Supplementary Information

A Survey of Basic Concepts and Applications of Machine Learning to Chemistry

Julio Cesar Duarte, [a,b] ***Antonio G. S. de Oliveira-Filho,*** [c] ***Matheus Máximo-Canadas,*** [d] ***Rubens C. Souza*** [b] ***and Itamar Borges Jr.*** [*,c,d]

[a]*Departamento de Engenharia da Computação, Instituto Militar de Engenharia (IME), Praça General Tibúrcio, 80, 22290-270 Rio de Janeiro-RJ, Brazil*

[b]*Departamento de Engenharia de Defesa, Instituto Militar de Engenharia (IME), Praça Gen. Tibúrcio 80, 22290-270 Rio de Janeiro-RJ, Brazil*

[c]*Instituto de Química de São Carlos, Universidade de São Paulo, Av. Trabalhador São-Carlense, 400, 13566-090 São Carlos-SP, Brazil*

[d]*Departamento de Química, Instituto Militar de Engenharia (IME), Praça General Tibúrcio, 80, 22290-270 Rio de Janeiro-RJ, Brazil*

1. The machine learning approach and different algorithms

Before applying machine learning (ML) techniques, it is essential to organize the data to ensure robust training and proper evaluation. In a regression task, the dataset is typically divided into three subsets: the training set, which is used to build the model by learning patterns from the data; the validation set, which helps fine-tune hyperparameters and the test set, which evaluates the final model's performance on unseen data, ensuring its generalization capability. A common practice is to allocate approximately 70% of the data for training, while the remaining 30% is split equally between validation and testing. Once trained with the training set and its weights adjusted with the validation set, the model can make variations for new inputs, and its performance is evaluated using analyses such as root mean squared error (RMSE) or mean absolute error (MAE) on the test set; to learn more about these metrics, see the next section. In Python, this process can be implemented using the Scikit-learn package,[1] where the *train_test_split* function is used for data partitioning.

In the following sub-sections, we describe briefly the ML models listed in Table 1 of the review.

*e-mail: itamar@ime.eb.br

1.1. Bayes[2]

Naïve Bayes is a family of probabilistic machine learning algorithms based on Bayes' theorem, which defines the probability of an event given prior knowledge of conditions related to that event. The algorithm is termed "naïve" because it assumes that the features used for classification are conditionally independent, meaning that the presence of one feature does not affect the presence of another. Despite this strong assumption, Naïve Bayes often performs remarkably well in practice, especially for text classification, spam filtering, and sentiment analysis.[3] The core principle involves calculating the posterior probability of a class based on the input features using the formula:

$$\mathrm{P(C \mid X)} = \frac{\mathrm{P(X|C)P(C)}}{\mathrm{P(X)}} \qquad \text{(S1)}$$

where $P(C \mid X)$ is the probability of class C given the feature set X, $P(X \mid C)$ is the likelihood of observing X given class C, $P(C)$ is the prior probability of class C, and $P(X)$ is the overall probability of X.

The Scikit-learn package offers examples of how to apply this technique. However, it does not display a specific input explainability tool,[1] so it is advisable to use the SHAP technique[4] to evaluate the importance of your input parameters.

1.2. Logistic regression[5]

Logistic regression is a statistical and machine learning algorithm used for binary classification tasks, although it can also be extended to multiclass problems. Despite its name, logistic regression is a classification algorithm, not a regression technique. It models the probability that an instance belongs to a particular class using the logistic (sigmoid) function, which maps any real-valued input to a range between 0 and 1.[6] The following equation defines the decision boundary:

$$\mathrm{P(Y = 1 \mid X)} = \frac{1}{1 + \mathrm{e}^{-(\beta 0 + \beta 1 X1 + \beta 2 X2 + \cdots + \beta n Xn)}} \qquad \text{(S2)}$$

where $P(Y = 1 \mid X)$ represents the probability of the positive class given input features X, and $\beta 0, \beta 1, \ldots, \beta n$ are the model coefficients (weights) learned during training.

The training process involves optimizing these coefficients to maximize the likelihood function, which measures how well the model predicts the observed data. The model outputs a probability score, and a classification decision is made by applying a threshold (commonly 0.5). If the probability is greater than or equal to 0.5, the instance is assigned to the positive class; otherwise, it is assigned to the negative class.

1.3. Linear regression[7]

Linear regression is a supervised learning algorithm used to model the relationship between a dependent variable and one or more independent variables by fitting a linear equation to the observed data. It assumes that the dependent variable changes proportionally with the independent variables and aims to find the best-fitting line that minimizes the difference between predicted and actual values.[8]

The mathematical representation of a linear regression model is:

$$Y = \beta_0 + \beta_1 X_1 + \beta_1 X_2 + \ldots + \beta_n X_n + \epsilon \qquad (S3)$$

where Y is the predicted value of the dependent variable, $X_1, X_2, \ldots, X_n$ are the independent variables (features), β_0 is the intercept, representing the expected value of Y when all X values are zero, $\beta_1, \beta_2, \ldots, \beta_n$ are the regression coefficients, indicating the contribution of each independent variable to Y and, ϵ is the error term, accounting for variations not explained by the model.[8]

The Scikit-learn[3] package shows examples of how to apply this technique. The importance of each input (independent variable) is represented by its coefficient (β_i). Evaluating these coefficients can help you understand how each input feature of the model affects the result or explains why the model delivers the result it does.

1.4. Meanshift[9]

The Mean Shift algorithm is an iterative procedure that shifts each data point towards the mean of the data points in its neighborhood. The difference between the sample mean and the current point is referred to as the mean shift. The algorithm generalizes this iterative process by repeatedly moving data points towards the sample means.[10] It operates as follows. First, a finite set S of data points in an n-dimensional Euclidean space X is defined. A kernel K, such as the Gaussian kernel, is selected, which is a non-negative and non-increasing function. For each data point x in X, the sample mean $m(x)$ is computed using the formula:

$$m(x) = \frac{\sum_{s \in S} K(s - x)s}{\sum_{s \in S} K(s - x)} \qquad (S4)$$

where $K(s - x)$ is the kernel evaluated at the difference between each data point s in S and x. The data point x is then shifted to the sample mean $m(x)$, with the magnitude of the mean shift proportional to the ratio of the gradient to the local density estimate using the kernel K. This process is repeated until convergence, which occurs when the mean shift becomes smaller than a specified threshold or a maximum number of iterations is reached. Mean Shift is commonly used in image segmentation, object tracking, and anomaly detection. It is implemented in Scikit-Learn[1] as MeanShift. For direct explainability, one can analyze the cluster centers found by the algorithm (.cluster_centers_) to interpret the locations of high-density regions in the data.

1.5. Artificial Neural Networks[11]

Artificial neural networks (ANNs) are computational models inspired by the structure and functioning of the human brain, designed to recognize complex patterns and perform tasks such as classification, regression, and prediction. Based on layers of interconnected neurons, these networks are capable of learning and generalizing from data, adjusting their internal parameters through iterative training processes.[12]

Since neural networks are based on human brains, the first idea to be presented is the simplified model of a neuron, also known as a unit or node. The signals transmitted by other neurons are received as inputs (also called synapses) by the neuron being processed.[13] This processing involves performing a linear combination of these inputs. This output can be described as

$$f\left(\sum_{i=1}^{n} x_i . w_i - \mu\right) \qquad (S5)$$

where x_i is the value of each input, w_i is the weight related to each input, and μ is the activation threshold.[13]

ANNs are widely used in image recognition, natural language processing, and scientific modeling. In Python, it can be implemented using TensorFlow/Keras (*tf.keras.Sequential*) or PyTorch[14] (*torch.nn*). For direct explainability, one can analyze weight magnitudes, feature importance via input gradients, or activation maps to understand how the network processes information.

1.6. Support Vector Machines[15]

Support Vector Machines (SVM) are powerful supervised learning models utilized for classification and regression tasks. The central concept is to identify the optimal hyperplane that most effectively separates data points belonging to different classes while maximizing the margin, which is the distance between the hyperplane and the nearest data points (support vectors). In situations where the data is not linearly separable, SVM applies the kernel trick to transform the data into a higher-dimensional space, making it easier to locate a separating hyperplane. Common kernel functions include linear, polynomial, radial basis function (RBF), and sigmoid.[16]

In more detailed terms, given a linearly separable training dataset $S = \{(x_1, y_1), \dots, (x_l, y_l)\} \subseteq (X \times Y)^l$, where $X \subseteq \mathbb{R}^n$ denotes the input space, $Y = \{-1, +1\}$ the output domain for binary classification. Support vector classifiers are based on the class of hyperplanes $\langle w, x \rangle + b = 0$ with $w, x \in \mathbb{R}^n, b \in R$, corresponding to the decision functions $f(x) = sign(\langle w, x \rangle + b)$. The algorithm searches for the separating hyperplane with the largest margin while minimizing the norm of w. This can be formulated by the following optimization problem:

$$\begin{aligned} &min \ < w, w > \\ &s.t. \ y_i(< x_i, w > + b) - 1 \geq 0 \end{aligned} \qquad (S6)$$

In Python, SVM is available in Scikit-learn[1] via SVC for classification and SVR for regression. For direct explainability, support vectors (*.support_vectors_*), coefficients (*.coef_* for linear SVM), and decision function values *(.decision_function( ))* can provide insights into which features and data points influence the model's decisions.

1.7. Decision Table[17]

A Decision Table is a structured way of representing complex decision-making logic in a tabular format. It is commonly used in rule-based systems, business processes, and expert systems to model conditional logic clearly and systematically. A decision table consists of four main components: conditions, condition values, actions, and action outcomes. Each row represents a rule that maps a specific combination of conditions to an action.

Mathematically, a decision table can be expressed as a function f(x), where a set of conditions $C_1, C_2, \dots, C_n$ determines the outcome:

$$f(C_1, C_2, \dots, C_n) = A_m \qquad (S7)$$

where A_m is the corresponding action based on the given conditions. Decision tables are particularly useful in situations with multiple interdependent rules, as they offer a structured approach to avoid redundancy and ensure consistency.

In Python, decision tables can be implemented using pandas DataFrames, if-elif conditions, or rule-based systems like *DecisionTreeClassifier* from Scikit-learn. For direct explainability, the table itself serves as an interpretable model, where each rule can be explicitly analyzed to understand how decisions are made based on the input conditions.

1.8. Alternating Decision Tree[18]

An Alternating Decision Tree (ADTree) is a machine learning model that combines the interpretability of decision trees with the predictive power of boosting. Unlike traditional decision trees, which consist only of decision nodes (splitting points) and leaf nodes (final decisions), an ADTree alternates between split nodes (decision rules) and prediction (boosting) nodes that accumulate confidence scores. Instead of assigning a single class label at the leaves, ADTrees provide a sum of scores along the path from the root to the instance, which is then used for classification.[19]

Mathematically, given an instance x, the final classification score is computed as the sum of prediction node values encountered along the decision path:

$$\mathrm{F(x)} = \sum_{\mathrm{i}} \mathrm{h_i(x)} \qquad \text{(S8)}$$

where $h_i(x)$ represents the value of each boosting node. The sign of $F(x)$ determines the predicted class: positive scores are classified into one category, and negative scores are classified into the other.

In Python, ADTrees can be implemented using libraries like Weka[20] (Java-based) or manually constructed using boosting frameworks. One can analyze decision paths, node scores, and accumulated confidence values for direct explainability to understand how different conditions contribute to the final prediction.

1.9. Logistic Model Tree[21]

A Logistic Model Tree (LMT) is a hybrid machine learning model that combines decision trees with logistic regression to improve predictive accuracy while maintaining interpretability. Unlike traditional decision trees, which assign class labels at the leaf nodes, LMTs fit logistic regression models at the leaves, allowing for smoother decision boundaries. This makes LMTs particularly effective for datasets with both categorical and continuous features where linear separability may vary across different regions of the feature space.[21]

The regression function considers a subset of the attributes present in the data and models the probabilities of class membership. The general formula for calculating the probabilities is:

$$\Pr(\mathrm{G = j \mid X = x}) = \frac{e^{\mathrm{F_j(x)}}}{\sum_{\mathrm{k=1}}^{\mathrm{j}} e^{\mathrm{F_k(x)}}} \qquad \text{(S9)}$$

where

$$F_j(x) = \alpha_0^j + \sum_{v \epsilon V_t}^{m} \alpha_v^i \, . v \qquad (S10)$$

and $Pr(G = j \mid X = x)$ Represents the probability of an instance X belonging to class j, given the attribute vector x; G is the class variable. X denotes the space of all features present in the data; x is the vector of attribute values for the instance, a constant component is assumed in the input vector to accommodate the intercept. For the function $F_j(x)$ that combines the attributes for class j, we have that V_t is the subset of attributes considered in the tree node and J is the total number of classes.[13]

In Python, LMTs are not natively available in Scikit-learn, but they can be implemented using Weka[12] (Java-based) or by combining *DecisionTreeClassifier* with *LogisticRegression* in a custom pipeline. For direct explainability, LMTs allow interpretation at two levels: the tree structure, which explains global decision rules, and the logistic regression coefficients at the leaves, which describe local feature importance within each sub-region of the data.

1.10. Random Trees

Random Trees can be understood in a mathematical sense through the framework of Galton-Watson trees, which describe branching processes in probability theory. These trees model the evolution of populations where each individual (or node) produces a random number of offspring according to a specified offspring distribution μ. Formally, a random tree can be defined using a collection of independent random variables $(K_u, u \in U)$, where K_u follows the law μ and is indexed by the label set U.[22] The set of nodes forming the tree is then given by

$$\theta = \{u = u^1 \dots u^n \in U : u^i \leq K_{u^1 \dots u^{j-1}} \text{ for every } 1 \leq j \leq n\} \qquad (S11)$$

Random Trees are commonly used in ensemble methods like Random Forests, where multiple trees are trained independently, and predictions are aggregated via majority voting (classification) or averaging (regression).[22]

In Python, Random Trees can be implemented using *RandomForestClassifier* and *RandomForestRegressor* from the Scikit-Learn library. If a single Random Tree is required, *ExtraTreeClassifier* and *ExtraTreeRegressor* allow for extremely randomized trees, where both feature selection and split values are randomly determined. For interpretability, the *.feature_importances_* attribute provides insights into how different features influence predictions.

1.11. Reduced Error Pruning Tree[23]

Reduced Error Pruning (REP) Tree is a post-pruning technique used to simplify decision trees and improve their generalization by reducing overfitting. The method works by evaluating each subtree and determining whether removing it improves classification accuracy on a separate validation set. If pruning a node does not decrease or improve accuracy, a leaf node replaces the subtree, assigning it the most frequent class in the pruned region.

The pruning process begins with a fully grown decision tree trained on a dataset. Then, using a validation set, the algorithm iteratively examines each internal node, replacing it with a leaf if the pruning does not negatively impact accuracy. This reduces tree complexity and enhances interpretability while maintaining performance. Since REP is a

greedy algorithm, it does not guarantee a globally optimal pruned tree but provides a good trade-off between complexity and accuracy.[24]

In Python, Reduced Error Pruning is not directly available in Scikit-learn, but decision trees built with *DecisionTreeClassifier* and *DecisionTreeRegressor* can be pruned by setting parameters like *max_depth* or *min_samples_leaf*. Additionally, custom implementations of REP can be created by evaluating nodes on a validation set and pruning based on error reduction.

1.12. Decision Tree Regressor[25]

A Decision Tree Regressor is a supervised learning algorithm used for regression tasks, where the goal is to predict continuous numerical values. It works by recursively splitting the dataset into smaller subsets based on feature values, minimizing a chosen error metric (such as mean squared error (MSE)) at each step. The final model consists of a tree structure where the leaves represent predicted values, computed as the average of training samples within that leaf, like in Figure S1.[26]

\

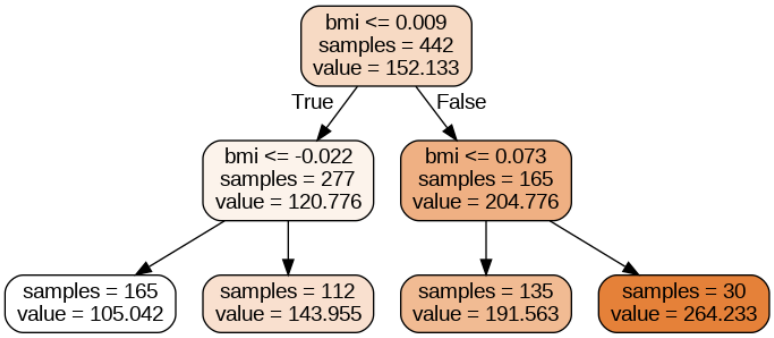

Figure S1. Decision Tree for regression.

In Python, a Decision Tree Regressor can be implemented using Scikit-learn (*sklearn.tree.DecisionTreeRegressor*). After training, the *.feature_importances_* attribute provides insights into how different features influence predictions, and hyperparameters like *max_depth*, *min_samples_split*, can be tuned to control overfitting and improve generalization.

1.13. Gradient Boosting[27]

Gradient Boosting is a powerful machine learning method that incrementally builds a predictive model by combining multiple simpler models, often called “base learners”. The main idea behind Gradient Boosting is to add new models to the mix sequentially. Each new model is trained to correct the errors made by the existing models.[28]

The Gradient Boosting algorithm works by initializing a simple model as an average of the training data. For each iteration t from 1 to M (total number of models to be built) compute the negative gradient of the loss function with respect to the current model output. The negative gradient represents the direction in which the model needs to be adjusted to reduce the loss and fit a new base learner to the negative gradient values. The goal is to find a model that is highly correlated with the negative gradient.[28] Compute the optimal step size (ρt) for the new base learner. This can be done through numerical optimization. Finally, Update the model by adding the new base learner multiplied by the step size:

$$\hat{\hat{f}}_t \leftarrow \hat{f}_{t-1} + \lambda \rho_t h(x, \theta_t) \qquad (S12)$$

where $\hat{f}_t$ is the estimate of the function at step t, $\hat{\hat{f}}_{t-1}$ is the estimate of the function at the previous step, λ is the learning rate (shrinkage), ρ_t is the step size, and $h(x, \theta_t)$ is the new base learner. The final model is the weighted sum of all base learners:

$$\hat{f}(x) = \hat{f}^M(x) = \sum_{i=0}^{M} \hat{f}_i(x) \qquad (S13)$$

Gradient Boosting is highly flexible and can be used for both regression and classification problems, by choosing different loss functions and can be implemented with Scikit-learn, which offer optimized performance and better categorical data handling. The direct explainability is available via *.feature_importances_*, ranking each feature's contribution that helps understand model decisions.

1.14. AdaBoost[29]

AdaBoost (Adaptive Boosting) is an ensemble learning method that combines multiple weak classifiers, typically decision trees with a single split (stumps), to create a strong predictive model. It works iteratively by adjusting the weights of training samples, focusing more on misclassified instances in each step. Initially, all samples have equal weights, but after each iteration, misclassified samples receive higher weights, making the next weak learner focus more on these complex cases. The final model is a weighted sum of all weak learners, where the weight of each learner depends on its accuracy.[30]

Mathematically, given a weak learner $h_m(x)$, its weight α_m is computed as:

$$a_m = \frac{1}{2} \ln\left(\frac{1 - e_m}{e_m}\right) \qquad (S14)$$

where e_m is the error rate of the weak learner.[31] This model is widely used for both classification and regression tasks and is known for improving weak learners while being less prone to overfitting. In Python, it can be implemented using Scikit-learn[1] via *AdaBoostClassifier* and *AdaBoostRegressor*. For direct explainability, the .feature_importances_ attribute provides insights into feature importance, showing which features contribute most to the model's decisions.

1.15. Bagging[32]

Bagging (Bootstrap Aggregating) is a computationally intensive method to reduce the variance of unstable predictors, particularly useful for high-dimensional datasets. It involves creating multiple bootstrap datasets, each generated by random sampling with replacement from the original training set. Each of these subsets is used to train an independent predictive model, and the final prediction is obtained by averaging the predictions of individual models.[33] The bagged predictor is defined as:

$$\hat{\theta}_{n;B}(x) = E^*\left[\hat{\theta}_n^*(x)\right] \qquad (S15)$$

where E^*denotes the bootstrap expectation, and $\hat{\theta}_n^*(x)$ represents the predictor obtained from a bootstrap dataset, computed as $\hat{\theta}_n^*(x) = h_n(L_1^*, \ldots, L_n^*)(x)$, where $L_i^* = (Y_i^*, X_i^*)$ is a sample drawn from the empirical distribution of the pairs $L_i = (Y_i, X_i)$.

In Python, Bagging can be implemented using *BaggingClassifier* and *BaggingRegressor* from the scikit-learn library. For direct explainability, one can analyze feature importances across base models, inspect individual trees (*.estimators_*), and compute sample influence by tracking which data points are frequently used across different bootstrap samples.

1.16. Random Subspace[34]

The Random Subspace Method is an approach for constructing a decision forest by pseudo-randomly selecting subsets of the feature vector to build multiple decision trees. Given a feature space of n dimensions, there are 2^n possible feature selections, each of which can be used to construct a distinct decision tree. At each iteration, a subset of features is chosen, and a subspace is formed where all unselected dimensions remain fixed. The training samples are then projected into this subspace, and a decision tree is built using only the selected features.[34]

For classification, an unknown sample is projected into the same subspace and classified using the corresponding tree. The final classification is obtained by averaging the conditional probabilities of each class at the terminal nodes across all trees. Let $v_j(x)$ be the terminal node in which a sample x falls, and let $P(c|v_j(x))$ represent the probability of x belonging to class c. This probability is estimated as the fraction of training samples in class c within $v_j(x)$. The discriminant function is given by:

$$g_c(x) = \frac{1}{n_t} \sum_{j=1}^{n_t} P\,(c|v_j(x)) \qquad (S16)$$

where n_t is the number of trees. The decision rule assigns x to the class c that maximizes $g_c(x)$. This method enhances diversity among trees and is particularly effective in high-dimensional spaces. In Python, the Random Subspace Method can be implemented using Scikit-learn's *BaggingClassifier* with the *max_features* parameter or through *RandomForestClassifier*, which inherently applies feature subspace selection.[1] For explainability, analyzing feature importances across different models and checking individual subspace selections (*.estimators_* in ensemble models) can provide insights into how different features contribute to predictions.

1.17. Random Forest

Random forests are an ensemble learning method that enhances the performance and stability of decision trees by constructing multiple trees and aggregating their predictions. The algorithm operates through a process known as bootstrap aggregating (bagging), where multiple subsets of the training data are generated by random sampling with replacement. Each individual decision tree is trained on a different subset, introducing variability that diminishes overfitting. Additionally, at each node split within a tree, only a random subset of features is considered, further promoting diversity among trees. In classification tasks, the final prediction is made by majority voting across all trees, while in regression, the outputs are averaged. This method is particularly effective in handling high-dimensional data, reducing

the impact of noise, and providing robust predictions. Moreover, random forests inherently estimate feature importance, allowing for interpretability in various machine learning applications. [35] Figure S2 shows an example of a Random Forest.

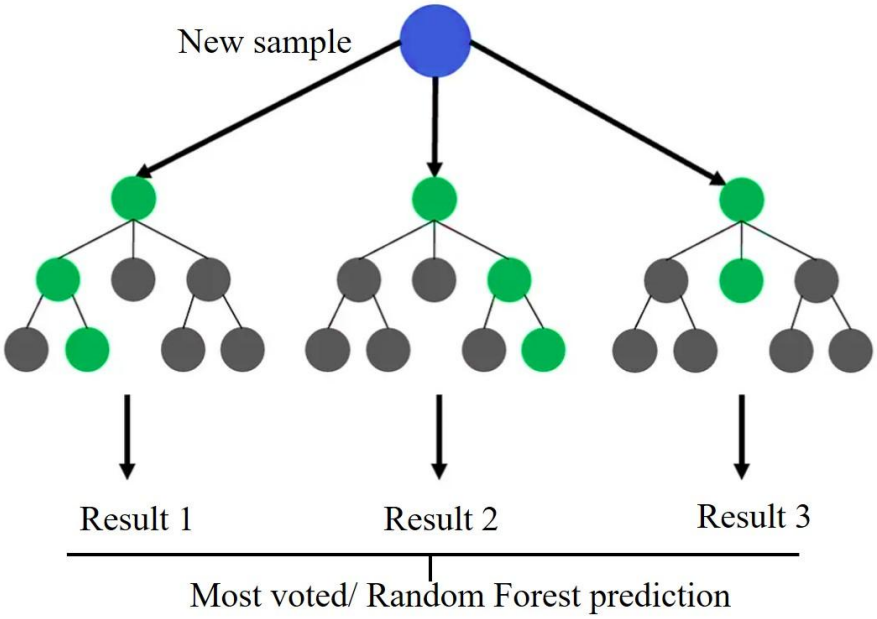

Figure S2. Random Forest.

Furthermore, the Scikit-Learn package offers examples of how to apply this technique. This package also demonstrates a method for searching for explainability in the parameters used for model input; the tool is called features_importances.

1.18. LASSO[36]

The Lasso (Least Absolute Shrinkage and Selection Operator) method is a regression technique that performs shrinkage and variable selection simultaneously. It minimizes the sum of squares of the residuals, subject to a restriction on the absolute value of the sum of the regression coefficients. The problem formulation is given by:

$$\text{argmin}\left\{\sum_{i=1}^{N}\left(y_i-\alpha-\sum_j \beta_j x_{ij}\right)^2\right\} \text{ subject to } \sum_j |B_j| \le t. \qquad (S17)$$

where:

- Y_i is the response for the i-th observation.
- X_{ij} is the regressor for the i-th observation and j-th variable.
- β is the vector of coefficients.
- α is the intercept.
- $t \geq 0$ is a fitting parameter that controls the amount of shrinkage.

Due to the nature of this constraint, Lasso tends to produce some coefficients that are exactly zero, resulting in more interpretable models. The 't' parameter controls the amount of shrinkage applied to the coefficients; smaller values of 't' lead to more shrinkage, with some coefficients becoming precisely zero. Lasso attempts to retain the sound characteristics of subset selection and Ridge regression. It provides interpretable models like subset selection and exhibits the stability of Ridge regression.[37]

LASSO can be easily implemented in Python using the scikit-learn[1] library. The model's coefficients, which indicate feature importance, can be accessed directly using the *.coef_* attribute after training. Features with nonzero coefficients are those selected by LASSO, while those with coefficients reduced to zero are considered irrelevant for prediction.

1.19. LASSO LARS IC[37]

LASSO LARS IC (Least Angle Regression with Information Criteria) is a method that combines Least Angle Regression (LARS) with LASSO (Least Absolute Shrinkage and Selection Operator) while using Information Criteria (IC) to determine the optimal model complexity automatically. This approach is particularly useful for high-dimensional regression problems where variable selection and regularization are crucial.

LARS is an efficient algorithm for solving LASSO regression, following a path-based approach where variables enter the model gradually based on their correlation with the residuals. Unlike standard LASSO, where a penalty parameter λ controls sparsity, LASSO LARS IC leverages information criteria such as Akaike Information Criterion (AIC) and Bayesian Information Criterion (BIC) to determine the best stopping point along the LASSO path. This avoids the need for cross-validation, making the method computationally efficient. The model selection is performed by evaluating the goodness of fit while penalizing excessive complexity, ensuring a balance between variance and bias.[38]

In Python, LASSO LARS IC is implemented in scikit-learn[1] via the LassoLarsIC class (sklearn.linear_model.LassoLarsIC). It allows the automatic selection of the best model using either the AIC or BIC criteria. After fitting the model, the .coef_ attribute provides insights into selected features, and the .alpha_ attribute gives the optimal regularization parameter. This method is particularly useful for problems where an automated and principled approach to sparsity is needed without manual tuning of hyperparameters.

2. Validation Metrics of the Machine Learning Algorithms

The error metrics presented below are those recommended and used to measure the performance of ML models. It is important to note that models have many levels of depth, so a good metric is not always indicating a good model; several steps are necessary to avoid overfitting. However, most of these processes are based on these metrics.

The mean absolute error (MAE) value is an error metric widely used in machine learning to compare and evaluate regression models.[39] In particular, very low RMSE values in the training set may indicate model overfitting. The MSE is defined as $\frac{1}{n}\sum_{i=1}^{n}|y-\bar{y}|$,[39] where n represents the total number of samples, y is the observed value, and $\bar{y}$ is the value predicted by the algorithm. The Root Mean Square Error (RMSE), another helpful error metric, is conventionally defined as

$$\mathrm{RMSE} = \sqrt{\frac{1}{\mathrm{n}}\sum_{\mathrm{i=1}}^{\mathrm{n}}(\mathrm{y_i}-\bar{\mathrm{y}}_\mathrm{i})^2} \qquad \text{(S18)}$$

The squared difference $(y-\bar{y})^2$ penalizes larger deviations more severely, particularly significant outliers. The coefficient of determination, R^2, quantifies the extent to which an ML model accounts for variability in the predicted values. This statistical measure is important for evaluating the model's ability to fit the data. The R^2 value ranges from 0

to 1, where a value of 1 signifies a perfect fit, indicating that the model explains all variability in the dataset. A value approaching 1 suggests that most of the data's variability is captured by the model, whereas a value near 0 implies that the model fails to effectively describe the observed variations.

R^2, the coefficient of determination, is a metric that measures the proportion of variability in the predicted values by an ML model. The R^2 metric is defined as:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \bar{y}_i)^2}{\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2} \qquad (S19)$$

where n is the total number of samples, y is the observed value, $\bar{y}$ is the value estimated by the algorithm, and $\tilde{y}$ is the average of the observed values. It is a valuable statistic quantity for assessing how well the model fits the data. R^2 ranges from 0 to 1, where 1 indicates a perfect fit of the model to the data.[40] Conversely, when this value is close to 0, it suggests that the model does not adequately explain the variability in the data.

For classification tasks, accuracy is the most straightforward metric. It represents the proportion of correctly classified instances (both true positives and true negatives) out of the total number of instances. It is defined as:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \qquad (S20)$$

where TP = true positive, FP = false positive, TN = true negative and, FN = false negative.[41] A high accuracy indicates that the model performs well, although it can be misleading in imbalanced datasets where one class dominates.

Precision measures the proportion of correctly predicted positive instances (true positives) out of all instances predicted as positive (true positives and false positives). It is defined as:

$$\text{Precision} = \frac{TP}{TP + FP} \qquad (S21)$$

This metric is important when the cost of false positives can be high (for example, in spam detection, where incorrectly labeling a non-spam email as spam is undesirable, or in chemistry, incorrectly labeling a metal as a non-metallic material). In contrast, Recall, defined in equation 22, is critical when the cost of false negatives is high. Recall measures the proportion of correctly predicted positive instances (true positives) out of all actual positive instances (true positives and false negatives).[44] It is defined as:

$$\text{Recall} = \frac{TP}{TP + FN} \qquad (S22)$$

The F1 score (or F-measure) is the harmonic mean of precision and recall, providing a balanced measure of the two. It is particularly useful when there is an imbalance between the classes. The F1 score is defined as:

$$\text{F1 Score} = \frac{2}{\frac{1}{\text{Precision}} + \frac{1}{\text{Recall}}} \qquad (S23)$$

This is a single metric that balances precision and recall, making it useful for imbalanced datasets or when both false positives and false negatives need to be minimized.[44]

It is highly recommended to use all relevant metrics to compare your results with as many similar studies as possible and to ensure that your metrics remain comparable with future research. However, we emphasize that MAE, MSE, RMSE, and R^2 are widely used in regression problems, while accuracy, precision, recall, and F1-score are commonly used in classification problems.

References

1. Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; Vanderplas, J.; Passos, A.; Cournapeau, D.; Brucher, M.; Perrot, M.; Duchesnay, E.; *J. Mach. Learn. Res.* **2011**, *12*, 2825. [Crossref]
2. Liang, W.; Dai, H. In *Quantum Chemistry in the Age of Machine Learning*; Dral, P. O., ed.; Elsevier, 2023, ch. 10, p. 233. [Crossref]
3. Tipping, M. E.; *Bayesian Inference: An Introduction to Principles and Practice in Machine Learning*, vol. 3176; Springer: Berlin, Heidelberg, 2004, p. 41-62. [Crossref]
4. Lundberg, S. M.; Lee, S. I.; *SHAP: SHapley Additive exPlanations*, version 0.46.0; GitHub, USA, 2025. [Link] accessed in February 2025
5. Hosmer, D. W.; Lemeshow, S.; *Applied Logistic Regression*; John Wiley & Sons, Inc.: New York, 2000. [Crossref]
6. Nick, T. G.; Campbell, K. M. In *Topics in Biostatistics*, vol. 404; Humana Press, 2007, p. 273-301. [Crossref]
7. Weher, E.; *Biom. J.* **1977**, *19*, 83. [Crossref]
8. James, G.; Witten, D.; Hastie, T.; Tibshirani, R.; Taylor, J. In *An Introduction to Statistical Learning*; Springer: Cham, 2023, p. 69-134. [Crossref]
9. Comaniciu, D.; Meer, P.; *IEEE Trans. Pattern Anal. Mach. Intell.* **2002**, *24*, 603. [Crossref]
10. Yizong, C.; *IEEE Trans. Pattern Anal. Mach. Intell.* **1995**, *17*, 790. [Crossref]
11. Fausett, L.; *Fundamentals of Neural Networks, Architectures, Algorithms, and Applications*; Prentice-Hall: Englewood Cliffs, New Jersey, 1994.
12. Huang, Y.; Kangas, L. J.; Rasco, B. A.; *Crit. Rev. Food Sci. Nutr.* **2007**, *47*, 113. [Crossref]
13. Rauber, T. W.; *Redes Neurais Artificiais*; Universidade Federal do Espírito Santo, 2005. [Link] accessed in February 2025.
14. Ketkar, N.; Moolayil, J.; Ketkar, N.; Moolayil, J.; *Deep Learning with Python*; Apress LP, 2020. [Crossref]
15. Drucker, H.; Burges, C. J. C.; Kaufman, L.; Smola, A. J.; Vapnik, V.; *Adv. Neural Inf. Process. Syst.* **1996**, *1*, 155. [Crossref]
16. Mammone, A.; Turchi, M.; Cristianini, N.; *WIREs Comp. Statistics* **2009**, *1*, 283. [Crossref]
17. Kohavi, R.; *European Conference on Machine Learning*; Springer: Berlin Heidelberg, 1995, p. 174. [Crossref]
18. Freund, Y.; Mason, L.; In *International Conference on Machine Learning* Inc.: Bled,1999. [Crossref]
19. Holmes, G.; Pfahringer, B.; Kirkby, R.; Frank, E.; Hall, M.; *Machine Learning: ECML 2002, Lecture Notes in Computer Science*, vol. 2430; Springer, Berlin, Heidelberg, 2002, p. 161-172. [Crossref]
20. Lang, S.; Bravo-Marquez, F.; Beckham, C.; Hall, M.; Frank, E.; *Knowl. Based Syst.* **2019**, *178*, 48. [Crossref]
21. Landwehr, N.; Hall, M.; Frank, E.; *Mach. Learn.* **2005**, *59*, 161. [Crossref]
22. Le Gall, J.-F.; *Probability Surveys* **2005**, *2*, 245. [Crossref]

23. Witten, I. H.; Frank, E.; Hall, M. A.; *Data Mining: Practical Machine Learning Tools and Techniques*; Morgan Kaufmann Publishers Inc., 2011.
24. Elomaa, T.; Kaariainen, M.; *J. Artificial Intelligence Research* **2001**, *15*, 163. [Crossref]
25. Kingsford, C.; Salzberg, S. L.; *Nat. Biotechnol.* **2008**, *26*, 1011. [Crossref]
26. Géron, A.; *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*, 2nd ed.; O'Reilly Media: CA, USA, 2019. [Link] accessed in February 2025
27. Friedman, J. H.; *The Annals of Statistics* **2001**, *29*, 1189. [Crossref]
28. Natekin, A.; Knoll, A.; *Front. Neurorobot* **2013**, *7*, 21. [Crossref]
29. Freund, Y.; Schapire, R. E.; *J. Jpn. Soc. Artif. Intell.* **1999**, *14*, 771. [Crossref]
30. Schapire, R. E. In *Empirical Inference*; Springer: Berlin, Heidelberg, 2013, p. 37-52. [Crossref]
31. Rojas, R.; *TechRxiv* 2024. [Crossref]
32. Breiman, L.; *Mach. Learn.* **1996**, *24*, 123. [Crossref]
33. Bühlmann, P.; Yu, B.; *The Annals of Statistics* **2002**, *30*, 927. [Crossref]
34. Tin Kam, H.; *IEEE Trans. Pattern Anal. Mach. Intell.* **1998**, *20*, 832. [Crossref]
35. Breiman, L.; *Mach. Learn.* **2001**, *45*, 5. [Crossref]
36. Wu, L.; Yang, Y.; Liu, H.; *Computational Statistics & Data Analysis* **2014**, *70*, 116. [Crossref]
37. Tibshirani, R.; *J. R. Stat. Soc. Series B. Stat. Methodol.* **1996**, *58*, 267. [Crossref]
38. Fraley, C.; Hesterberg, T.; *Statistical Analysis and Data Mining* **2009**, *1*, 251. [Crossref]
39. Hodson, T. O.; *Geosci. Model Dev.* **2022**, *15*, 5481. [Crossref]
40. Balal, A.; Pakzad Jafarabadi, Y.; Demir, A.; Igene, M.; Giesselmann, M.; Bayne, S.; *Emerging Sci. J.* **2023**, *7*, 1052. [Crossref]
41. Olson, D. L.; Delen, D.; *Advanced Data Mining Techniques*; Springer Science & Business Media, 2008. [Crossref]