=pal_cnt)
    return res
# creata a color palette from clr_edge
# (68, 63, 254) (247, 77, 77)
clr_cnt = 12
clr_pal0 = np.transpose(np.array([np.linspace(68/255, 247/255, clr_cnt),
                                  np.linspace(63/255, 77/255, clr_cnt),
```

```
                                            np.linspace(254/255, 77/255, clr_cnt),
                                            np.ones(clr_cnt)]))
sns_pal = ColorPalette(clr_pal0)
#%%% 4.1 Correlation check
c_method = 1
TimeGet('Correlation maps:', 2)
t0 = time.time()
# whole features
fig_corr_full = CorrelationAnal(mof_x_scal[4], c_method, 1)
fig_corr_full.savefig('correlation full features.tif', dpi=300)
# check tar-specified faetures
for i in range(cnt_tar):
    fig_corr = CorrelationAnal(x_dfs[i], c_method, 1)
    fig_corr.savefig(f'{corr_method[c_method]} correlation {i}.tif', dpi=300)
t1 = time.time()
TimeCalcu(t0, t1)
#%%% 4.2 Target values distribution
TimeGet('Target properties distribution:', 2)
t0 = time.time()
ds_x = ['MOF_Mass', 'MFR_Mass', 'Add_Mass', 'Mat_Modulus', 'Mat_TTI',
'Mat_pHRR', 'Mat_THR']
ds_y = ['Modulus_dc', 'TTI_dc', 'pHRR_dc', 'THR_dc']
# func to draw seaborn pairgrid
def PGdisplay(df, x_feats, y_feats, hue_kw, dia_line=0):
    if hue_kw:
        df_use = df.sort_values(hue_kw) # sort by hue
        g = sns.PairGrid(df_use, y_vars=y_feats, x_vars=x_feats,
                         height=4, aspect=1, hue=hue_kw, palette=sns_pal)
        g.map(sns.scatterplot)
    else:
```

```
            g = sns.PairGrid(df, y_vars=y_feats, x_vars=x_feats,
                                    height=4, aspect=1)
            g.map(sns.scatterplot, color=clr_face[0])
        if dia_line: # add diagonal line in diagonal axes
            axes_cnt = g.axes.shape[0]
            for i in range(axes_cnt):
                ax_dia = g.axes[i,i]
                ax_dia.plot(ax_dia.get_xlim(), ax_dia.get_xlim(), 'k--', alpha=0.6)
        g.add_legend()
        plt.close()
        return g
fig_pg01 = PGdisplay(mof_show, ds_x, ds_y, 'MOF_Mgrp')
fig_pg01.savefig('dataset pairgrid all.tif', dpi=300)
fig_pg02 = PGdisplay(mof_show[mof_show['MOF_Mgrp']!='No_MOF'], ds_x, ds_y,
'MOF_Mgrp')
fig_pg02.savefig('dataset pairgrid added MOF.tif', dpi=300)
ds_y = ['Modulus', 'TTI', 'pHRR', 'THR']
fig_pg03 = PGdisplay(mof_show, ds_x, ds_y, 'MOF_Mgrp')
fig_pg03.savefig('dataset pairgrid all without conversion.tif', dpi=300)
fig_pg04 = PGdisplay(mof_show[mof_show['MOF_Mgrp']!='No_MOF'], ds_x, ds_y,
'MOF_Mgrp')
fig_pg04.savefig('dataset pairgrid added MOF without conversion.tif', dpi=300)
# separate additive-PG and matrix-PG
x_pg = [['MOF_Mass', 'MFR_Mass', 'Add_Mass'], ['Mat_Modulus', 'Mat_TTI',
'Mat_pHRR', 'Mat_THR']]
fig_pg05 = PGdisplay(mof_show[mof_show['MOF_Mgrp']!='No_MOF'], x_pg[0],
ds_y, 'MOF_Mgrp')
fig_pg05.savefig('PG with MOF no conversion ADD part.tif', dpi=300)
fig_pg06 = PGdisplay(mof_show[mof_show['MOF_Mgrp']!='No_MOF'], x_pg[1],
ds_y, 'MOF_Mgrp', 1)
```

```
fig_pg06.savefig('PG with MOF no conversion MAT part.tif', dpi=300)
fig_pg07 = PGdisplay(mof_show[mof_show['MOF_Mgrp']!='No_MOF'], x_pg[0],
ds_y, 'MOF_Class')
fig_pg07.savefig('PG with MOF no conversion ADD part MOF type.tif', dpi=300)
fig_pg08 = PGdisplay(mof_show[mof_show['MOF_Mgrp']!='No_MOF'], x_pg[1],
ds_y, 'MOF_Class', 1)
fig_pg08.savefig('PG with MOF no conversion MAT part MOF type.tif', dpi=300)
# without HUE
x_pg1 = [['MOF_Mass', 'MOF_Mgrp', 'MOF_Class'], ['Mat_Modulus', 'Mat_TTI',
'Mat_pHRR', 'Mat_THR']]
fig_pg09 = PGdisplay(mof_show[mof_show['MOF_Mgrp']!='No_MOF'], x_pg1[0],
ds_y, 0)
fig_pg09.savefig('PG with MOF no conversion ADD part mof.tif', dpi=300)
fig_pg10 = PGdisplay(mof_show[mof_show['MOF_Mgrp']!='No_MOF'], x_pg1[1],
ds_y, 0, 1)
fig_pg10.savefig('PG with MOF no conversion MAT part mof.tif', dpi=300)
# ds_y = ['Modulus_dc', 'TTI_c', 'pHRR_dc', 'THR_dc']
# fig_pg11 = PGdisplay(mof_show[mof_show['MOF_Mgrp']!='No_MOF'], x_pg1[0],
ds_y, 0)
# fig_pg11.savefig('PG dc with MOF no conversion ADD part mof.tif', dpi=300)
# fig_pg12 = PGdisplay(mof_show[mof_show['MOF_Mgrp']!='No_MOF'], x_pg1[1],
ds_y, 0, 1)
# fig_pg12.savefig('PG dc with MOF no conversion MAT part mof.tif', dpi=300)
t1 = time.time()
TimeCalcu(t0, t1)
#%% 5 Estimators and DataFrames
rand_cnt, split_size = 100, 0.15
rand_states = np.arange(rand_cnt)
est_used = pd.DataFrame([['RF', 'classifier', 0, 0],
                         ['RF', 'regressor', 1, 0],
```

```
                                        ['SVM', 'classifier', 0, 1],
                                        ['SVM', 'regressor', 1, 1],
                                        ['MLP', 'classifier', 0, 2],
                                        ['MLP', 'regressor', 1, 2],
                                        ['LightGBM', 'classifier', 0, 3],
                                        ['LightGBM', 'regressor', 1, 3]],
                                       columns=['Name', 'Type', 'Type Mark', 'Name Mark'])
def EstFullName(t, n):
    mask = (est_used['Name Mark']==n) & (est_used['Type Mark']==t)
    res = est_used[mask].iloc[0,0] + ' ' + est_used[mask].iloc[0,1]
    return res
# optimised HyperParameters
# hp = {tar_spec[0]:{'max_depth':10, 'max_features':'sqrt', 'min_samples_split':5,
#                       'n_estimators':1500},
#            tar_spec[1]:{'colsample_bytree':0.8, 'learning_rate':0.01, 'max_depth':4,
'num_leaves':5,
#                            'n_estimators':500, 'reg_alpha':0.1, 'reg_lambda':0.1,
'subsample':0.8},
#             tar_spec[2]:{'C':20.384933982524633, 'gamma':2.324697059985648,
'kernel':'rbf'},
#         tar_spec[3]:{'max_depth':15, 'max_features':'sqrt', 'min_samples_split':5,
#                       'n_estimators':1000},
#            tar_spec[4]:{'colsample_bytree':0.8, 'learning_rate':0.01, 'max_depth':6,
'num_leaves':20,
#                         'n_estimators':500, 'reg_alpha':0.144, 'reg_lambda':0.428,
'subsample':0.8},
#             tar_spec[5]:{'C':106.42092440647247, 'gamma':6.335804992658251,
'kernel':'rbf'},
#         tar_spec[6]:{'max_depth':15, 'max_features':'sqrt', 'min_samples_split':5,
#                       'n_estimators':100},
```

```
#                   tar_spec[7]:{'colsample_bytree':1,  'learning_rate':0.01,  'max_depth':5,
'num_leaves':10,
#                                 'n_estimators':500, 'reg_alpha':0.1, 'reg_lambda':2.63665,
'subsample':0.8},
#             tar_spec[8]:{'C':400.80160320641284, 'gamma':1, 'kernel':'rbf'},
#             tar_spec[9]:{'max_depth':10, 'max_features':'sqrt', 'min_samples_split':5,
#                                'n_estimators':100},
#                tar_spec[10]:{'colsample_bytree':0.8,  'learning_rate':0.01,  'max_depth':5,
'num_leaves':10,
#                                'n_estimators':500, 'reg_alpha':0.1, 'reg_lambda':0.428133,
'subsample':0.8},
#                 tar_spec[11]:{'C':126.50337203959037,  'gamma':3.0724688427090037,
'kernel':'rbf'}}
hp = {}
# func to optimize HP and build models
def EstimateXY(x, y, est_base=0, est_type=0, hp_d0=0, strat=1, smt=0):
    t_est0 = time.time()
    if est_base == 0: # 0 for random forest
        if not est_type:
            if hp_d0:
                hp_optmz = hp_d0
            else:
                hp_optmz, hp_cvresults = est_rfc.rfc_optmzHP(x, y)
            # print('HP for '+tar0+':\n',hp_optmz)
            split, mod, pred, est, fi = est_rfc.rfc_modding(x, y, hp_optmz, rand_cnt,

                                                                              split_size,
strat, smt)
        else:
            if hp_d0:
```

```
                hp_optmz = hp_d0
            else:
                hp_optmz, hp_cvresults = est_rfr.rfr_optmzHP(x, y)
            # print('HP for '+tar0+':\n',hp_optmz)
            split, mod, pred, est, fi = est_rfr.rfr_modding(x, y, hp_optmz, rand_cnt,
split_size)
        t_est1 = time.time()
        time_consum = TimeCalcu(t_est0,t_est1)
        return [split, mod, pred, est, fi, hp_optmz, time_consum, hp_cvresults]
    elif est_base == 1: # 1 for support vector
        if not est_type:
            if hp_d0:
                hp_optmz = hp_d0
            else:
                hp_optmz, hp_cvresults = est_svc.svc_optmzHP(x, y)
            # print('HP for '+tar0+':\n',hp_optmz)
            split, mod, pred, est = est_svc.svc_modding(x, y, hp_optmz, rand_cnt,
                                                            split_size,    strat,
smt)
        else:
            if hp_d0:
                hp_optmz = hp_d0
            else:
                hp_optmz, hp_cvresults = est_svr.svr_optmzHP(x, y)
            # print('HP for '+tar0+':\n',hp_optmz)
            split, mod, pred, est = est_svr.svr_modding(x, y, hp_optmz, rand_cnt,
split_size)
        t_est1 = time.time()
        time_consum = TimeCalcu(t_est0,t_est1)
        return [split, mod, pred, est, hp_optmz, time_consum, hp_cvresults]
```

```
    elif est_base == 2: # 2 for multilayer perceptron
        if not est_type:
            if hp_d0:
                hp_optmz = hp_d0
            else:
                hp_optmz, hp_cvresults = est_mlc.mlc_optmzHP(x, y)
            # print('HP for '+tar0+':\n',hp_optmz)
            split, mod, pred, est = est_mlc.mlc_modding(x, y, hp_optmz, rand_cnt,
                                                                        split_size, strat,
smt)
        else:
            if hp_d0:
                hp_optmz = hp_d0
            else:
                hp_optmz, hp_cvresults = est_mlr.mlr_optmzHP(x, y)
            # print('HP for '+tar0+':\n',hp_optmz)
            split, mod, pred, est = est_mlr.mlr_modding(x, y, hp_optmz, rand_cnt,
split_size)
        t_est1 = time.time()
        time_consum = TimeCalcu(t_est0,t_est1)
        return [split, mod, pred, est, hp_optmz, time_consum, hp_cvresults]
    elif est_base == 3: # 3 for Light GBM
        if not est_type:
            if hp_d0:
                hp_optmz = hp_d0
            else:
                hp_optmz, hp_cvresults = est_lgbc.lgbc_optmzHP(x, y)
            # print('HP for '+tar0+':\n',hp_optmz)
            split, mod, pred, est, fi = est_lgbc.lgbc_modding(x, y, hp_optmz,
rand_cnt,
```

```
split_size, strat, smt)
        else:
            if hp_d0:
                hp_optmz = hp_d0
            else:
                hp_optmz, hp_cvresults = est_lgbr.lgbr_optmzHP(x, y)
            # print('HP for '+tar0+':\n',hp_optmz)
            split, mod, pred, est, fi = est_lgbr.lgbr_modding(x, y, hp_optmz,
rand_cnt, split_size)
        t_est1 = time.time()
        time_consum = TimeCalcu(t_est0,t_est1)
        return [split, mod, pred, est, fi, hp_optmz, time_consum, hp_cvresults]
    # if there is other algorithm (like GradientBoosting), add here
# func to get best splitting from mod_dict
def BestSplit(arr):
    train, test = arr[0], arr[1]
    k = -1
    while test[np.argsort(train)[k]] < 0.75:
        k -= 1
        if k < -30:
            res_rs = np.argsort(train)[-1]
            res_r2 = test[res_rs]
            break
    else:
        res_rs = np.argsort(train)[k]
        res_r2 = test[res_rs]
    return res_rs, res_r2, k
# start modelling
spl_d, hp_d, mod_d, pred_d, est_d, fi_d, tim_con, hp_cv = {}, {}, {}, {}, {}, {}, {}, {}
```

```
mod_things = {}
x_inp, y_out = [], []
est_chosen = [0, 1, 0, 1, 0, 1, 0, 1] # 0 for RF, 1 for SVM, 2 for MLP, 3 for lgbm
est_type = [0 for i in range(cnt_tar)] # 0 for classifier; 1 for regressor
sort_max = [0 for i in range(cnt_tar)]
TimeGet('ML models assessment:')
for i in range(cnt_tar):
    tar = tar_spec[i]
    x = x_dfs[i]
    y = y_dfs[i]
    TimeGet(f'{tar} Modeling start:', 2)
    if not est_type[i]:
        x, y, k_smote = SMOTExy(x, y)
    est_full_name = EstFullName(est_type[i], est_chosen[i])
    # save x and y into x_inp and y_out
    x_inp.append(x)
    y_out.append(y)
    # check if hp optimised already and start estimation
    if tar in list(hp.keys()):
        print(f'        {est_full_name} (FIT only):')
        hp_optim = hp[tar]
        est_results = EstimateXY(x, y, est_chosen[i], est_type[i], hp_optim)
    else:
        print(f'        {est_full_name} (HP + FIT):')
        est_results = EstimateXY(x, y, est_chosen[i], est_type[i], 0, 1, 0)
    if est_chosen[i] in [0, 3]: # 0 for forest-based algorithms
        spl_d[tar] = est_results[0]
        mod_d[tar] = est_results[1]
        pred_d[tar] = est_results[2]
        est_d[tar] = est_results[3]
```

```
            fi_d[tar] = est_results[4]
            hp_d[tar] = est_results[5]
            tim_con[tar] = est_results[6]
            hp_cv[tar] = est_results[7]
        else: # for models without feature importance tool
            spl_d[tar] = est_results[0]
            mod_d[tar] = est_results[1]
            pred_d[tar] = est_results[2]
            est_d[tar] = est_results[3]
            hp_d[tar] = est_results[4]
            tim_con[tar] = est_results[5]
            hp_cv[tar] = est_results[6]
        size_x, size_y = spl_d[tar]['x_train'][0].shape[0], spl_d[tar]['y_train'][0].shape[0]
        size_feats = spl_d[tar]['x_train'][0].shape[1]
        print(f'        Size of dataset:{size_x} X {size_y} with {size_feats} features')
        # find best splitting
        best_rs, best_r2_test, sort_k = BestSplit(mod_d[tar])
        sort_max[i] = best_rs
        mod_things[i] = {'results': f'        Best R2 in {best_rs}({abs(sort_k)}) with
{best_r2_test:.2f}',
                         'hyperparameters': hp_d[tar],
                         'time consumed': tim_con[tar]}
        print(f'        Best R2 in {best_rs}({abs(sort_k)}th) with {best_r2_test:.2f}')
# Build an Anverage-Model from the mathematical avergae of the 3 optimized models
tar_avge = ([f'Modulus({i})' for i in ['RF','SVM', 'AVG']] +
            [f'TTI({i})' for i in ['RF','SVM', 'AVG']] +
            [f'pHRR({i})' for i in ['RF','SVM', 'AVG']] +
            [f'THR({i})' for i in ['RF','SVM', 'AVG']])
# func to get avg predictions and model indices
def AvgPrediction(target):
```

```
        # get average predictions
        avg_train, avg_test = [], []
        for i in range(rand_cnt):
            avg_train.append(((pred_d[f'{target}(RF)']['pred_train'][i] +
                                        pred_d[f'{target}(SVM)']['pred_train'][i])/2).round())
            avg_test.append(((pred_d[f'{target}(RF)']['pred_test'][i] +
                                        pred_d[f'{target}(SVM)']['pred_test'][i])/2).round())
        pred_d[f'{target}(AVG)'] = {'pred_train': avg_train, 'pred_test': avg_test}
        # get average R2, MAE and RMSE
        arr = np.zeros((6,100))
        for j in range(rand_cnt):
            y_0,         y_1         =         spl_d[f'{target}(RF)']['y_train'][j],
spl_d[f'{target}(RF)']['y_test'][j]
            arr[0, j] = r2_score(avg_train[j], y_0)
            arr[1, j] = r2_score(avg_test[j], y_1)
            arr[2, j] = mean_absolute_error(avg_train[j], y_0)
            arr[3, j] = mean_absolute_error(avg_test[j], y_1)
            arr[4, j] = np.sqrt(mean_squared_error(avg_train[j], y_0))
            arr[5, j] = np.sqrt(mean_squared_error(avg_test[j], y_1))
        mod_d[f'{target}(AVG)'] = arr
        convert_r2 = {'Modulus': 2, 'TTI': 5, 'pHRR': 8, 'THR': 11}
        best_r2, _, _ = BestSplit(arr)
        sort_max.insert(convert_r2[target], best_r2)
# start averaging
TimeGet('Construct Average-Models:', 2)
for i in ['Modulus', 'TTI', 'pHRR', 'THR']:
        AvgPrediction(i)
#%% 6 Results Presentation
mpl.rcParams['axes.grid'] = False
# func to convert categories back with levels_dict
```

```
def CateToRange(conv_tar, conv_cate):
    # get the dataframe from levels_dict
    keys = list(levels_dict.keys())
    keys_short = [item[0:3] for item in keys]
    loc_key = keys_short.index(conv_tar[0:3])
    df_cate = levels_dict[keys[loc_key]]
    # convert cates in conv_cate to ranges
    classes = list(df_cate['class'])
    res = []
    for i in conv_cate:
        if i in classes:
            res.append(df_cate.loc[classes.index(i) , 'range'])
        else:
            res.append(i)
    return res
#%%% 6.1 Model indices with Average Model
sort_num = [0,4,8, 1,5,9, 2,6,10, 3,7,11] # re-order the axes
# plot model indices with avg-models
def PlotIndex1(ind_arr, ind, var, fig_name, ylim=[0,0.81], show=1):
    cnt = len(tar_avge)
    f_n, ax_n = MakeFig(cnt)
    for i in range(cnt):
        ax = ax_n[sort_num[i]] # change i to sort_num[i]
        tar = tar_avge[i]
        ax.plot(rand_states, ind_arr[tar][var],
                    c= clr_edge[0], label= ind+' in Trainset')
        ax.plot(rand_states, ind_arr[tar][var+1],
                    c= clr_edge[1], label= ind+' in Testset')
        if ind != 'R2':
            ax.legend(loc=1)
```

```
        else:
            ax.legend(loc=4)
        ax.set_xlabel('Random States')
        ax.set_ylabel(ind)
        ax.set_xlim([0,100])
        ax.set_ylim(ylim)
        ax.text(-0.02,1.03,f'({markers[i]}) {tar_avge[i]}',size=12,
                transform=ax.transAxes,weight='bold')
    # plt.tight_layout()
    if not show:
        plt.close()
    f_n.savefig(fig_name, dpi=300)
    return
# plot R2, MAE and MSE
mae_ylim, mse_ylim = 2, 2
PlotIndex1(mod_d, 'R2', 0, 'R2.tif', [0, 1.01])
PlotIndex1(mod_d, 'MAE', 2, 'MAE.tif', [0, mae_ylim], show=0)
PlotIndex1(mod_d, 'RMSE', 4, 'RMSE.tif', [0, mse_ylim], show=0)
# best models with indices
def BMr2(models, combine=1):
    cnt = len(models)
    if combine:
        fig, axe = MakeFig(cnt*3)
        for i in range(cnt):
            # get R2, MAE and RMSE of this model
            r2_series = [mod_d[models[i]][0], mod_d[models[i]][1]]
            mae_series = [mod_d[models[i]][2], mod_d[models[i]][3]]
            rmse_series = [mod_d[models[i]][4], mod_d[models[i]][5]]
            # R2
            ax0 = axe[i]
```

```
        ax0.plot(rand_states, r2_series[0], c= clr_edge[0], label= 'R2 in Trainset')
        ax0.plot(rand_states, r2_series[1], c= clr_edge[1], label= 'R2 in Testset')
        ax0.legend(loc=4)
        ax0.set_xlabel('Random States')
        ax0.set_ylabel('R2')
        ax0.set_xlim([0,100])
        ax0.set_ylim([0, 1.01])
        ax0.text(-0.02,1.03,f'({markers[i]}) {models[i]}', size=12,
                transform=ax0.transAxes, weight='bold')
        # MAE
        ax1 = axe[i+cnt]
        ax1.plot(rand_states, mae_series[0], c= clr_edge[0], label= 'MAE in Trainset')
        ax1.plot(rand_states, mae_series[1], c= clr_edge[1], label= 'MAE in Testset')
        ax1.legend(loc=1)
        ax1.set_xlabel('Random States')
        ax1.set_ylabel('MAE')
        ax1.set_xlim([0,100])
        ax1.set_ylim([0, 1.5])
        ax1.text(-0.02,1.03,f'({markers[i]}) {models[i]}', size=12,
                 transform=ax1.transAxes, weight='bold')
        # RMSE
        ax2 = axe[i+cnt*2]
        ax2.plot(rand_states, rmse_series[0], c= clr_edge[0], label= 'MAE in Trainset')
        ax2.plot(rand_states, rmse_series[1], c= clr_edge[1], label= 'MAE in Testset')
```

```
            ax2.legend(loc=1)
            ax2.set_xlabel('Random States')
            ax2.set_ylabel('RMSE')
            ax2.set_xlim([0,100])
            ax2.set_ylim([0, 2])
            ax2.text(-0.02,1.03,f'({markers[i]}) {models[i]}', size=12,
                     transform=ax2.transAxes, weight='bold')
        fig.savefig('model indices (R2, MAE, RMSE).tif', dpi=300)
    else:
        fig0, axe0 = MakeFig(cnt)
        fig1, axe1 = MakeFig(cnt)
        fig2, axe2 = MakeFig(cnt)
        for i in range(cnt):
            # get R2, MAE and RMSE of this model
            r2_series = [mod_d[models[i]][0], mod_d[models[i]][1]]
            mae_series = [mod_d[models[i]][2], mod_d[models[i]][3]]
            rmse_series = [mod_d[models[i]][4], mod_d[models[i]][5]]
            # R2
            ax0 = axe0[i]
            ax0.plot(rand_states,   r2_series[0],   c=   clr_edge[0],   label=   'R2   in
Trainset')
            ax0.plot(rand_states,   r2_series[1],   c=   clr_edge[1],   label=   'R2   in
Testset')
            ax0.legend(loc=4)
            ax0.set_xlabel('Random States')
            ax0.set_ylabel('R2')
            ax0.set_xlim([0,100])
            ax0.set_ylim([0, 1.01])
            ax0.text(-0.02,1.03,f'({markers[i]}) {models[i]}', size=12,
                    transform=ax0.transAxes, weight='bold')
```

```
            # MAE
            ax1 = axe1[i]
            ax1.plot(rand_states, mae_series[0], c= clr_edge[0], label= 'MAE in
Trainset')
            ax1.plot(rand_states, mae_series[1], c= clr_edge[1], label= 'MAE in
Testset')
            ax1.legend(loc=1)
            ax1.set_xlabel('Random States')
            ax1.set_ylabel('MAE')
            ax1.set_xlim([0,100])
            ax1.set_ylim([0, 1.5])
            ax1.text(-0.02,1.03,f'({markers[i]}) {models[i]}', size=12,
                     transform=ax1.transAxes, weight='bold')
            # RMSE
            ax2 = axe2[i]
            ax2.plot(rand_states, rmse_series[0], c= clr_edge[0], label= 'MAE in
Trainset')
            ax2.plot(rand_states, rmse_series[1], c= clr_edge[1], label= 'MAE in
Testset')
            ax2.legend(loc=1)
            ax2.set_xlabel('Random States')
            ax2.set_ylabel('RMSE')
            ax2.set_xlim([0,100])
            ax2.set_ylim([0, 2])
            ax2.text(-0.02,1.03,f'({markers[i]}) {models[i]}', size=12,
                     transform=ax2.transAxes, weight='bold')
        fig0.savefig('model indices (R2).tif', dpi=300)
        fig1.savefig('model indices (MAE).tif', dpi=300)
        fig2.savefig('model indices (RMSE).tif', dpi=300)
# get highest and average values for making Table
```

```
# sort_max done in modeling part
cnt_tar = len(tar_avge) # using set with AVG-models
mod_all_ind = np.zeros((cnt_tar,12))
mod_single_ind = np.zeros((cnt_tar,6))
mod_mean_ind = np.zeros((cnt_tar,6))
ind_list = ['R2', 'MAE', 'RMSE']
# save the values in mod_all_ind
for i in range(cnt_tar):
    for j in range(3):
        train_arr = mod_d[tar_avge[i]][j*2]
        test_arr = mod_d[tar_avge[i]][j*2+1]
        # save all indices
        mod_all_ind[i,j*4] = train_arr[sort_max[i]]
        mod_all_ind[i,j*4+1] = np.average(train_arr)
        mod_all_ind[i,j*4+2] = test_arr[sort_max[i]]
        mod_all_ind[i,j*4+3] = np.average(test_arr)
        # save average value of 100 random states
        mod_mean_ind[i,j*2] = np.average(train_arr)
        mod_mean_ind[i,j*2+1] = np.average(test_arr)
        # save single value for best splitting
        mod_single_ind[i,j*2] = train_arr[sort_max[i]]
        mod_single_ind[i,j*2+1] = test_arr[sort_max[i]]
mod_all_ind = mod_all_ind.round(2)
mod_mean_ind = mod_mean_ind.round(2)
mod_single_ind = mod_single_ind.round(2)
# draw a table containing all model indices
tab0_cols = ['train', 'train_avg','test', 'test_avg',
             'train', 'train_avg','test', 'test_avg',
             'train', 'train_avg','test', 'test_avg']
tab1_cols = ['train', 'test', 'train', 'test', 'train', 'test']
```

```
tab_cols = ['R2', 'MAE', 'MSE']
head_clr = ['lightblue']
row_clrs = [cl for cl in head_clr for i in range(len(tar_avge))]
def TableIndex(data, cols, figname, fig_not_show=1):
    col_clrs = [cl for cl in head_clr for i in range(len(cols))]
    fig, ax =plt.subplots(figsize=(12,12))
    ax.axis('off')
    ax.axis('tight')
    tb = ax.table(cellText=data, bbox=(0,0,1,1), loc='center',
                  cellLoc='center', rowLoc='center', colLoc='center',
                  rowLabels=tar_avge, colLabels=cols,
                  rowColours=row_clrs , colColours=col_clrs)
    tb.auto_set_font_size(False)
    h = tb[0,0].get_height()
    w = tb[0,0].get_width()
    # add another line of cells to make R2, MAE and MSE
    header = [tb.add_cell(-1,pos,w,h,loc='center',facecolor='none')\
              for pos in range(data.shape[1])]
    for i in range(data.shape[1]):
        header[i].visible_edges = 'TB'
    # set division between R2, MAE and MSE based on mod shape
    for i in range(3):
        header[0+i*int(data.shape[1]/3)].visible_edges = 'TBL'
        header[int(data.shape[1]/3)*(i+1)-1].visible_edges = 'TBR'
        ind_cell = header[int((data.shape[1]/3)*(i+0.5))]
        ind_cell.set_text_props(size=22, weight='bold', text=tab_cols[i])
    tot_head = [tb.add_cell(-1,-1,w,h), tb.add_cell(0,-1,w,h)]
    tot_head[0].visible_edges = 'TLR'
    tot_head[1].visible_edges = 'BLR'
    tot_head[1].set_text_props(va='bottom', size=22,
```

```
                              weight='bold', text='Indice')
    if fig_not_show:
        plt.close()
    fig.savefig(figname)
TableIndex(mod_all_ind, tab0_cols, 'All indices (single and avg).tif', 0)
TableIndex(mod_mean_ind, tab1_cols, 'Average indice.tif')
TableIndex(mod_single_ind, tab1_cols, 'Single indice.tif')
#%%% 6.2 Feature Importance for RF
# func to seprate and correspond the data of feature importance
def FIcollect(which_tar, fi_lgbm=0):
    model_no = tar_spec.index(which_tar)
    feats = {}
    for i in range(len(feat_x[model_no])): # {feat_name: feat_importance} for single
model
        feats[feat_x[model_no][i]] = fi_d[which_tar][sort_max[model_no], i]
    # return of 'sorted' is [(key1, value1),(key, value)...]
    feats_sorted = sorted(feats.items(), key=lambda item:item[1])
    names = list(np.asarray(feats_sorted)[:,0])
    values   =   np.asarray(pd.Series(np.asarray(feats_sorted)[:,1]).map(lambda
X:float(X)))
    if fi_lgbm:
        values = values / np.sum(values)
    return feats, names, values
# func to show bar width
def BarWidth(bar_a, axe_a):
    for i in bar_a:
        wid = i.get_width()
        if wid < 0.01:
            axe_a.text(wid+0.002, i.get_y() + i.get_height()/2,
                        '< 0.010', ha='left', va='center', fontsize=8)
```

```
            else:
                axe_a.text(wid+0.002, i.get_y() + i.get_height()/2,
                                f'{wid:.3f}', ha='left', va='center', fontsize=8)
# func to classify feature type
feat_grps = ['Polymer_Matrix', 'MOF_Material', 'Main_FR',
                'Other_Parameters']
feat_kind = len(feat_grps)
art_legend = {feat_grps[x]:clr_face[x] for x in range(feat_kind)}
def ClassifyFeat(which_feat, which_tar):
    res_ind = feat_name_dic[which_tar].index(which_feat)
    if which_feat[0:3]=='Mat':
        res_grp = feat_grps[0]
    elif which_feat[0:3]=='MOF':
        res_grp = feat_grps[1]
    elif which_feat[0:3]=='MFR':
        res_grp = feat_grps[2]
    else:
        res_grp = feat_grps[3]
    return res_ind, res_grp
# arrange the data
feat_imp_dic, feat_name_dic, feat_value_dic = {}, {}, {}
num_tree_mod, num_nontree = [], []
cnt_tar = len(tar_spec)
for i in range(cnt_tar):
    if est_chosen[i] in [0, 3]:
        num_tree_mod.append(i)
        tar = tar_spec[i]
        feat_imp_dic[tar], feat_name_dic[tar], feat_value_dic[tar] = FIcollect(tar,
est_chosen[i])
    else:
```

```
            num_nontree.append(i)
    cnt_FImods = len(num_tree_mod)
    # FI in crowds V0
    with mpl.rc_context({'axes.grid': False}):
        f_fi, ax_fi = MakeFig(cnt_FImods)
    fig_bar_plots, axes_list = [], []
    sort_num = [0,1,2,3,5,6,7,8] # re-order the axes from a1-a2-b1-b2 to a1-b1-c1-d1
    for i in range(cnt_FImods):
        axe_i = ax_fi[sort_num[i]] # change i to sort_num[i]
        bar_legend ={}
        tar = tar_spec[num_tree_mod[i]]
        feat_list = feat_name_dic[tar]
        bars = axe_i.barh(range(len(feat_list)), feat_value_dic[tar],
                          color=clr_face[0],height=0.5)
        # color features from different groups in different color
        for j in range(len(feat_list)):
            feat_no, feat_grp = ClassifyFeat(feat_list[j], tar)
            bars[feat_no].set_color(art_legend[feat_grp])
            bars[j].set_edgecolor('dimgrey')
            if feat_grp not in bar_legend.keys():
                bar_legend[feat_grp] = bars[feat_no]
        # sort the bar_lenend dict by feat_grps
        # sorted dict.items returning tuples in list, using dict() to convert back to dict
        # sort order is index order of key (x[0]) in the oder list (feat_grps)
        bar_legend = dict(sorted(bar_legend.items(), key=lambda
x:feat_grps.index(x[0])))
        axe_i.legend(list(bar_legend.values()), list(bar_legend.keys()),
                     loc='lower right')
        axe_i.set_yticks(range(len(feat_list)))
        lab = axe_i.set_yticklabels(feat_list, fontsize=8)
```

```
        max_fi_tick = round(feat_value_dic[tar][-1]*1.2,2)
        axe_i.axis(xmax=max_fi_tick)
        axe_i.text(-0.02,1.03,f'({markers[i]}) {tar}',size=12,
                   transform=axe_i.transAxes,weight='bold')
        # axe_i.text(0.78*max_FI_ticks, 0.1, 'Var ' + str(i), fontsize=24)
        BarWidth(bars, axe_i)
# plt.tight_layout()
f_fi.savefig('feature importance V0.tif', dpi=300)
# sum features in same group into a feature group
# create and sort the array according to xth row in nest_list
def SortListToArray(nest_list, x_th):
    arr = np.array(nest_list)
    # must sorted in axis=1 and return argsort index to array, so T is needed
    res = arr.T[arr.T[:,x_th].argsort()].T
    return res
# calculated feature groups' importance and feature counts
feat_grp_dict, feat_grp_dict1 = {}, {}
for i in num_tree_mod:
    feat_list = feat_name_dic[tar_spec[i]]
    feat_val_list = feat_value_dic[tar_spec[i]]
    feat_sum_res, feat_sum_cnt = [0 for i in range(feat_kind)], [0 for i in range(feat_kind)]
    for j in feat_list:
        feat_no, feat_grp = ClassifyFeat(j,tar_spec[i])
        no_x = feat_grps.index(feat_grp)
        feat_sum_res[no_x] += feat_val_list[feat_no]
        feat_sum_cnt[no_x] += 1
    feat_grp_dict[tar_spec[i]] = feat_sum_res
    # calculate division and create array to sort
    feat_sum_div = [feat_sum_res[i]/feat_sum_cnt[i] for i in range(feat_kind)]
```

```
        fg_arr = SortListToArray([feat_grps,feat_sum_cnt,feat_sum_div], 2)
        feat_grp_dict1[tar_spec[i]] = fg_arr
# func to sort the FI
def SortFeatImport(which_feat, which_dict):
        fg_tar_dict = which_dict[which_feat]
        tar_dict = {feat_grps[i]:fg_tar_dict[i] for i in range(len(feat_grps))}
        tar_sorted_tup = sorted(tar_dict.items(), key=lambda item:item[1])
        tar_sorted_dict = {tar_sorted_tup[i][0]:tar_sorted_tup[i][1]\
                                    for i in range(len(tar_sorted_tup))}
        return tar_sorted_dict
# draw plots v1
with mpl.rc_context({'axes.grid': False}):
        f_fi1, ax_fi1 = MakeFig(cnt_FImods)
for i in range(cnt_FImods):
        axe_i    = ax_fi1[sort_num[i]]
        tar = tar_spec[num_tree_mod[i]]
        feat_grp_FI = SortFeatImport(tar, feat_grp_dict)
        bars = axe_i.barh(range(len(feat_grps)), list(feat_grp_FI.values()),
                                    height=0.5)
        # color features from different groups in different color
        for j in range(len(bars)):
                bars[j].set_color(art_legend[list(feat_grp_FI.keys())[j]])
                bars[j].set_edgecolor('dimgrey')
        axe_i.set_yticks(range(len(list(feat_grp_FI.keys()))))
        yLables_FI = [f'{list(feat_grp_FI.keys())[t]:>20s}' for t in range(feat_kind)]
        axe_i.set_yticklabels(yLables_FI, fontsize=10)
        max_fi_tick = round(max(list(feat_grp_FI.values()))*1.2,2)
        axe_i.axis(xmax=max_fi_tick)
        axe_i.text(-0.05,1.02,f'({markers[i]}) {tar}',
                        size=12,weight='bold',
```

```
                        transform=axe_i.transAxes)
        # axe_i.text(0.78*max_FI_ticks, 0.1, 'Var ' + str(i), fontsize=24)
        BarWidth(bars, axe_i)
# plt.tight_layout()
plt.close()
f_fi1.savefig('feature importance V1.tif', dpi=300)
# draw plots v2
# func to show bar width and feature counts
def BarWidthCount(bar_a, axe_a, cnt_list):
        bar_cnt = 0
        for i in bar_a:
                wid = i.get_width()
                cnt = cnt_list[bar_cnt]
                if wid < 0.01:
                        axe_a.text(wid+0.002, i.get_y() + i.get_height()/2,
                                    f'< 0.010 ({cnt})', ha='left', va='center', fontsize=8)
                else:
                        axe_a.text(wid+0.002, i.get_y() + i.get_height()/2,
                                    f'{wid:.3f} ({cnt})', ha='left', va='center',
                                    fontsize=8)
                bar_cnt += 1
with mpl.rc_context({'axes.grid': False}):
        f_fi2, ax_fi2 = MakeFig(cnt_FImods, 8)
for i in range(cnt_FImods):
        axe_i    = ax_fi2[sort_num[i]]
        tar = tar_spec[num_tree_mod[i]]
        fg_grp_list = feat_grp_dict1[tar][0]
        fg_cnt_list = feat_grp_dict1[tar][1].astype('int')
        fg_div_list = feat_grp_dict1[tar][2].astype('float')
        bars = axe_i.barh(range(len(fg_grp_list)), fg_div_list,
```

```
                              height=0.5)
        # color features from different groups in different color
        for j in range(len(bars)):
                bars[j].set_color(art_legend[fg_grp_list[j]])
                bars[j].set_edgecolor('dimgrey')
        axe_i.set_yticks(range(len(fg_grp_list)))
        axe_i.set_yticklabels(fg_grp_list, fontsize=10)
        max_fi_tick = round(max(fg_div_list)*1.3,2)
        axe_i.axis(xmax=max_fi_tick)
        axe_i.text(-0.05,1.02,f'({markers[i]}) {tar}',
                      size=12,weight='bold',
                      transform=axe_i.transAxes)
        # axe_i.text(0.78*max_FI_ticks, 0.1, 'Var ' + str(i), fontsize=24)
        BarWidthCount(bars, axe_i, fg_cnt_list)
# plt.tight_layout()
plt.close()
f_fi2.savefig('feature importance V2.tif', dpi=300)
# draw plots v3: RF for modulus, LGBM for TTI, pHRR and THR
with mpl.rc_context({'axes.grid': False}):
        f_fi3, ax_fi3 = MakeFig(4)
for i in range(4):
        axe_i    = ax_fi3[i]
        tar = tar_spec[num_tree_mod[i]]
        fg_grp_list = feat_grp_dict1[tar][0]
        fg_cnt_list = feat_grp_dict1[tar][1].astype('int')
        fg_div_list = feat_grp_dict1[tar][2].astype('float')
        bars = axe_i.barh(range(len(fg_grp_list)), fg_div_list,
                              height=0.5)
        # color features from different groups in different color
        for j in range(len(bars)):
```

```
            bars[j].set_color(art_legend[fg_grp_list[j]])
            bars[j].set_edgecolor('dimgrey')
        axe_i.set_yticks(range(len(fg_grp_list)))
        axe_i.set_yticklabels(fg_grp_list, fontsize=10)
        max_fi_tick = round(max(fg_div_list)*1.3,2)
        axe_i.axis(xmax=max_fi_tick)
        axe_i.text(-0.05,1.02,f'({markers[i]}) {tar}',
                      size=12,weight='bold',
                      transform=axe_i.transAxes)
        # axe_i.text(0.78*max_FI_ticks, 0.1, 'Var ' + str(i), fontsize=24)
        BarWidthCount(bars, axe_i, fg_cnt_list)
# plt.tight_layout()
plt.close()
f_fi2.savefig('feature importance V2.tif', dpi=300)
#%%% 6.3 Predictions vs. Measurements
cnt_tar = len(tar_avge)
# plot
sort_num = [0,4,8, 1,5,9, 2,6,10, 3,7,11] # re-order the axes
f_pvm, ax_pvm = MakeFig(cnt_tar)
for i in range(cnt_tar):
        ax1 = ax_pvm[sort_num[i]] # change i to sort_num[i]
        tar = tar_avge[i]
        ax1.scatter(spl_d[tar_avge[int(i/3)*3]]['y_train'][sort_max[i]],
                       pred_d[tar]['pred_train'][sort_max[i]],
                       c=clr_edge[0], alpha = 0.5, label= tar+' in Trainsets')
        ax1.scatter(spl_d[tar_avge[int(i/3)*3]]['y_test'][sort_max[i]],
                       pred_d[tar]['pred_test'][sort_max[i]],
                       c=clr_edge[1], marker = '*', s=50, label= tar+' in Testsets')
        x_min = min(min(spl_d[tar_avge[int(i/3)*3]]['y_test'][sort_max[i]]),
```

```
min(spl_d[tar_avge[int(i/3)*3]]['y_train'][sort_max[i]]),
                        min(pred_d[tar]['pred_train'][sort_max[i]]),
                        min(pred_d[tar]['pred_test'][sort_max[i]]))
    x_max = max(max(spl_d[tar_avge[int(i/3)*3]]['y_test'][sort_max[i]]),

max(spl_d[tar_avge[int(i/3)*3]]['y_train'][sort_max[i]]),
                        max(pred_d[tar]['pred_train'][sort_max[i]]),
                        max(pred_d[tar]['pred_test'][sort_max[i]]))
    axe_ticks = [max(abs(x_min)-2, 0)-0.5, x_max+1]
    ax1.set_xlim(axe_ticks)
    ax1.set_ylim(axe_ticks)
    ax1.set_xlabel('meas. '+tar)
    ax1.set_ylabel('pred. '+tar)
    ax1.set_aspect(1)
    ax1.text(-0.02,1.03,f'({markers[i]}) {tar}',
                transform=ax1.transAxes,size=12,weight='bold')
    ax1.plot(axe_ticks, axe_ticks, c="g", linestyle='--',
                alpha=0.5)
plt.close() # do not show picture
f_pvm.savefig('modelling pvm.tif', dpi=300)
# best models
def BMpvm(models, mod_type=3):
    cnt = len(models)
    fig, axe = MakeFig(cnt)
    for i in range(cnt):
        ax = axe[i] # change i to sort_num[i]
        tar = models[i]
        r2_b = sort_max[tar_avge.index(models[i])] # best model index
        y0 = spl_d[tar_avge[i*mod_type]]['y_train'][r2_b]
        y1 = spl_d[tar_avge[i*mod_type]]['y_test'][r2_b]
```

```
            y2 = pred_d[tar]['pred_train'][r2_b]
            y3 = pred_d[tar]['pred_test'][r2_b]
            ax.scatter(y0, y2, c=clr_edge[0], alpha = 0.5, label= tar+' in Trainsets')
            ax.scatter(y1, y3, c=clr_edge[1], marker = '*', s=50, label= tar+' in Testsets')
            x_min = min(min(y0), min(y1), min(y2), min(y3))
            x_max = max(max(y0), max(y1), max(y2), max(y3))
            axe_lim = [x_min-0.5, x_max+0.5]
            ax.set_xlim(axe_lim)
            ax.set_ylim(axe_lim)
            axe_ticks = np.arange(x_min, x_max+1, 1.0)# set ticks explicity in case lim
!= ticks
            ax.set_xticks(axe_ticks)
            ax.set_yticks(axe_ticks)
            # replace tick labels from numbers to text
            xtick_lab = ax.get_xticks().tolist()
            new_ticks = CateToRange(tar, xtick_lab)
            ax.set_xticklabels(new_ticks, rotation=45) # important! reload the ticks
            ax.set_yticklabels(new_ticks)
            ax.set_xlabel('meas. '+tar)
            ax.set_ylabel('pred. '+tar)
            ax.set_aspect(1)
            ax.text(-0.02,1.03,f'({markers[i]}) {tar}',
                    transform=ax.transAxes, size=12, weight='bold')
            ax.plot(axe_lim, axe_lim, c="g", linestyle='--', alpha=0.5)
        fig.savefig('PvM [best models].tif', dpi=300)
#%%% 6.4 ROC calculation
# func to get confusion matrix for data pair
def ConfusionMat(y_true, y_pred, axe):
    conf_mx = confusion_matrix(y_true, y_pred) # get matrix
    row_sum = conf_mx.sum(axis=1, keepdims=True) # sum along axis=1 with dim
```

```
    row_sum[row_sum==0] = 1 # replace 0 with 1 in row_sum
    norm_conf_mx = conf_mx / row_sum # norm row values with row_sum
    np.fill_diagonal(norm_conf_mx, 0) # fill diagonal with 0
    axe.matshow(norm_conf_mx, cmap=mpl.cm.gray) # visulization
    min_t, max_t = -0.5, row_sum.shape[0] + 0.5
    axe.set_xticks(np.arange(min_t, max_t, 1), minor=True)
    axe.xaxis.grid(False, which='major')
    axe.xaxis.grid(True, which='minor')
    axe.set_yticks(np.arange(min_t, max_t, 1), minor=True)
    axe.yaxis.grid(False, which='major')
    axe.yaxis.grid(True, which='minor')
    return axe
# func to draw ROC according to predicitons (one-vs-rest)
def RocCurve(true, pred, axe, tar):
    lb = preprocessing.LabelBinarizer()
    lb.fit(true)
    y_true = lb.transform(true)
    y_pred = lb.transform(pred)
    fpr, tpr, _ = metrics.roc_curve(y_true.ravel(), y_pred.ravel())
    auc = metrics.roc_auc_score(y_true.ravel(), y_pred.ravel(), average='micro')
    auc *= 100
    axe.plot(fpr, tpr)
    axe.set_aspect('equal')
    axe.set_xlabel('False Posistive Rate')
    axe.set_ylabel('True Posistive Rate')
    axe.text(0.96, 0.12, f'AUC={auc:.2f}%\n for {tar}',
             size=12, transform=axe.transAxes, ha='right', va='center',
             bbox=dict(facecolor='whitesmoke', alpha=0.5))
    return axe
# func to replace elements in list_a from list_b
```

```
def ListReplace(list0, list1, list2): # list0 contains list1
    for i in list1:
        ind0 = list0.index(i)
        ind1 = list1.index(i)
        list0[ind0] = list2[ind1]
# plot for tar_prop
fig_cm, axe_cm = MakeFig(cnt_tar)
fig_roc, axe_roc = MakeFig(cnt_tar)
for i in range(cnt_tar):
    y_real = spl_d[tar_avge[int(i/3)*3]]['y_test'][sort_max[i]]
    y_eval = pd.Series(pred_d[tar_avge[i]]['pred_test'][sort_max[i]])
    # confusion matrix
    ax1 = axe_cm[sort_num[i]] # change i to sort_num[i]
    ax1 = ConfusionMat(y_real, y_eval, ax1)
    ax1.set_xlabel(tar_avge[i])
    # replace tick labels from numbers to ranges
    if tar_avge[i].find('Modul') == 0:
        cat_class = list(levels_dict[tar_feat[1]]['class'])
        cat_list = list(levels_dict[tar_feat[1]]['range'])
    elif tar_avge[i].find('TTI') == 0:
        cat_class = list(levels_dict[tar_feat[2]]['class'])
        cat_list = list(levels_dict[tar_feat[2]]['range'])
    elif tar_avge[i].find('pHRR') == 0:
        cat_class = list(levels_dict[tar_feat[3]]['class'])
        cat_list = list(levels_dict[tar_feat[3]]['range'])
    elif tar_avge[i].find('THR') == 0:
        cat_class = list(levels_dict[tar_feat[4]]['class'])
        cat_list = list(levels_dict[tar_feat[4]]['range'])
    tick_labs = ax1.get_xticks().tolist() # may contain unshown ticks
    ListReplace(tick_labs, cat_class, cat_list)
```

```
        ax1.set_xticklabels(tick_labs, rotation=45) # important! reload the ticks
        ax1.set_yticklabels(tick_labs)
        # ROC curves
        ax2 = axe_roc[sort_num[i]] # change i to sort_num[i]
        ax2 = RocCurve(y_real, y_eval, ax2, tar_avge[i])
plt.close(fig=fig_cm)
fig_cm.savefig('confusion matrix of classidication TARs.tif', dpi=300)
fig_roc.savefig('ROC curves of classidication TARs.tif', dpi=300)
#%% 6 Validation Experiments
# Obtain the predictions
# val_list = [?] # number in dataset, +2 in EXCEL
cnt_val = len(val_list)
val_samples = ['P0', 'P1', 'P2', 'P3','M1', 'M2', 'A1', 'A2']
# get prediction of validation set
y_ori, y_cat = {}, {}
y_cat_arr = {} # for calculating AVG-results
cnt_tar = len(tar_spec)
for i in range(cnt_tar):
        tar = tar_spec[i]
        pred_est = est_d[tar_spec[i]][sort_max[i]]
        y_cat_arr[tar] = pred_est.predict(x_dfs_val[i])
        y_cat[tar] = pd.Series(pred_est.predict(x_dfs_val[i]), index=val_samples)
        ori_y = copy.deepcopy(y_dfs_val[i])
        ori_y.index = val_samples
        y_ori[tar] = ori_y
# add Avg_models
def AvgModelVal(target):
        avg_val = ((y_cat_arr[f'{target}(RF)'] +
                        y_cat_arr[f'{target}(SVM)'])/2).round()
        y_cat[f'{target}(AVG)'] = pd.Series(avg_val, index=val_samples)
```

```
for i in ['Modulus', 'TTI', 'pHRR', 'THR']:
    AvgModelVal(i)
# func to show bar height
def BarHeight(bar_a, axe_a, hei_adj=0.12):
    for i in bar_a:
        hei = i.get_height() - 0.5
        axe_a.text(i.get_x()+i.get_width()/3, hei+hei_adj,
                   f'{hei:.0f}', ha='left', va='center', fontsize=8)
# making barlots of categories (predictions vs. measurements)
cnt_tar = len(tar_avge)
val_range = np.arange(cnt_val)
bar_val, bar_valax = MakeFig(cnt_tar)
sort_num = [0,4,8, 1,5,9, 2,6,10, 3,7,11] # re-order the axes
y_tixks = [(-0.5, 4.5), (-0.5, 7.5), (-0.5, 7.5), (-0.5, 5.5)]
for i in range(cnt_tar):
    # draw the bars
    ax_val = bar_valax[sort_num[i]]
    bars_1 = ax_val.bar(val_range-0.15, y_cat[tar_avge[i]]+0.5,
                        width=0.3, color=clr_face[0], edgecolor='dimgrey',
                        label= 'Predictions', bottom=-0.5)
    bars_2 = ax_val.bar(val_range+0.15, y_dfs_val[int(i/3)*2]+0.5, # y_dfs_val are
the same
                        width=0.3, color=clr_face[1], edgecolor='dimgrey',
                        label= 'Validations', bottom=-0.5)
    if 'Modulus' in tar_avge[i]:
        ax_val.legend(loc=2)
    elif 'TTI' in tar_avge[i]:
        ax_val.legend(loc=2, bbox_to_anchor=(0.18,0.5,0.5,0.5))
    elif 'pHRR' in tar_avge[i]:
        ax_val.legend(loc=1)
```

```
        else:
            ax_val.legend(loc=1, bbox_to_anchor=(0.4,0.5,0.5,0.5))
        ax_val.text(-0.05,1.05,f'({markers[i]}) {tar_avge[i]}',
                    transform=ax_val.transAxes, size=12, weight='bold')
        ax_val.set_xticks(val_range, val_samples)
        ax_val.set_ylim(y_tixks[int(i/3)])
        ax_val.set_xlabel('Validation samples', fontsize=10)
        ax_val.set_ylabel('Target property (Category)', fontsize=10)
        # ax_val.yaxis.set_major_locator(MaxNLocator(integer=True))
        # ax_val.set_ylim(0,bar_ymax[i])
        BarHeight(bars_1, ax_val)
        BarHeight(bars_2, ax_val)
bar_val.savefig('Bar plot pvm Cat.tif')
# making barlots of categories (predictions vs. measurements) with best validation
best_val_model = ['Modulus(RF)', 'TTI(SVM)', 'pHRR(AVG)', 'THR(AVG)']
bar_val1, bar_valax1 = plt.subplots(2, 2, figsize=(12,12))
val_num = [0,5,10,11]
for i in range(4):
        # draw the bars
        ax_val = plt.subplot(2,2,i+1)
        bars_1 = ax_val.bar(val_range-0.15, y_cat[best_val_model[i]]+0.5,
                            width=0.3, color=clr_face[0], edgecolor='dimgrey',
                            label= 'Predictions', bottom=-0.5)
        bars_2 = ax_val.bar(val_range+0.15, y_dfs_val[2*i]+0.5,
                            width=0.3, color=clr_face[1], edgecolor='dimgrey',
                            label= 'Validations', bottom=-0.5)
        if 'Modulus' in best_val_model[i]:
            ax_val.legend(loc=2)
        elif 'TTI' in best_val_model[i]:
            ax_val.legend(loc=2, bbox_to_anchor=(0.18,0.5,0.5,0.5))
```

```
        else:
            ax_val.legend(loc=1)
        ax_val.text(-0.05,1.05,f'({markers[i]}) {best_val_model[i]}',
                    transform=ax_val.transAxes, size=12, weight='bold')
        ax_val.set_xticks(val_range, val_samples)
        ax_val.set_ylim(y_tixks[i])
        # replace tick labels from numbers to ranges
        if best_val_model[i].find('Modul') == 0:
            cat_class = list(levels_dict[tar_feat[1]]['class'])
            cat_list = list(levels_dict[tar_feat[1]]['range'])
        elif best_val_model[i].find('TTI') == 0:
            cat_class = list(levels_dict[tar_feat[2]]['class'])
            cat_list = list(levels_dict[tar_feat[2]]['range'])
        elif best_val_model[i].find('pHRR') == 0:
            cat_class = list(levels_dict[tar_feat[3]]['class'])
            cat_list = list(levels_dict[tar_feat[3]]['range'])
        elif best_val_model[i].find('THR') == 0:
            cat_class = list(levels_dict[tar_feat[4]]['class'])
            cat_list = list(levels_dict[tar_feat[4]]['range'])
        tick_labs = ax_val.get_yticks().tolist() # may contain unshown ticks
        ListReplace(tick_labs, cat_class, cat_list)
        ax_val.set_yticklabels(tick_labs)
        # ax_val.yaxis.set_major_locator(MaxNLocator(integer=True))
        # ax_val.set_ylim(0,bar_ymax[i])
        BarHeight(bars_1, ax_val)
        BarHeight(bars_2, ax_val)
    bar_val1.text(0.5, 0.06, 'Validation samples', fontsize=16, ha='center')
    bar_val1.text(0.035, 0.5, 'Target property (Category)', fontsize=16, rotation=90,
    va='center')
    bar_val1.savefig('Bar plot pvm Cat selected.tif')
```

```
#%% 7 Best Models
# best models
# [models] of chosen models with best performance in target_feature(MOD)
# mod_type is the number of algorithms (here: RF, SVM, LGBM, AVG)
def BMroc(models, mod_type=3, combine=1):
    cnt = len(models)
    if combine:
        fig0, axe0 = MakeFig(2*cnt)
        for i in range(cnt):
            r2_b = sort_max[tar_avge.index(models[i])] # splitting index
            y_real = spl_d[tar_avge[i*mod_type]]['y_test'][r2_b]
            y_eval = pd.Series(pred_d[models[i]]['pred_test'][r2_b])
            # confusion matrix
            ax0 = axe0[i]
            ax0 = ConfusionMat(y_real, y_eval, ax0)
            # ax0.set_xlabel(models[i])
            # replace tick labels from numbers to text
            tlabs = ax0.get_xticks().tolist()
            newt = CateToRange(models[i], tlabs)
            ax0.set_xticklabels(newt, rotation=45) # important! reload the ticks
            ax0.set_yticklabels(newt)
            # ROC curves
            ax2 = axe0[i+cnt]
            ax2 = RocCurve(y_real, y_eval, ax2, models[i])
            # add marker
            ax0.text(-0.08,-0.15,f'({markers[i]}) {models[i]}',
                      transform=ax0.transAxes, size=20, weight='bold')
        fig0.savefig('confusion matrix & ROC curves [best models].tif', dpi=300)
    else:
        fig0, axe0 = MakeFig(cnt)
```

```
        fig1, axe1 = MakeFig(cnt)
        for i in range(cnt):
            r2_b = sort_max[tar_avge.index(models[i])] # splitting index
            y_real = spl_d[tar_avge[i*mod_type]]['y_test'][r2_b]
            y_eval = pd.Series(pred_d[models[i]]['pred_test'][r2_b])
            # confusion matrix
            ax0 = axe0[i]
            ax0 = ConfusionMat(y_real, y_eval, ax0)
            ax0.set_xlabel(models[i])
            # replace tick labels from numbers to text
            tick_labs = ax2.get_xticks().tolist()
            new_ticklabs = CateToRange(models[i], tick_labs)
            ax2.set_xticklabels(new_ticklabs, rotation=45) # important! reload the
ticks
            ax2.set_yticklabels(new_ticklabs)
            # ROC curves
            ax2 = axe1[i]
            ax2 = RocCurve(y_real, y_eval, ax2, models[i])
            # add marker
            ax0.text(-0.08,-0.15,f'({markers[i]})',
                     transform=ax0.transAxes, size=24, weight='bold')
        plt.close(fig=fig0)
        fig0.savefig('confusion matrix [best models].tif', dpi=300)
        fig1.savefig('ROC curves [best models].tif', dpi=300)

# best_val_model = ['Modulus(RF)', 'TTI(RF)', 'pHRR(AVG)', 'THR(SVM)']
BMr2(best_val_model) # model indices
BMpvm(best_val_model) # PvM
BMroc(best_val_model) # confusion matrix and ROC curves
#%% 8 SHAP analysis
```

```
TimeGet('SHAP analysis:', 2)
t0 = time.time()
def ShapValue(t, k): # t for which model, k for bg size
    if k<1: # use partial dataframe
        x0, x1, y0, y1 = train_test_split(x_inp[t], y_out[t], test_size=k, random_state= 2)
    else:
        x1 = x_inp[t] # use whole dataframe
    if est_chosen[t] in [0, 3]: # use est_chosen to judge model type
        explainer = shap.TreeExplainer(est_d[tar_spec[t]][sort_max[t]])
    else:
        explainer = shap.KernelExplainer(est_d[tar_spec[t]][sort_max[t]].predict, x1)
    SHAPs = explainer.shap_values(x1)
    EXPLAINs = explainer(x1)
    return x1, EXPLAINs, SHAPs
# plot in summary_plot
cnt_tar = len(tar_spec)
shap_bg, shap_exp, shap_val = {}, {}, {}
for i in range(cnt_tar):
    shap_bg[tar_spec[i]], shap_exp[tar_spec[i]], shap_val[tar_spec[i]] = ShapValue(i, 0.9)
    # summary_plot
    fig_shap_sum = plt.figure()
    values = shap_val[tar_spec[i]]
    arrays = shap_bg[tar_spec[i]]
    shap.summary_plot(values, arrays, max_display=arrays.shape[1])
    plt.close()
    fig_shap_sum.savefig(f'shap values in summary of {tar_spec[i]}.tif', dpi=300)
t1 = time.time()
```

```
TimeCalcu(t0, t1)
# summary_plot of SVM models with max_display=12
svm_tar = [1, 3, 5, 7]
for i in range(len(svm_tar)):
    fig_shap_sum1 = plt.figure()
    values = shap_val[tar_spec[svm_tar[i]]]
    arrays = shap_bg[tar_spec[svm_tar[i]]]
    shap.summary_plot(values, arrays, max_display=12)
    plt.xlabel(f'SHAP analysis of Model {tar_spec[svm_tar[i]]}')
    plt.close()
    fig_shap_sum1.savefig(f'SVM SHAP of {tar_spec[svm_tar[i]]}.tif', dpi=300)
#%% 9 save results
data_sets = [mof_df_enc, x_inp, y_out, tar_spec, tar_avge]
data_processing = [enc_feats, enc_dict, scalers, levels_dict]
model_related = [mod_d, pred_d, est_d, fi_d, hp_d, mod_things, sort_max]
model_explanation = [shap_bg, shap_exp, shap_val]
with open('C:\\LocalML\\Val-20240703 MOF-all.pkl',
          'wb') as f:
    pickle.dump(data_sets,f)
    pickle.dump(data_processing,f)
    pickle.dump(model_related,f)
    pickle.dump(model_explanation,f)
```