University of Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study program/ Specialization: MSc. Petroleum Geosciences	Spring semester, 2021 Open
Author: John Paul Masapanta Pozo	[illegible] …………………………………… John Masapanta Pozo
Faculty Supervisor: UiS – Prof, Equinor. Arild Buland	
Thesis title: MACHINE AND DEEP LEARNING FOR LITHOFACIES CLASSIFICATION FROM WELL LOGS IN THE NORTH SEA.	
Credits (ECTS): 30 ECTS	
Keywords: Lithofacies, well logs, machine Learning, deep learning, neural networks.	Number of pages: 194 + enclosure: 135 + supplement material/other: python pseudo-library and execution files, 59 pages. Stavanger, 06th July, 2021

John Paul Masapanta Pozo

Machine and Deep Learning for Lithofacies Classification from Well Logs in the North Sea.

Master Thesis Project for the degree of
Master of Science in Petroleum Geosciences.

Stavanger, July 2021
University of Stavanger
Faculty of Science and Technology
Department of Energy Resources

University of Stavanger

Abstract

Lithology identification by using well log data is an initial and fundamental step within petroleum geosciences; same that provides essential information about the subsurface and plays a crucial role in reservoir characterization. In addition, well log interpretation is a process that involves a great amount of data, same that is currently handled by experts in order to attain an accurate portrayal of the subsurface. However, as humanity enters the era of big data in companion of the increasing technological and computational development, data science and machine learning are progressively taking over the forefront of the future of the oil and gas industry in order to improve and optimize processes.

In consequence, the objective of current study is to explore and compare the potentiality of different supervised machine learning and deep learning algorithms to classify 12 different lithology facies by using the well log data of 118 wells located in the North Sea, same that are divided into three subsets for training, validation, and testing purposes. Additionally, we explore and discuss a machine-learning-based feature augmentation methodology as an attempt to improve the quality of the original dataset and consequently the final classification results. The analyzed models include standalone algorithms such as Logistic Regression, K-Nearest Neighbor, Supervised Vector Machines, Decision Trees, ensemble gradient boosting tree-based algorithms such as Random Forest, Categorical Gradient Boosting, Light Gradient Booting, and eXtreme Gradient Boosting, and a two-hidden layer Neural Network.

The results showed that by incorporating machine-learning-based feature augmentation every model experienced a performance enhancement, where trees-based gradient boosting algorithms along with random forest, and neural networks appeared to achieve the highest classification performances. Finally, we compare all the models performances and discuss possible reasons why although many algorithms offer high classification performances, they found problems to properly predict mixed-based lithologies, as well as how the interpreters' subjectivity impact the models performances, and possible future approaches to enhance our best classification accuracy of 82.5% on previously unseen objects.

Keyword: Lithofacies, Well Logs, Machine Learning, Deep Learning, Neural Networks.

Acknowledgments

I would like to express my gratitude to my supervisor Prof. Arild Buland for his time, openness, and continuous support and supervision. Thank you for bringing great comments and ideas along this great journey of continuous learning.

To my family for having a space for me in their minds and hearts regardless of the distance, in special to my parents who have been always there with me with no excuses. My profound gratitude to them for helping me to make my deepest dreams come true.

John. M.

Table of Content

List of Figures

List of Tables

Chapter 1

1. INTRODUCTION, DATASET DESCRIPTION, AND METHODOLOGY

1.1 Introduction

Lithofacies is a term evolved from the term facies that was defined by Amnaz Gressly in the nineteen century as the total sum of lithological and faunal characteristics of sedimentary rocks. These characteristics include mineral composition, organic-matter content, geomechanical properties, texture, stratification, grain size distribution, and degree of rounding and sorting (Wang and Carr, 2012a).

Lithofacies identification is important for many geological and engineering disciplines, goals of which might include palaeo-environmental context understanding (Wang and Carr, 2012b), hydrodynamic conditions determination sediments transport typology modelling (Gong et al., 2012), and porosity and permeability interpretations improvements (Akatsuka, 2000). Moreover, the accurate lithofacies identification has a special significance for reservoir characterization and stable hydrocarbon production and forecast. Standard methodologies to recognize and identify lithology include outcrops, core data collection and petrography, the first of which may not adequately reflect the reality of the subsurface while the second one offers limitations due to the costs it involves. Thus, great efforts are focused on building less costly qualitative and quantitative relationships between core data and conventional wireline logs, which normally includes gamma ray (GR), density (RHOB), neutron (NPHI), photoelectric index (PE), and resistivity logs (RES), in order to accurately identify lithofacies (Wang and Carr, 2012a). In addition, even though wireline logs are able to provide important information that leads petrophysicists into an accurate subsurface interpretation, the massive size of the data makes of it an extremely time-consuming assignment while, at the same time, it incorporates the interpreter's subjectivity into it.

In the other side, as humanity enters the era of big data in companion of the increasing technological and computational development, data science have taken over the forefront in several industry domains. In consequence, as part of the digitalization era, machine learning and deep learning have currently attracted great attention in petroleum geosciences because of its advantages in addressing big data issues in a relative small amount of time, introducing in this way exiting challenges and opportunities into the oil and gas industry (Huang et al., 2017; Zuo, 2017; Arabameri et al., 2020). These techniques, as summarized by Anifowose et al., (2017) and Mahmoud et al., (2021), are able to explore and learn from the hidden patterns and connections between large multivariate datasets in order to ultimately make informed decisions. Although, deep learning is considered a machine learning subfield, it is also considered as the evolution of machine learning as it performs based on an auto-regulated learning process similar to the human brain.

In addition, during the past decade, several researches have been performed to predict litholofacies based on wireline measurements by applying different artificial intelligence algorithms. These studies included the usage of naive bayes (NB) classifiers (Li and Anderson-Sprecher, 2006), artificial neural networks (ANN) (Zhang et al., 1999; Dubois et al., 2007), and support vector machines (SVM) (Al-Anazi and Gates, 2010; Sebtosheikh et al., 2015; Hall, 2016) to mention a few.
Finally, among the methods previously investigated, the current study aims to give a description and a fair comparison between the performances that logistic regression (LR), Support Vector Machines (SVM), k-nearest neighbor (KNN), decision trees (DT), Random Forest (RF), gradient boosting decision trees algorithms (GBDT), and neural networks (NN) can provide to sort out the lithofacies classification problem, same that ultimately will help in the near future to design a robust and automated methodology to carry out this assignment in a human performing-comparable manner.

1.2 Dataset description

The datasets used for the current study was taken from the 'Machine-Learning Lithology Prediction Contest' organized in the second semester of 2020 by FORCE, which is a cooperating forum managed by oil and gas companies and authorities in Norway that was created to improve exploration, enhance oil and gas recovery, and increase production efficiency throughout cooperation between the oil and gas industry, academia and the Norwegian government authorities.

The dataset used during the competition is composed by 118 wells from offshore Norway, location of which covers the south and the north of the Viking Graben as shown in Figure 1; besides, the wells penetrate a highly variable geology from the Permian evaporites in the south and the deeply buried Brent delta facies in the northern area of the North Sea (NPD, 2021).

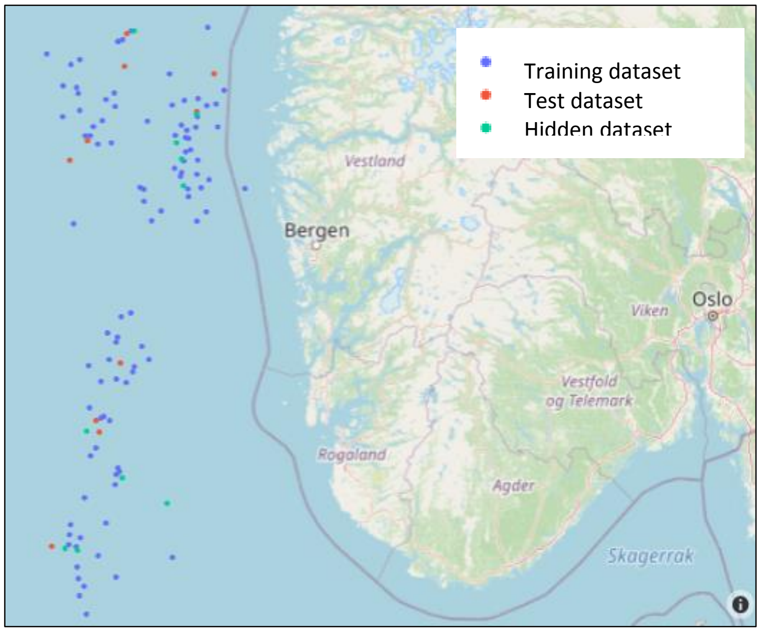


Figure 1 Wells geographical location.

In addition, the provided data is conformed by three different data subsets serving to different purposes each. The training, open test, and hidden test subsets are composed by 98, 10, and 10 wells, respectively. In addition, it is necessary to note that only the first two subsets were available for the contestants during the FORCE competition, while the hidden test subset was unavailable for them and was only used for assessing the final score that leaded to define the competition winner. In fact, instead of using standard performance metrics for assessing the models provided by the competitors, a new scoring function based on a penalty matrix was introduced, which in brief attempts to penalize misclassification similarly as a petrophysicist would do (See Appendix H).

Table 1 summarizes the petrophysical wireline logs measurements, and additional metadata including lithostratigraphy, UTM location coordinates, measured depth, and the interpreted lithofacies that make up the datasets as well as their missing data summaries, which at first glance appear to represent highly sparse datasets, fact that may influence while implementing supervised learning for lithofacies classification.

Table 1 Training, open test, and hidden test datasets description and missing data summary.

MEASURED PROPERTIES				
DESCRIPTION	**LOG**	**Missing Data Percentages (%)**		
		Training	Open	Hidden
Caliper	CALI	7.51	4.13	2.81
Deep Resistivity	RDEP	0.94	0.04	0.01
Medium Resistivity	RMED	3.33	0.43	8.02
Shallow Resistivity	RSHA	46.12	71.42	79.02
Flushed Zone Resistivity	RXO	72.03	78018	92.73
Micro Resistivity	RMIC	84.95	91.73	87.60
Spontaneous Potential Log	SP	26.16	51.29	61.83
Sonic (Shear Slowness)	DTS	85.08	68.40	40.46
Sonic (Compressional Slowness)	DTC	6.91	0.60	3.35
Neutron Porosity	NPHI	34.61	23.94	21.11
Photoelectric Absorption Factor	PEF	42.62	17.02	17.94
Raw gamma data	GR	0.00	0.00	0.00
Bulk Density	RHOB	13.78	12.40	7.78
Density Correction	DRHO	15.60	18.44	8.28
Bit Size	BS	41.68	51.04	39.14
Differential Caliper	DCAL	74.47	90.12	64.78
Average Rate of Penetration	ROPA	83.57	59.21	47.53
Spectral Gamma Ray	SGR	94.07	100.00	99.07
Weight of Drilling Mud	MUDWEIGHT	72.99	85.18	100.00
Rate of Penetration	ROP	54.29	50.06	25.53
METADATA				
DESCRIPTION	**NAME**	**Missing Data Percentages (%)**		
		Training	Open	Hidden
Well Name	WELL	0.00	0.00	0.00
Measured Depth	DEPTH_MD	0.00	0.00	0.00
UTM coordinate	X_LOC	0.92	0.04	0.01
UTM coordinate	Y_LOC	0.92	0.04	0.01
True Vertical Depth	Z_LOC	0.92	0.04	0.01
Lithostratigraphic Group	GROUP	0.11	0.00	0.00
Lithostratigraphic Formation	FORMATION	11.70	5.17	6.66
Interpretation Confidence Quality	LITHO_CONF	0.00	0.00	0.00

Moreover, in regard to the interpretation of the wells, the Norwegian company EXPLOCROWD, a consultancy and services company outsourced by the FORCE organizers committee, provided the interpretation for 104 wells, and 14 more wells were interpreted and provided by the data science and software development company IG2.

1.3 Methodology

Whenever one think about implementing machine learning for solving a particular problem, the first question one should ask is if ML is the most suitable approach for solving it. Additionally, considering that machine learning will never perform perfectly in real-life problems there are a set of considerations must be fulfilled before commencing a ML project. These considerations include that a large amount of data to be available, that a very high accuracy not being desired, and that the problem is deeply understood so it would provide a basis to develop suitable algorithms (Awad and Khanna, 2015). Consequently, once the basic conditions are met, the process we will follow while developing the current machine-learning project can be describes in the following workflow diagram.

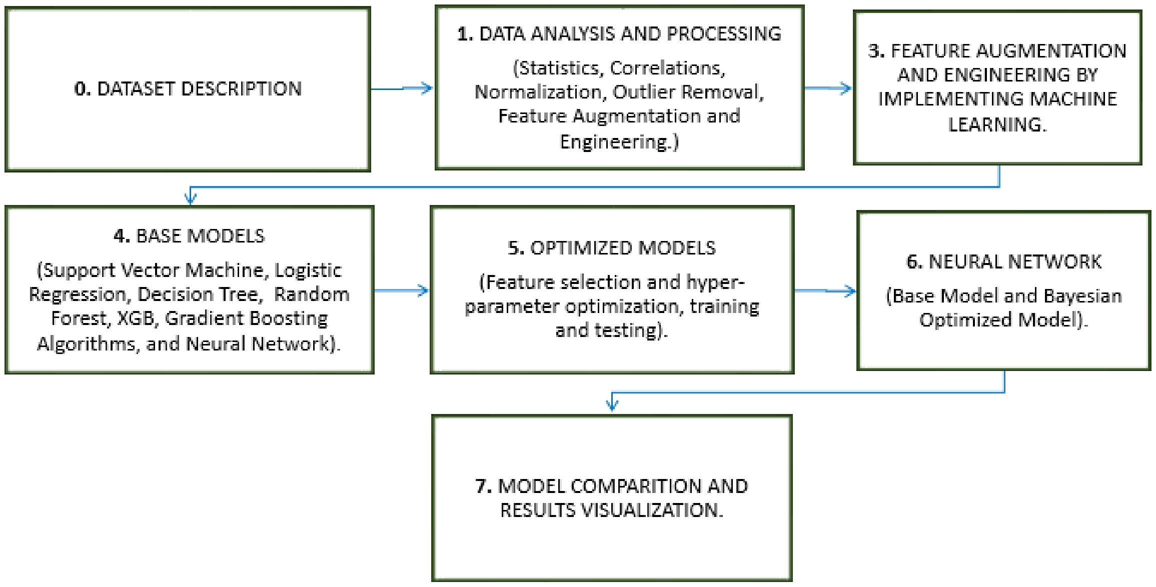

0. DATASET DESCRIPTION
1. DATA ANALYSIS AND PROCESSING
(Statistics, Correlations, Normalization, Outlier Removal, Feature Augmentation and Engineering.)
3. FEATURE AUGMENTATION AND ENGINEERING BY IMPLEMENTING MACHINE LEARNING.
4. BASE MODELS
(Support Vector Machine, Logistic Regression, Decision Tree, Random Forest, XGB, Gradient Boosting Algorithms, and Neural Network).
5. OPTIMIZED MODELS
(Feature selection and hyper-parameter optimization, training and testing).
6. NEURAL NETWORK
(Base Model and Bayesian Optimized Model).
7. MODEL COMPARITION AND RESULTS VISUALIZATION.

Figure 2 Machine and deep leaning methodology workflow.

Chapter 2

2. SUPERVISED LEARNING THEORETICAL BACKGROUNG

2.1 STANDARD MACHINE LEARNING ALGORITHMS

Supervised learning is a learning mechanism that infers and learns from the underlying relationships between the input data and a target variable that might be a continuous numerical attribute or a multiclass categorical attribute for regression or classification problems, respectively. The learning task uses labeled data that comprises a set of observed vectors normally called predictors or features and a desired output called supervisory signal or class label. Broadly, the purpose of these mechanisms is to generalize the underlying relationship between the feature vectors and the supervisory signal in order to be able to predict the output while unlabeled input instances are used (Awad and Khanna, 2015).

The training process is deeply dependent on the training data quality, which means that a well-trained supervised machine-learning algorithm could accurately predict the output for unfamiliar or unobserved data instances only if the input data used for training the algorithm has a high-level quality. In contrast, if a poor-quality input is used for training, this might derive in overfitting problems, which represents the difficulty for an algorithm to generalize the underlying predictors-target relationships that will derive in an unsuccessful regression or classification performance.

2.1.1 Logistic Regression

Logistic regression is a statistical model that follows almost the same theory as linear regression; however, it is considered as a probabilistic algorithm used for solving binary or multiclass problems by using a logistic function that can be mathematically expressed as follows $\frac{e^z}{1+e^z}$, were $z \in [-\alpha, \alpha]$. In general, a logistic regression model predicts the probability of occurrence of a specific event by modeling the relationship between a dependent variable X and a categorical outcome Y (Awad and Khanna, 2015). Mathematically the previously stated logistic function can be expressed as

$$P(Y|X) = \frac{e^{\beta_o + \beta_1 X}}{1 + e^{\beta_o + \beta_1 X}} \tag{1}$$

were β_o and β_1 represent the estimated log-odds of a unit change for their respective input they are associated with, or in other words they can be seen as weights that translates any change in the input variables to the probability outcome. In addition, by extracting the inverse of the logistic function a new function called logit or log-odds is obtained which allows generating the logistic regression coefficients, β_o and β_1 for a one-predictor-based case.

$$logit\big(P(Y|X)\big) = \ln\left(\frac{P(Y|X)}{1 - P(Y|X)}\right) = \beta_o + \beta_1 X \tag{2}$$

Once the log-odds is calculated, a logistic function receive it as input, $\beta_0 + \beta_1 X$, and returns the likelihood probability $P(Y|X)$ of the occurrence of the event Y belonging to a positive class when the variable X is used as input as depicted by Figure 3.

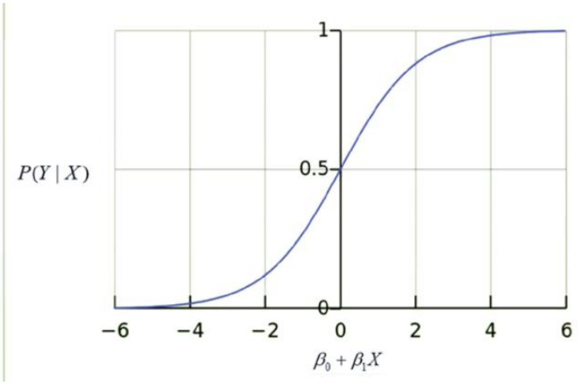


Figure 3 Logistic Function (allows transforming the log-odds parameters to the probability of an instance belonging to a certain positive class).

To conclude, as in the case of a linear regression, we are interested on the intercept β_0 and gradient β_1 coefficients, but by the aid of a logistic function, we transform these values into the probability of a value belonging a particular class known as positive class.

2.1.2 K-Nearest Neighbor, KNN

The K-nearest neighbor classification methodology, KNN for short, is a fairly simple clustering classification algorithm which identifies the group of k-objects in the training set that are closest to the test object and assigns a label based on the most dominant class in the neighborhood the instance belongs to (Awad and Khanna, 2015). KNN belongs to a particular family of algorithms called instance-based learning methods. The inference, learning, and predictions performed by a direct comparison of new samples with previously existing instances based on the distance between each other. This methodology could be applicable for classification, regression, and clustering purposes (Bonaccorso, 2020). The main idea of the algorithm can be explained if we consider a bunch of data samples $x_1, x_2, \dots, x_n$, each of which has a dimensionality equal to N. Mathematically expressed as follow

$$X = \{x_1, x_2, \dots, x_n\} \ where \ x_i \in \mathbb{R}^N \tag{3}$$

Then we can introduce a distance function $d(x_1, x_2)$ as a function of a new factor p that might take different values. For instance, *p=2* represents the Euclidean distance and *p=1* represents the Manhattan distance to mention a few.

$$d_p(x_1, x_2) = \left(\sum_{i=1}^{N} \left| x_1^{(j)} - x_2^{(j)} \right|^p \right)^{\frac{1}{p}} \tag{4}$$

The results obtained by the KNN algorithm when assigning an instance to a particular class might be diverse when different distances are implemented. To exemplify this Figure 4 helps visualize how the computed distance between the point A (0, 0) and B (15, 10) varies when different p values are used (Bonaccorso, 2020).

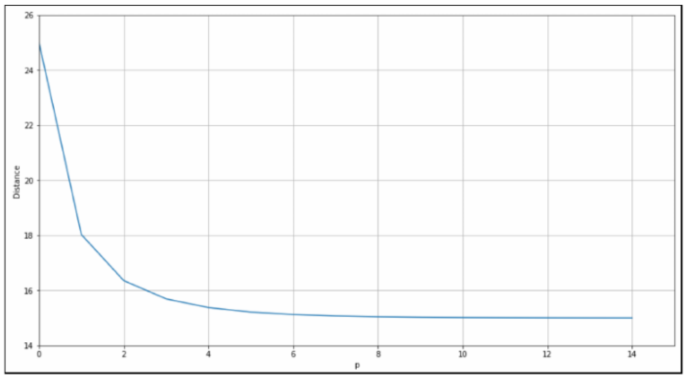


Figure 4 Distance between (0, 0) and (15, 10) as a function of parameter (Bonaccorso, 2020).

Finally, once every the distances is computed, the KNN algorithm determines the k closest samples for each training point; thus, when a new sample is presented the process is repeated with a predefined value of k samples (Bonaccorso, 2020). The philosophy of the KNN methodology is that similar samples should share their features or predictors, which normally may provide high training and testing accuracies; however, since every distance has to be computed every time a new instance is introduced, it might become an extremely slow process for massive datasets.

2.1.3 Support Vector Machines, SVM

Support Vector Machines, SVM, is a machine-learning algorithm that from a geometrical perspective aim to find the equation of a multidimensional surface that best separated different classes in the feature space. SVM is a discriminant technique that solves the convex optimization problem analytically meaning that it will always return the same hyper-plane parameter every time the model is initialized with the same parameters. In contrast, other popular algorithms for classification problems like perceptron accomplishes its solutions depending on the parameters initialization and termination criteria making of it an heuristic approach (Awad and Khanna, 2015).

Several of the characteristics that make of SVM a powerful machine-learning technique for a large range of problems are that it is uses maximum margin separator and a kernel technique. As a maximum margin separator, SVM not only aims to minimize or maximize a cost function but also imposes an additional constrain or condition to the location for the hyper-plane, which has to be situated in a way that the distances between classes are maximized as an attempt to generalize its solution.

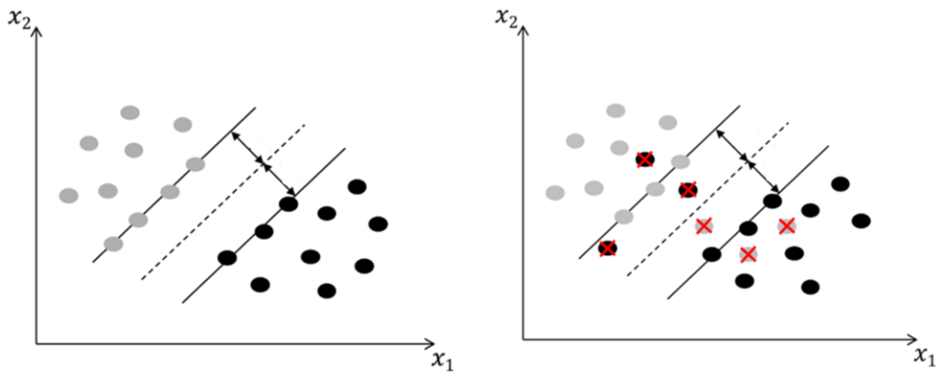


Figure 5 Hard (left) and soft (right) separating margins implemented on SVM (Awad and Khanna, 2015).

In this context, Figure 5 depicts two scenarios in which SVM constructs a separating hyper-plane to properly classify most of the instances encounter in a training set when the data is completely separable when there is not such a case. These hyper-planes are named hard-margin and soft-margin SVM, respectively. The first attempts to maximize the distance between classes, while the second allows for some classification error in the neighborhood of the separating boundary or hyper-plane (Awad and Khanna, 2015).

Besides, SMV includes kernel trick functionality that helps mapping the original data into a higher-dimensional space before solving a particular task considering that often the data involved is not linear separable in the original input space as exemplified on Figure 6. The principal objective for dimensionality transformation is to simplify the computational requirements for constructing a linear separator in a higher dimensional space where a linear separator would be able to discriminate between different classes.

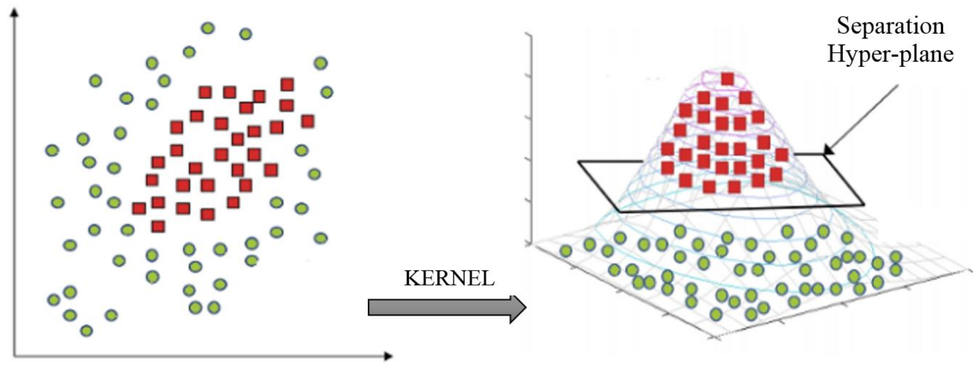


Figure 6 Support vector machines kernel trick functioning (Sharma, 2019).

In addition, kernel selection is highly dependent on data nature. For instance, a linear kernel is the simplest approach for solving medium complexity problems, a polynomial kernel is widely used for task related to image processing, ANOVA RB kernel is reserved for regression task mainly, and Gaussian and Laplace Radian Basis Function (RBF) kernels are mostly applied in the absence of prior knowledge. However, the great majority of them provide a better model performance once feature or data dimensionality reduction are performed. Moreover, SVM is a sparse technique that requires all the training data to be available in order to learn its optimal parameters. Once these parameters are identified, SVM will depend only on a significantly small subset of training instances called support vectors that would become the margins of the hyper-planes in the case of a multidimensional feature space.

Ultimately, the complexity of the classification task with SVM depends on the number of support vectors rather than the dimensionality of the feature space; thus, the number of support vectors that are ultimately retained by the model depends on the class separability. Therefore, SVM performance is highly dependent on the training and test data distributions and when trained with data that are not representative for the overall data population, hyper-planes are prone to poor generalization (Bonaccorso, 2020).

2.1.4 Decision or Classification Trees

Decision or classification trees are used to classify a data instance into a predefined set of classes based on its attributes called features or predictor in machine learning. Decision trees could be seen as expert decision or clarification systems, which partially attempt to mimic and automate the underlying knowledge of an expert on the entrusted task. Some of the advantages of decision trees models are that they are simple to implement and its self-explanatory characteristic help represent them graphically as hierarchical structures (Rokach and Maimod, 2014).

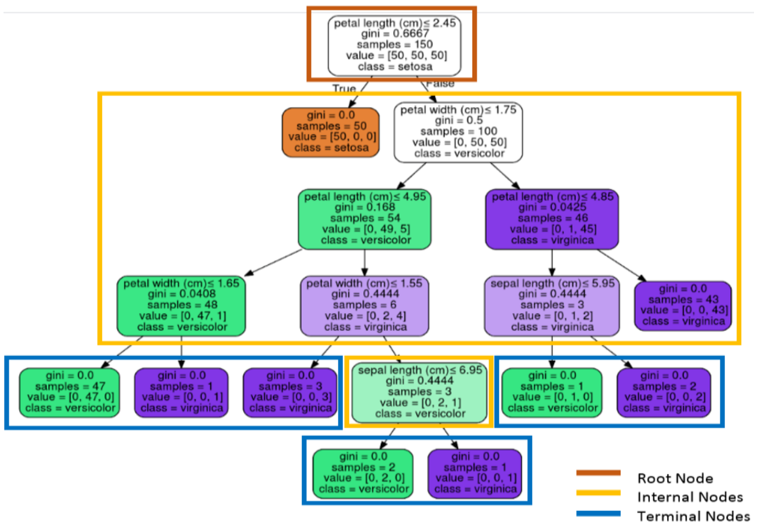


Figure 7 Decision tree applied on IRIS dataset (Pedregosa et al., 2011).

Further, a decision tree is as classifier expressed as a recursive partition on the instance space consisting of different types of nodes called root node, internal or test node, and terminal nodes also called leaves. The root node can be seen as the initial point with no incoming edges, while the internal node splits the instance space into two or more partitions according to a certain discrete attribute value to finally get to the terminal nodes, which represent the most appropriated outcome reached through the previous internal nodes. To exemplify what was previously stated, Figure 7 shows the implementation of decision trees for the well-known IRIS dataset where the root, internal, and terminal nodes can be identified.

In addition, the driving concepts for decision trees, entropy and information gain, will be discussed based on the example shown on Figure 8.

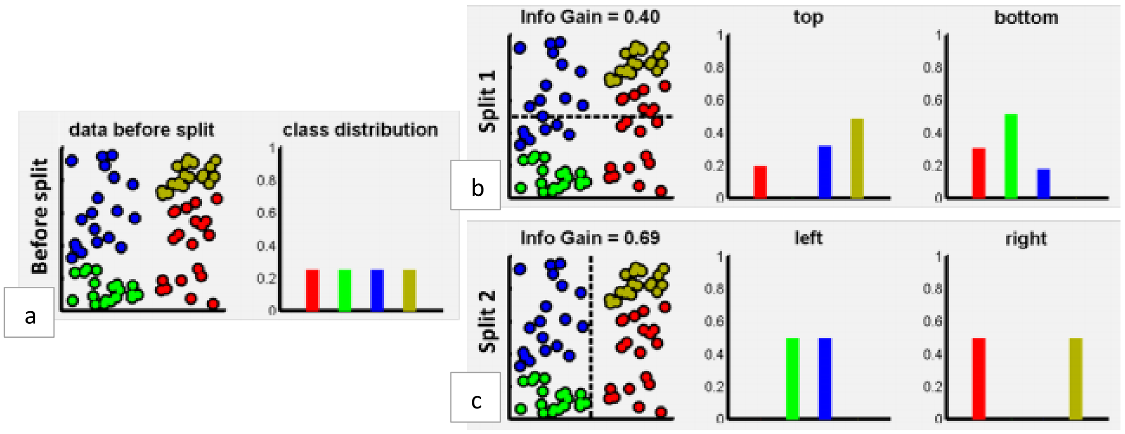


Figure 8 Information gain for discrete distributions. (a) Complete dataset before splitting. (b) Dataset after a horizontal split. (c) Dataset after a vertical split. (Criminisi et al., 2011).

Figure 8a shows a number of data points distributed on a 2D space color-labeled by different data classes. If we split the data horizontally or vertically as shown by Figure 8b and Figure 8c, respectively, two sets of data with lower entropy for the first splitting case and with higher entropy for the second one are produced. The information gain for each split type could be mathematically computed by equation (5), where $H(S)$ represents the entropy for a generic set of training points S.

$$I = H(S) - \sum_{I\epsilon[1,2]} \frac{|S^i|}{|S|} H(S^I) \tag{5}$$

The lower entropy split gives an information gain of I=0.4, while higher entropy splitting gives I=0.6 meaning that a better class separation is achieved by the second way of splitting the data as visible on Figure 8 (Criminisi et al., 2011).

To summarize, classification trees function by simply navigating every instance from the root of the tree down until they reach any specific leaf according to the outcome of the internal nodes and the information gain metric obtained afterwards. Note that the internal nodes are able to test both numerical and nominal attributes. Moreover, according to (Breiman et al., 1984), the decision trees accuracy is mainly influenced by its complexity, which could be measured by either the total number of nodes, total number of leaves, tree depth, or number of predictors used, or any possible combination of them.

2.2 ENSEMBLE MODELS

Ensemble methods are techniques that aim to combine multiple models into one to improve their overall performance. These methods fall into two broad categories defined as sequential and parallel ensemble techniques. Sequential ensemble techniques generate base learners sequentially where data dependency resides, so every other subsequent learner depends on the previous learner performance in order to get an optimized performance. Parallel ensemble techniques, in the other hand, generate their base learners in parallel in order to encourage independence between every learner, which aims to reduce their final performance error.

2.2.1 Random Forest, RF

Random Forest is a parallel machine learning technique founded on the decision trees theory in which decision trees are not treated and used as individual entities anymore. In their stead, all decision trees, also known as weak learners, are combined together in a newish emerged and robust predictive technique known as ensemble learners that have been mostly confined to classification tasks. They use a random feature sample to build each independent tree as an attempt to reduce variance by decreasing the correlation between each decision tree output.

Additionally, these kinds of machine learning algorithms are highly influenced by a number of important components but mostly by its randomness while constructing every individual decision tree differently from one another. Besides, forest randomness, which is introduced into the trees during the training phase, provides the model with high robustness with respect to noisy and imbalanced data. Moreover, randomness is normally achieved either by random training data sampling, also known as bagging, or by randomized node optimization (Criminisi et al., 2011).

In general terms, random forest training happens by optimizing the parameters of decision trees, known as weak learners, at each split node j via:

$$\theta^*_J = \arg\max I_j \tag{6}$$

For the specific case of classification problems, the objective function I, as stated in the previous section, is the information gain computed by equation 6. Subsequently, once every decision tree has been trained independently and efficiently, all these 'weak' predictions are combined into a single forest prediction by an averaging operation using the following expression in the case of classification tasks.

$$p(c|v) = \frac{1}{T}\sum_{t=1}^{T} p_t(c|v) \tag{7}$$

Where T represents the total number of decision trees, v represents an attribute instance, and $p(c|v)$ is the ensemble posterior probability distribution of an attribute instance belonging to any discrete class (Criminisi et al., 2011). In other words, classification forest produce probabilistic outputs as they return an entire class distribution as illustrated in Figure 9.

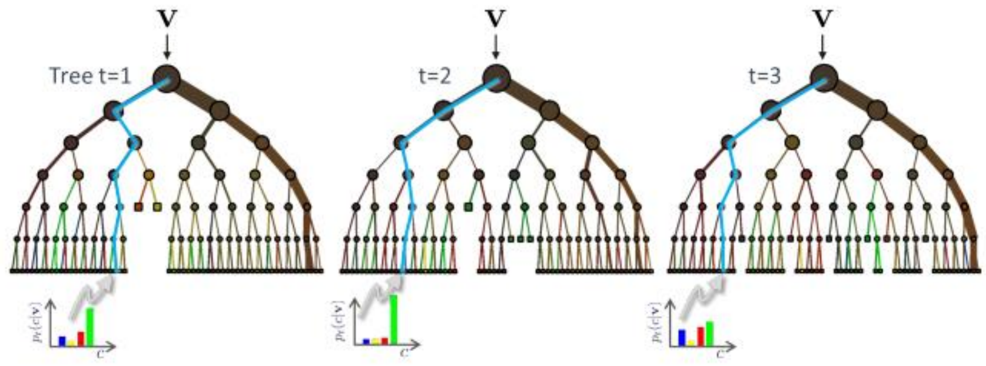


Figure 9 Three different decision trees part of a random forest reproducing different probability distribution outputs (Criminisi et al., 2011).

Figure 9 describes how the same input value v is conducted differently from the root node until it reaches a leaf node; here every posterior $p_t(c|v)$ is read off and averaged together to an ensemble posterior $p(c|v)$.

Finally, random forest algorithms generally yield to high accuracies and generalization; however, their performance is importantly affected by several parameters such as their size, number of discrete classes to be classified, classes' similitude, training data noise or quality, and individual performance of each decision tree included in the random forest.

2.2.2 Gradient Boosting Decision Tree Ensembles, GBDT

Gradient Boosting Machines, GBMs for short, are a family of powerful machine-learning techniques considered to be part of the sequential ensemble models category in which each independent learner acquires information, learns, and gets constructed based on previous learners' mistakes by performing gradient descent in a functional space in order to optimize their overall performance in subsequent steps.

Unlike common ensemble techniques like random forest, which rely on simple averaging techniques to get the final model, boosting ensemble methods base their functionality on consecutively training each base-learner with respect to the error obtained by the whole ensemble on previous stages. In addition, their robustness is partially attributed to their high flexibility while using pre-established or customized loss functions during the optimization stage, which has made of them very successful in practical applications and data challenges worldwide compared to single strong machine-learning models (Ghori et al., 2019).

Gradient Boosting Machines rely on three main elements that are the loss function, the base weak learner involved in the process, and the additive model receiving all the weak learners while a gradient descent process is performed in order to minimize the final additive performance loss.

Moreover, tree-based gradient boosting ensemble algorithms, which could be considered as a subgroup of Gradient Boosting Machines (GBM), were originally designed to be highly scalable to large datasets in different scenarios. These methods are able to run more than ten times faster than other existing popular algorithms. Mathematically a tree-based GB ensemble model can be expressed in the form

$$\widehat{y_i} = \sum_{k=1}^{K} f_k(x_i), \quad where\ f_k \in \mathcal{F} \tag{8}$$

where, K is the number of trees, f is a function part of the functional space $\mathcal{F}$, and $\mathcal{F}$ is the set of possible classification or regression trees known as CARTs. Additionally, considering that tree boosted and random forests are really the same model with the only difference in how they are trained. In consequence, as any other supervised machine-learning model the first step prior to enter the training stage is to define an objective function (Prokhorenkova et al., 2019).

Moreover, similarly to any gradient boosting model, tree-based models build an additive expansion of the objective function by minimizing a loss function which introduces a regularization term Ω in order to control the complexity of the base tree learners as follows:

$$obj(\theta) = \sum_{i}^{n} l(y_i - \widehat{y_\imath}) + \sum_{k=1}^{K} \Omega(f_i) \tag{9}$$

where, l is a differentiable convex loss function that measures the difference between the prediction $\widehat{y_\imath}$ and the target $\widehat{y_\imath}^{(t)}$. Additionally, in order to define the regularization term or complexity of the tree $\Omega(f)$, we need first to define a decision tree $f(x)$ as

$$f_i(x) = w_{q(x)}, \;\; w\epsilon\mathbb{R}^T, \;\; q: \mathbb{R}^d \rightarrow \{1,2, \dots., T\} \tag{10}$$

where, w is a vector containing the scores on the tree leaves, q is a function that assigns each data to its corresponding leaf, and T is the number of leaves. Thus, the regularization term can be mathematically expressed as

$$\Omega(f_i) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j{}^2 \tag{11}$$

Where, T represents the number of leaves of the tree, w are the output scores of the leaves, and γ controls the minimum loss reduction gain needed to split an internal node (Chen and Guestrin, 2016). This regularization term Ω penalizes the complexity of the model and serves as a regularization technique that helps to smooth the final learn weights to avoid over-fitting.

Additionally, to exemplify how boosting tress work let's assume the mean squared error (MSE) as loss function, then the objective function could be redefined as

$$obj^{(t)} = \sum_{i=1}^{n} \left[g_i f_i(x_i) + \frac{1}{2} h_i f_t{}^2(x_i) \right] + \Omega(f_i) \tag{12}$$

where,

$$g_i = \partial_{\hat{y}_i^{(t-1)}} l\left(y_i, \hat{y}_i^{(t-1)}\right) \tag{13}$$

$$h_i = \partial^2_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)}) \tag{14}$$

are the first and second derivatives of the objective function, normally called gradient statistics. Then, the objective function is reformulated as follows

$$obj^{(t)} = \sum_{i=1}^{n} \left[G_i w_i + \frac{1}{2}(H_i + ℷ) w_i^2 \right] + \gamma T \tag{15}$$

where, $G_i = \sum_{i \epsilon I_j} g_i$ and $H_i = \sum_{i \epsilon I_j} h_i$. Finally, after solving the equation for w, we get a final expression that measures how good a tree structure is.

$$w^*_i = -\frac{G_i}{H_i + ℷ} \tag{16}$$

$$obj^* = -\frac{1}{2} \sum_{i=1}^{T} \frac{G_i^2}{H_i + ℷ} + \gamma T \tag{17}$$

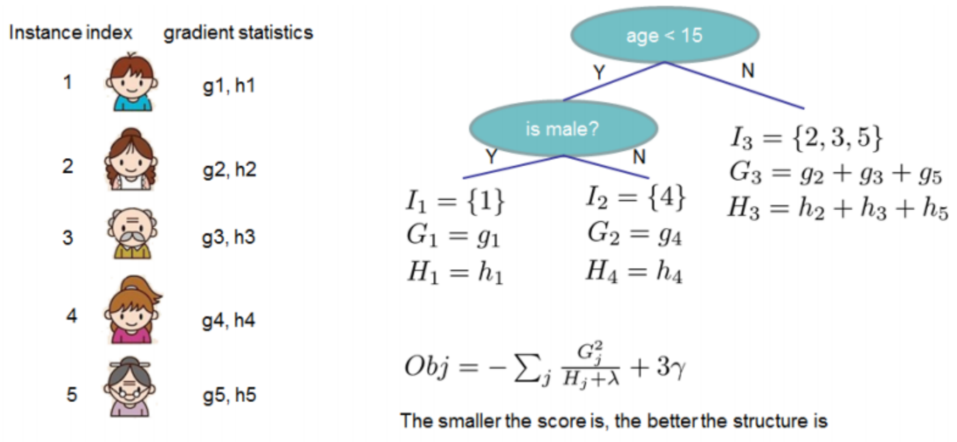


Figure 10 Boosting trees visual example training functionality (Chen and Guestrin, 2016).

Sometimes, understanding the whole process seems complicated, so we can intuitively understand the boosting trees training functionality by the following particular example described on Figure 10. Here, initially the statistics g_i and h_i are pushed until each instance reaches the leaves it belongs to, then these statistics are summed up together, and the objective function is used to calculate how good the tree is, similarly to impurity in decision tress but taking into account the model complexity (Chen and Guestrin, 2016).

Once, the utility of a tree has been calculated, the new step is to enumerate all possible trees and select the one that provides the best gain node by node. This can be computed by the following expression

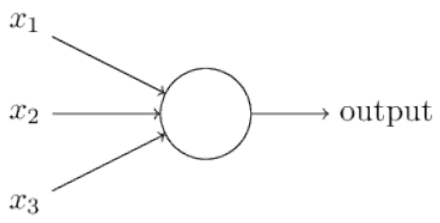
$$Gain = \frac{1}{2}\left[\frac{{G_L}^2}{H_L + λ} + \frac{{G_R}^2}{H_R + λ} - \frac{(G_L + G_L)^2}{H_L + H_R + λ}\right] - \gamma \tag{18}$$

which sums up the gain of the new leaves and subtracts the gain obtained by the original leaf and then compare the value to the minimum accepted gain γ to decide if performing a new split is beneficial or is not.

2.3 NEURAL NETWORKS AND DEEP LEARNING

Neural networks are an elegant programming paradigm in which computers learn how to solve a particular problem without explicitly being told how to solve it. Instead, computers learn by themselves how to overcome the problem at hand by solely using observed data; however, even though neural networks were promising in past years, it was only possible to properly train a neural network when deep neural networks were discovered in 2006.

Initially, in order to understand the mainly used neurons called sigmoid neurons, perceptrons need to be defined beforehand. To visually understand perceptron functionality let’s assume some binary inputs $x_1, x_2,\ x_3$, which are afterwards weighted internally to produce a binary output for an data instance belonging to a particular class, which is determined by comparing the weighted sum $\sum_j w_j x_j$ to a pre-established threshold as described on Figure 11.



$$\text{output} = \begin{cases} 0 & \text{if} \quad \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if} \quad \sum_j w_j x_j > \text{threshold} \end{cases}$$

Figure 11 Perceptron functionality diagram for a binary output.

Perceptrons can, in brief, be seen as decision makers based on evidenced data which may lead to different decision making models by adjusting the weights w and threshold. In this way, perceptrons are able to solve simple decision-making problems; however, by connecting different perceptrons parallels, a new and much more powerful structure called neural network becomes possible as described in Figure 12. Consequently, much more complex or abstract decision-making problems can be solved when preceding layer's outputs are considered as the new inputs for the subsequent layer in the neural networks (Nielsen, 2015).

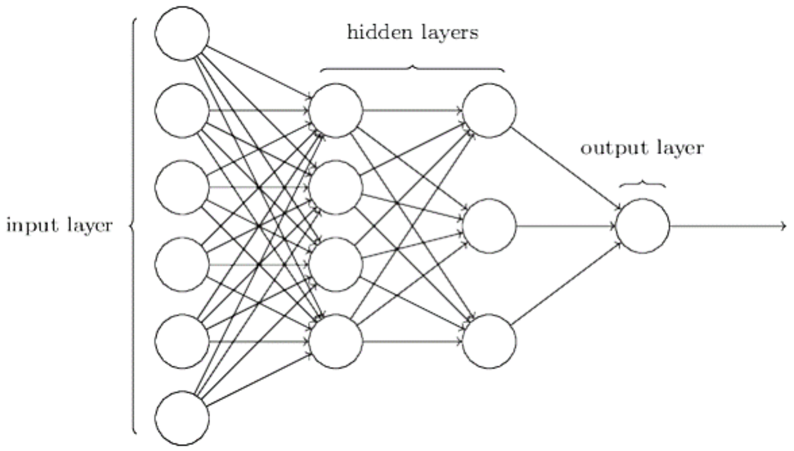


Figure 12 Neural Network Basic Structure.

More formally, the minus threshold term $-treshold$ is known as bias b, which can be understood as an analogous to the constant term in a linear function and allows perceptrons to better fit the observed data. Thus, the previous definition of a perceptron can be readjusted as follows.

$$output = \begin{cases} 0 & if \quad w \cdot x + b \leq 0 \\ 1 & if \quad w \cdot x + b > 0 \end{cases}$$

Furthermore, making a perceptron-based artificial neural network learn is a complicated task since this is normally achieved by continually changing the bias and weights so that mistaken predictions are correctly predicted. However, slight changes in these parameters lead to completely different results while using perceptron-based networks. In consequence, this problem is overcome by introducing a new type of artificial neuron called sigmoid neurons, which do not affect greatly the outputs when small changes in the weights and biases are performed, fact that is crucial to allow neural networks to learn.

Sigmoid neuron can be understood in almost the same way as perceptron, with the difference that the output sigmoid neurons provide may take any possible value between 0 and 1 by the aid of a sigmoid function also known as activation function. This output can be mathematically expressed as $\sigma(wx + b)$, where σ represents the sigmoid function (See Figure 3) defined by

$$\sigma(z) = \frac{1}{1 + e^{-z}} = \frac{e^{z}}{1 + e^{z}} \tag{19}$$

Consequently, the output that sigmoid neurons provide can be redefined as:

$$output = \frac{1}{1 + \exp(-\sum_j w_j b_j - b)} \tag{20}$$

where the sigmoid or activation function smoothness help to maintain the output with no substantial changes when the weights w and bias b are slightly varied during the training process.

Moreover, before entering the training stage a optimization cost function has to be defined, which in general terms is a measure of how well a neural network does with respect to the expected outputs. Depending on the problem to be solved the cost function may take different forms for regression, binary classification, and multi-class classification. Lastly, the cost function as a function of the weights and biases is optimized during the training process by implementing a gradient descent algorithm. In addition, optimizing a cost function could be achieved analytically by implementing calculus; however, this becomes almost impossible when the neural network involves hundreds, millions, or even billions of weights w and biases w to be optimized (Nielsen, 2015).

Furthermore, a reduced way to explain how gradient descent works in neural networks is to consider a particular cost function C which is a function of m variables $v_1, v_2, \dots, v_m$. Then any change ΔC in the cost function C produced by small changes $\Delta v = (\Delta v_1, \Delta v_2, \dots, \Delta v_m)^T$ is expressed as

$$\Delta C \approx \nabla C \cdot \Delta v \tag{21}$$

where the gradient ∇C is the transposed (T) vector made of the partial derivate of the cost function with respect to each variable weight w and bias b contained in the network, v_s for simplification.

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \dots, \frac{\partial C}{\partial v_m}\right)^T \tag{22}$$

so if we choose a change $\Delta v = -\eta \Delta C$, where η represents a parameter called learning rate, this guarantees that the cost function will always decrease $\Delta C \leq 0$ in order to find its global minimum (Nielsen, 2015).

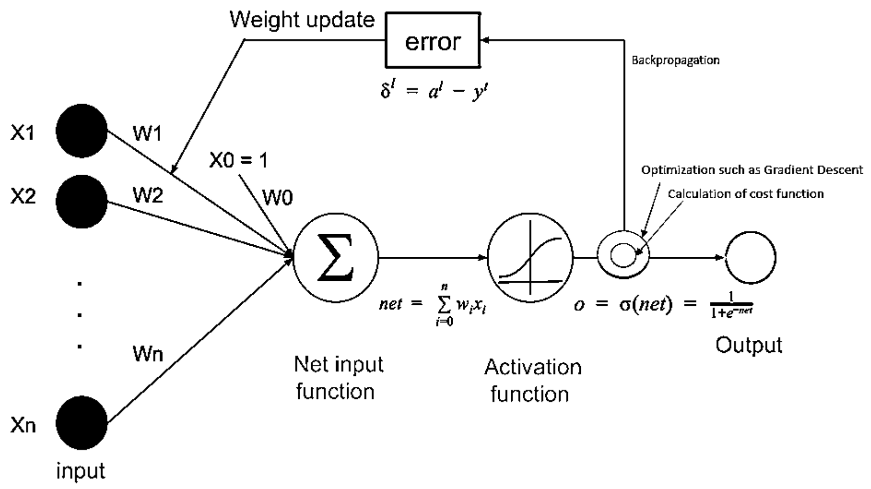

Weight update
error
$\delta^l = a^l - y^l$
Backpropagation
X1
X2
Xn
W1
W2
Wn
X0 = 1
W0
Optimization such as Gradient Descent
Calculation of cost function
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1+e^{-net}}$
Net input function
Activation function
Output
input

Figure 13 Neural Network training optimization process by implementing back propagation (Nielsen, 2015).

Finally, once the weights and biases have been calculated, the error is back propagated meaning that an error vector is calculated from the last layer in order to understand how the cost varies with earlier weights and biases. This final process is called back propagation and is profoundly explained in (Nielsen, 2015). The complete training process of neural networks while implementing back propagation to optimize all the trainable variables is depicted in Figure 13.

2.3.1 Evaluation Metrics for classification

Evaluating the performance a machine-learning model is a fundamental aspect during training, validation, and testing stages of a machine-learning project in order to understand the quality of the output and the influence input data has on this. Normally in real-life applications, the datasets to be used during classification tasks are imbalanced, meaning that some classes have fewer samples than the other classes, which are referred as minority and majority classes, respectively. This imbalance represents a great challenge while solving classification problems by machine learning since it might cause a bias in the prediction towards the majority class when standard machine learning are implemented, resulting in a poor generalization.

In consequence, while dealing with imbalanced datasets, a standard accuracy would be a biased metric for measuring the classification goodness; thus, weighted precision, weighted recall, and weighted f1 scores would be better indicators of the classifier performance. Also, a confusion matrix would provide a visual representation of the classification accuracy between the predicted versus the actual classes.

A confusion matrix is the most basic form of accuracy assessment while solving classification tasks. It provides us how many predicted classes were accurately and/or inaccurately outputted when compared to the actual classes. A confusion matrix for a binary classification task could be expressed as shown on Table 2, from which several classification metrics such as precision, recall, accuracy, and f1 score can be computed.

Table 2 Confusion matrix structure for a binary classification problem

Predicted/Actual Class	**Positive Class**	**Negative Class**
Positive Class	True Positive	False Positive
Negative Class	False Negative	True Negative

Precision represents the fraction of the correctly identified positive classes from all the predicted positive classes as follow:

$$Precision = \frac{True\ Positive}{(True\ Positive + False\ Positive)} \tag{23}$$

Recall, in the other hand, represent a measure of the correctly identified positive cases from all the actual positive cases as:

$$Recall = \frac{True\ Positive}{(True\ Positive + False\ Negative)} \tag{24}$$

Accuracy is the measure of all the correctly identified cases and is used normally while working with balanced datasets.

$$Accuracy = \frac{True\ Positive + True\ Negative}{\begin{array}{c}(True\ Positive + False\ Positive + True\ Neagtive \\ +False\ Negative)\end{array}} \tag{25}$$

F1 score is represents the harmonic mean between precision and recall and gives a better measure of the incorrectly classified cases than the accuracy metric.

$$F1\ score = \frac{2(precision\ \cdot\ recall)}{precision + recall} \tag{26}$$

Finally, to summarize we could say that different metrics could be used according to the purpose and the nature of the dataset. For instance, accuracy is a good choice when the true positive and true negative are highly important, while f1-score must be chosen when false negative and false positive are crucial. In addition, for imbalanced datasets, even though a standard accuracy might not be the best performance metric, it could be weighted by the number of instances belonging to each class to account for class imbalance; this would provide a more reliable performance metric if accuracy is used to asses a certain model performance.

Chapter 3

3. DATA ANALYSISI AND PROCESSING

3.1 EXPLORATORY DATA ANALYSIS

Exploratory data analysis is the process throughout which we study and attempt to find useful information and existent patterns within the data. The major purpose is to understand the nature of the data itself and establish initial potential methodologies or approaches for solving the lithofacies classification problems.

Furthermore, by the proper recognition of the relationship, and correlation between data readings, a new machine-learning based imputation technique will be subsequently proposed and discussed in Section 3 as a feature augmentation methodology in order to improve the final classification performance.

3.1.1 Exploring Lithofacies Labels

The datasets present 12 different lithofacies classes in their majority dominated by shale, shaly lithologies, and sandstone. Table 3 shows each lithofacie description and its presence percentage in the training, open test, and hidden test subsets.

Table 3 Lithofacies presence percentages summary.

			Lithology Presence Percentage (%)		
Lithofacie	**Label**	**Code**	**Training**	**Open Test**	**Hidden Test**
Sandstone	SS	0	14.40	17.60	11.50
Shaly sandstone	S-S	1	12.90	12.80	10.00
Shale	SH	2	61.60	61.40	58.70
Marl	MR	3	2.80	2.40	3.60
Dolomite	DOL	4	0.10	0.30	0.20
Limestone	LIM	5	4.80	3.50	3.80
Chalk	CH	6	0.90	0.50	2.40
Halite	HAL	7	0.70	-	5.30
Anhydrite	AN	8	0.10	0.10	0.50
Tuff	TF	9	1.30	0.90	0.80
Coal	CO	10	0.30	0.50	0.20
Basement	BS	11	0.01	-	-

Along with Table 3, the bar plot presented on Figure 14 also reflects more clearly the lithology distributions present on each data subset. As visible, there is a great class imbalance between different lithologies, fact that may have an important role during the classification. Besides, it is worth to mention that there is a great presence of lithology types that could be described as a mineral mixture, fact that might also have an important relevance while attempting to properly classify similar lithology classes as they are expected to have similar petrophysical property readings.

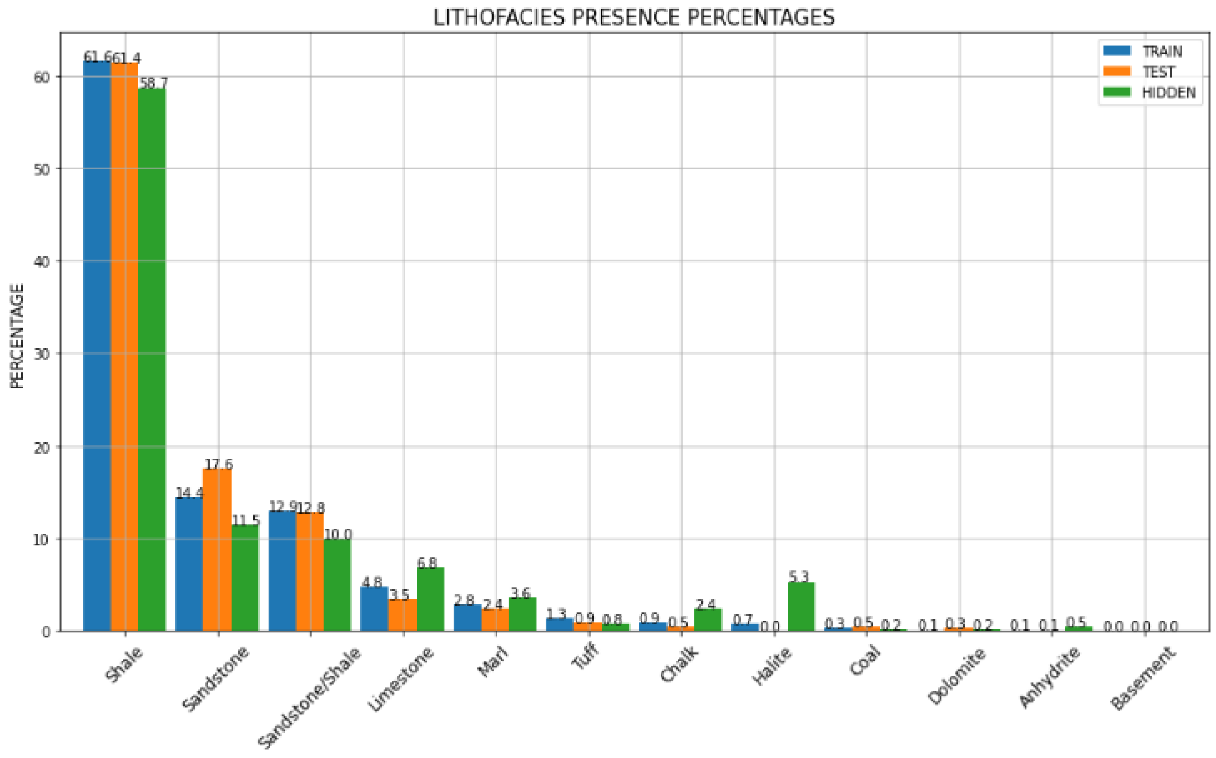


Figure 14 Lithofacies presence percentage distributions

Furthermore, based on Figure 14 and from a geological perspective, we can simply infer that the North Sea geology is widely dominated by shaly, sandy sediments, and carbonates mainly deposited during the Jurassic, Cretaceous, and Cenozoic ages.

This is not surprising given the geological evolution of the North Sea; which was initially characterized by an extensive marine transgression extended along the complete North Sea during the transition from the Triassic into the Jurassic. Subsequently, extensive deltaic systems containing sand, shale, and coal were developed during the late Jurassic in the northern North Sea and the Horda Platform once the marine transgression ended (See Figure

15). Besides, similar deltaic systems were developed during the same age along the Danish Basin and the Stord Basin. This sediment depositions accompanied by the major Jurassic rifting phase leaded to faulting and the formation of the most important source rocks for the hydrocarbon reservoirs located in the North Sea (NPD, 2015).

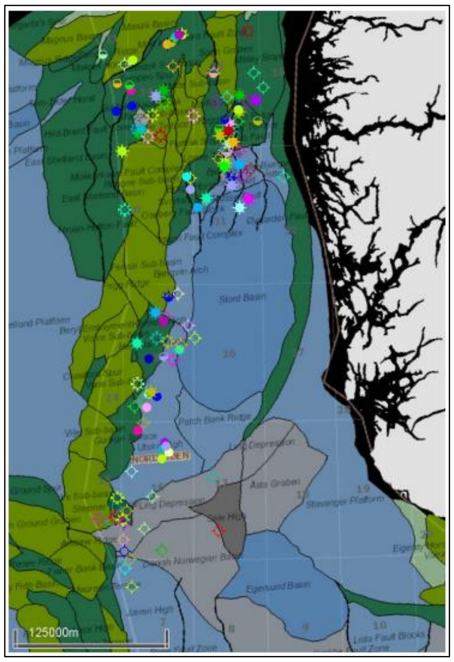


Figure 15 Wells geological location (NPD, 2021)

Following, the rifting phase ceased in the Early Cretaceous and the deposition two contrasting lithologies took place, chalk at the southern North Sea and siliclastic, clay-dominated sediment in the northern zone. Finally, chalk deposition took place and finished in the Early followed by a thermal subsidence that leaded into the creation of the intracratonic sedimentary basin of the North Sea as the continents moved to their current location; consequently, due to the basin margins uplift, submarine fans were transported from the Shetland Platform towards the east. Finally, several deltaic systems running from the Shetland Platform towards the east were formed and characterized the central North Sea, these correspond to the vast majority of the hydrocarbon reservoirs present in the North Sea (NPD, 2015).

3.1.2 Exploring Features

As stated previously, exploratory data analysis is a highly important step in any data science workflow due to its implications while understanding the data contents, extents, connections, and variations. The current datasets contain a wide group of characteristics available to be used as input data, normally known as features or predictors. These potential features involve 20 different types of log readings and 6 additional metadata characteristics describing well names, interpretation confidence, location, and lihtostratigraphical information (See Table 1). Unfortunately, as in many real world problems the dataset present incompleteness or sparsity in some predictor that might have been caused different reason such as cost considerations, borehole problems, logging tool failure, telemetry issues, or simply they were omitted by choice.

The following figures were designed to better visualize the logs data and metadata presence per well on the datasets before undergoing into the supervised-learning implementation. Figure 16 shows that from the 98 wells held in the training set most of the missing data in the training set relies on the SGR, DCAL, ROPA, RMIC, MUDWEIGHT, and DTS logs, same which are present in only 13, 22, 25, 27, 28, and 32 wells, respectively. Further, the open and hidden test sets (Figure 16) behave similarly in regard of data presence, where most of the missing data once again relies on the same well logs previously mentioned with minor differences between each testing data subsets.

Moreover, checking the statistics summary in order to have a feature values overview is essential to identify possible abnormal values that might be outside of the physical boundaries and may affect the classification performance. However, understanding the data based merely on numerical values lacks of meaning; thus, box plots of the most important features from a petrophysical point of view are displayed in Figure 17.

The gamma ray log (Figure 17a) shows that there are some values that exceed the physical boundaries, which normally go from zero to 300 or 350 API units in most of the offshore reservoirs. In addition, the lithology distributions for the mixed-based lithofacies such as sandstones, shaly-sandstones, and shales overlap between each other. This could probably indicate that some readings corresponding to these classes in the dataset may have been misinterpreted or mislabeled, or it could also be an inherent property of the formations due to presence of some radioactive minerals such as k-feldspar, zircon or mica. These radioactive minerals could raise sandy lithology readings over 150 API units similarly to shale.

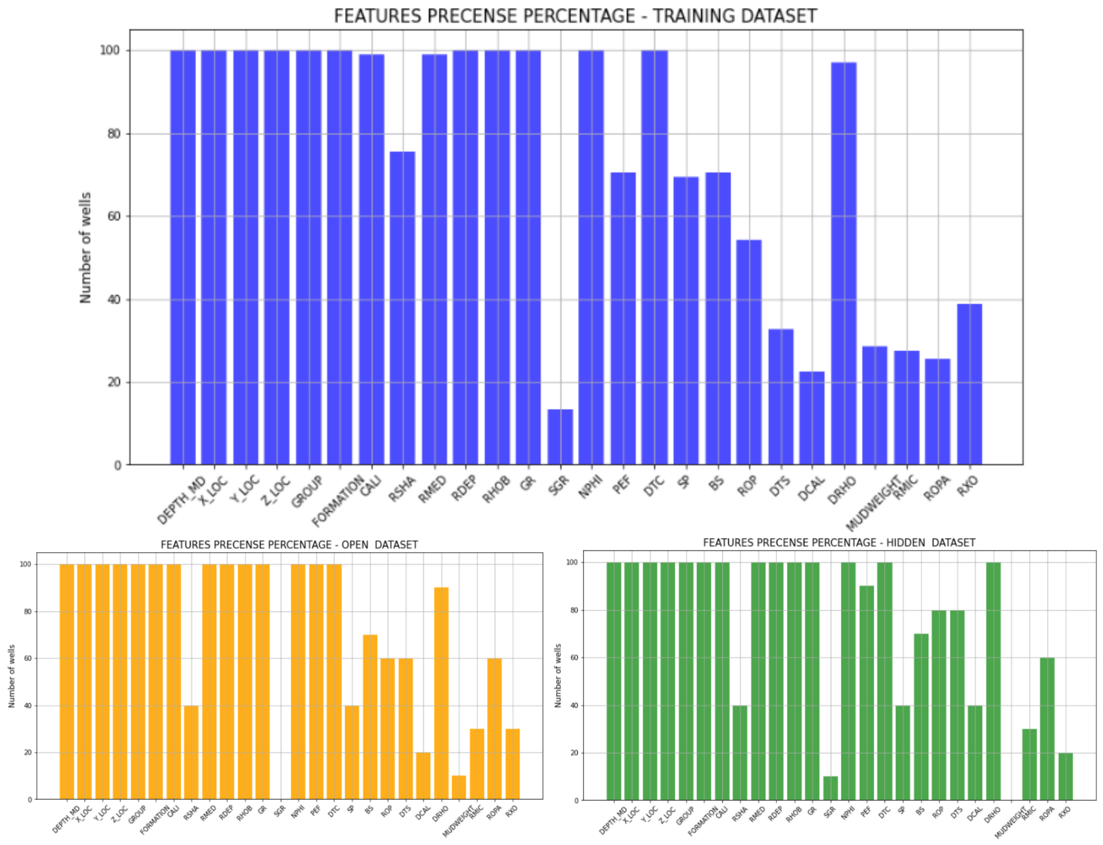


Figure 16 Feature Presence per well – Training Set (upper center), Open Test Set (lower left) Hidden Test Set (lower right)

Further, the spontaneous potential log SP (Figure 17b), normally used to identify gross lithology and differentiate between permeable and non-permeable formations, as expected shows a quite defined shale baseline reading and little deflections to higher and lower values based on the formation permeability and fluid content salinity. Thus, the values exceeding the whiskers of the SP log data mainly happen in sandstones, shaly-sands, and shale correspond to the fluid content, which might be formation water or hydrocarbons.

Besides, the neutron porosity log (Figure 17c), which is normally combined with the bulk density log for practicality, shows a shale trend line around 20-35% NPHI, while for other lithologies the NPHI rely around the expected values, almost cero for anhydrite, and between 10 to 45% for sandstone, dolomite and limestone.

In addition, the NPHI values that exceed the whisker values are presented mostly in sandstones, shale, and limestone and might linked to variations in the hydrogen index caused the formations fluid content. However, the current interpretation may still be considered subjective do to the facts that the NPHI log is based on limestone units, it has been studied isolately from the other wireline logs, and the gas effect on the readings has not been considered.

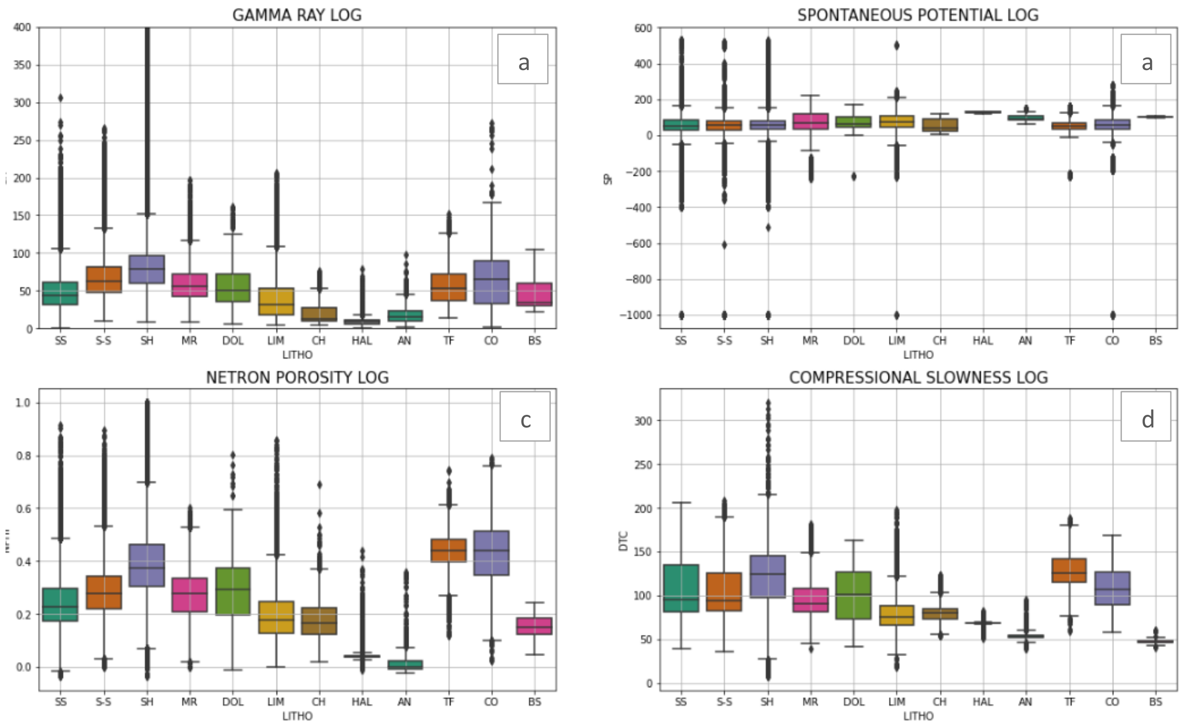


Figure 17 Wireline logs boxplots color labeled by lithology, (a) Gamma Ray, (b) Spontaneous Potential, (c) Neutron Porosity, (d) Compressional Slowness.

Finally, the compressional acoustic logs (Figure 17d) behave apparently as expected for most of the lithologies. However, there are several DTC reading identified as shale that are lower to 100 us/m which is not a normal range of reading for shaly lithologies.

On the other side, visually checking the relationship between wireline logs helps to understand the internal structure of the data and more importantly discard the predictors with high correlation in order to avoid or diminish overfitting during the training stage. For this purpose, the Spearman's Correlations between every possible combination of variables without considering the caliper and the bit size logs that normally do not have a direct connection with the lithology type is displayed on Figure 18. Besides, Figure 19 shows a scatter pair plot of the variables found to be highly correlated color-coded by lithofacie, which helps visualize and understand variation along the data.

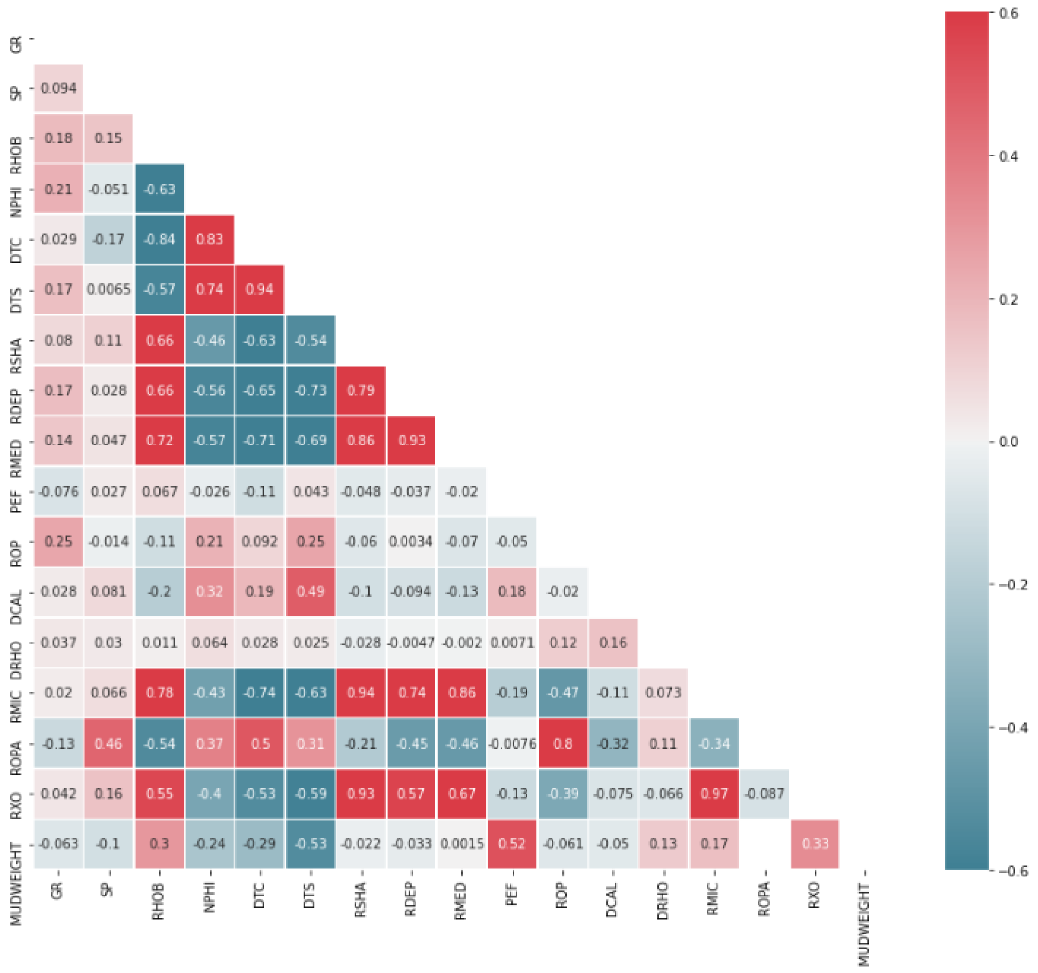


Figure 18 Spearman's correlation between wireline logs color-coded by correlation strength.

First, a great positive Spearman’s correlation of around 0.83 was found between NPHI and DTC logs, numerically exhibited on Figure 19 were the NPHI log increases as the DTC log does. This relationship is expected due the fact that compressional slowness depends on the amount solid minerals encounter in the rock media and its saturation; in other words, the less mineral material, the more porosity a rock has and for instance the higher the compressional slowness becomes. In the other hand, a negative Spearman’s correlation of -0.84 expectedly occurs between RHOB and DTC that we could explain as common relation if we consider rock compaction and fluid saturation, the higher the compaction, the higher the bulk density and the compressional wave velocity, and the lower the compressional slowness gets. Refer to Castagna's and Gassmann's researches to have extensive understanding of the effect and relationship between wave velocities and other rock-fluid properties.

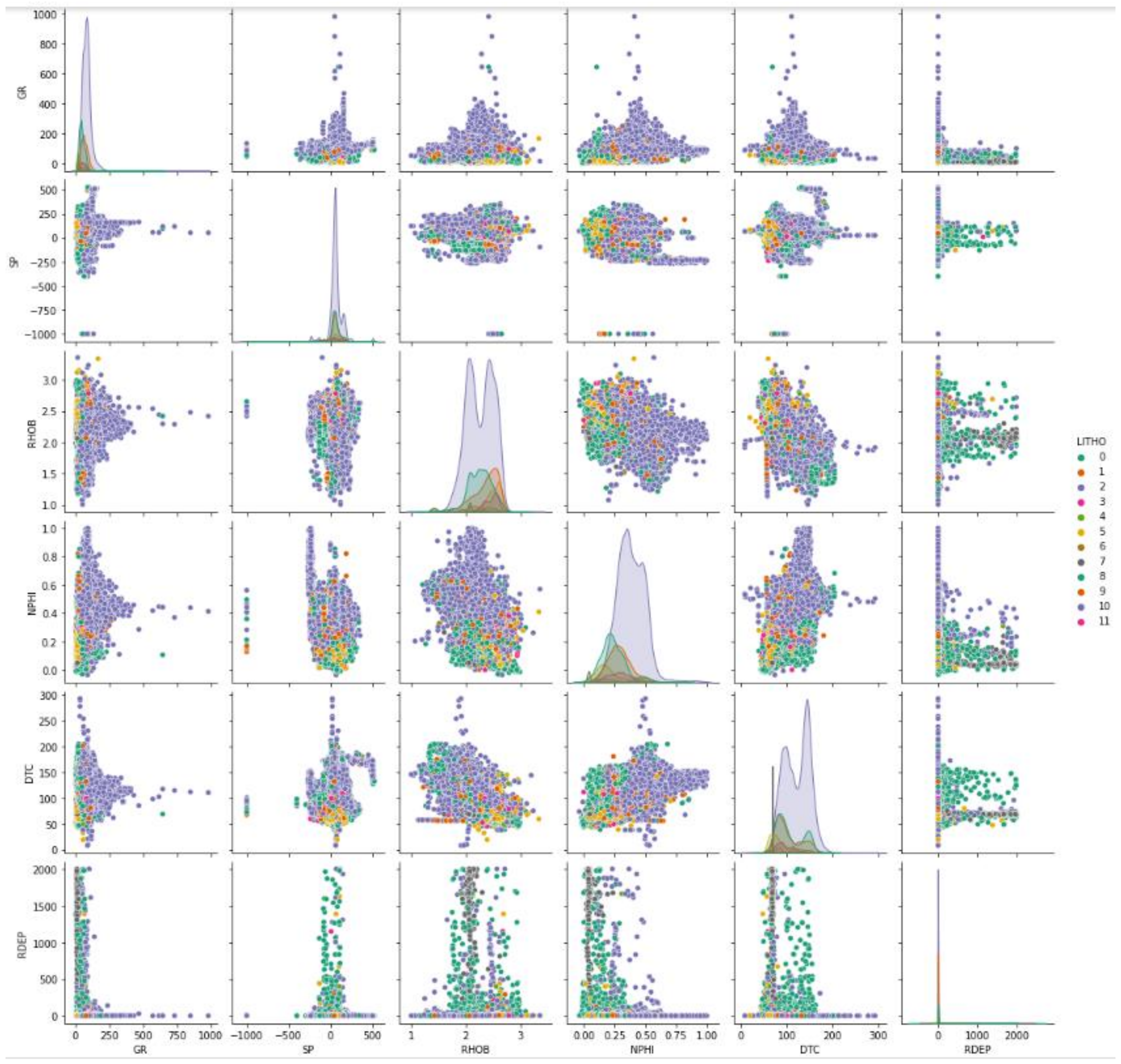


Figure 19 Bivariate correlation between most relevant logs for lithology identification, distributions color labeled by lithology shown on the diagonal.

Second, most of the resistivity logs RMIC, RSHE, RMED, RDEP, RXO present high correlations; however, the most dramatic ones were encounter between RMED, RDEP, and RSHA and between RMIC, RSHA and RXO as numerically exhibited on Figure 18. These strong correlations between variables might bring problems into some machine-learning models' performance or might reduce scalability and increase the running time a particular model requires to accomplish its task due to the increase in dimensionality we get by keeping correlated variables. We will further investigate the informativeness of these variables in Section 4 in order to perform a wise-driven feature selection for every machine-learning model being analyzed so that their performances do not get affected in a high extent if the less informative and highly correlated predictors are removed prior to start the training process.

Third, besides the existing linear correlation between the previously mentioned features, some other pair of variables did not show any apparent relationship at all. This occurs principally while plotting the photoelectric factor, gamma ray, and spontaneous potential logs against the other variables. Consequently, this apparently complex relationship between data and overlap in the readings for different lithologies make highly difficult to identify lithofacies based just on one or two wireline logs independently from the others. This is in general the reason while petrophysicists have always been in the need to use different log combinations in order to identify lithofacies in a proper manner, but also here is where machine learning plays an important role in order to understand and predict continuous or categorical values based on complex pre-existing patterns and relationships within the data.

Lastly, Figure 19 also displays on the diagonal the distributions for each wireline log; at first glance some variables appear to be more normally distributed than others, some distributions are slightly skewed towards the majority classes logs reading, and some others even present bimodal distributions as in the case of DTC, NPHI, and RHOB. This might be a problem while trying to find an optimal classification solution, especially while implementing distance-based and gradient descent-based machine-learning algorithms, which in the best-case scenario may still converge but in a quite slow manner considering that the distance between data instances and the learning rate are highly determined by the magnitude of the variables involved in the task. Consequently, in the incoming subsection we will attempt to prevent possible issues regarding data distributions and magnitudes through the implementation and evaluation of how different normalization techniques may impact the global lithofacies classification performance.

3.2 DATA PREPARATION

Even though most data-related projects follow a common process with regard of data preparation and processing, both are the most crucial and time demanding stages while deploying a machine or deep learning algorithm. In fact, to a certain degree the results and success of their applications depends principally upon them, as it is well known, the quality of the algorithms output depends strictly on the quality of the data used as input.

As consequence of the above mentioned, there is a huge need to accurately address this stages in order to help our data-driven project succeed. Moreover, reproducing consistent methodologies that can first handle and treat data accurately before developing appropriate and applicable machine-learning tools is the main inspiration for the current and related projects.

In consequence, considering that missing data from well logs is a common problem in subsurface and may have a great impact while predicting lithofacies classes, this subsection will mainly explore and test a machine-learning-based missing data imputation technique as well as a feature generation process, which aim to improve the quality and reduce sparcity on the datasets before entering the classification task.

3.2.1 Standard Data Imputation, Normalization, and Outlier Removal

The initial approach was to complete the emptiness existing in the original datasets by a standard and simple technique called median imputation. Since most of the techniques we analyze along this study are distance-based and gradient descent-based algorithms, it becomes imperative to normalize the datasets inasmuch as the magnitude of the variables might affect the size of the gradient descent step and the distance between instances that will be used to find an optimal solution. Consequently, three of the most frequently used data normalization techniques were implemented and tested on the datasets imputed by the median beforehand.

Moreover, prior to implement and test the different normalization techniques, the categorical variables present in the data such as the lithostratigraphic group and formation were label encoded by using a cat encoding functionality and the resistivity logs were log-scaled in order to equalize their magnitude to the neighbor variables' scales. Besides, in order avoid any kind of data leakage every scaling technique were implemented by fitting different type of scikit-learn scalers on the training data and then transforming the open and hidden test sets into similar scales. Figure 20 displays some of the wireline logs before applying any sort of scaling, and after applying a min-max scaler, an standard-scaler, and a normalizer.

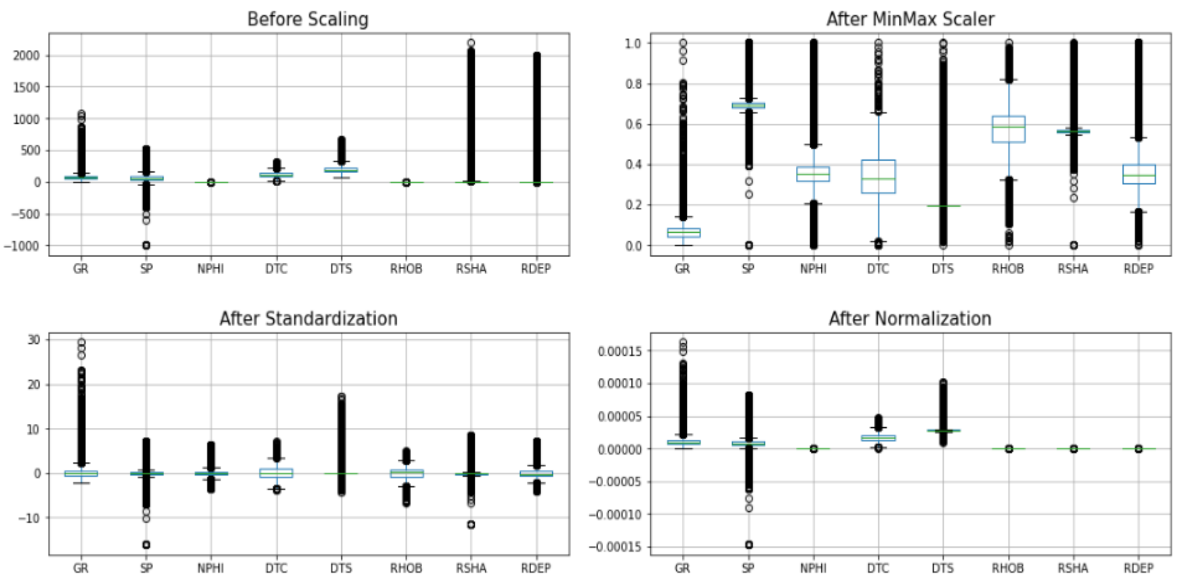


Figure 20 Different normalization techniques applied on the training dataset: Before scaling (upper-left), Min-max scaled data (upper-right), Standardized data (lower-left), (d) Normalized data (lower-left)

Accordingly, being not able to visually select the most suitable scaling method for our datasets, a logistic regression classifier was trained on a 10% stratified subsample of each differently scaled dataset by only using 23 out of the 28 original features and subsequently tested on the open test set. The 'SRG', 'ROPA', 'RXO', 'MUDWEIGHT, and 'LITHO_CONF' columns were removed for the three datasets before training basing our judgment principally on their missing data percentages.

The results shown on Table 4 demonstrate that by standardizing our data we achieved a greater classification performance of almost 8% when compared to the other implemented techniques such as min-max scaling and a normalization.

Moreover, even though standardization provided better results compared with normalization and max-min scaler methods, it also became more expensive in terms of running time and number of epochs needed to make the logistic regression model converge as described on Table 4. In addition, as visible on Figure 20, the standardized training data seemed affected by possible outliers and unrealistic readings, especially in the case of the GR and SP log where the outliers are quite visible.

Table 4 Different data normalization techniques tested on a logistic regression base model

NORMALIZATION METHODS - Base model: Logistic Regression			
Normalization Method	Test accuracy (%)	Number of iterations to converge	Time to converge [sec]
Without Normalization	61.4	-	No convergence
Max-Min Scaler	61.0	18	11
Standardization	**69.7**	**1126**	**256**
Normalization	61.4	25	16

Subsequently, in order perform outlier elimination, the same 10% stratified subsample used for testing the normalization techniques composed by 117050 instances was used for testing four different automatic outlier elimination methodologies available on the open source scikit-learn python library. Besides, the current training set subsample-based outlier elimination approach was taken due to the massive size of the original training set, which made of testing each method on the complete set a computationally expensive task.

First, the standard deviation outlier identification methodology needed seven standard deviations away from the mean to keep a reasonable number of instances for each lithofacie, specifically for the tuff, coal, and basement, which hold the most extreme GR and SP readings in the datasets. Second, the tree-based outlier detection known as isolation forest needed to establish a contamination parameter equal to 0.01 in order to keep a similar class distribution to the original training set similarly to the fist methodology. Third, a local outlier detection method was also tested by using different contamination fraction, where the highest test performance was achieved by using a contamination factor of 0.01. Fourth, a one class support vector machines outlier identification method was tested with different outlier fractions as well achieving the highest classification performance with a contamination fraction of 0.01.

As visible on Table 5, the accuracies obtained after applying each outlier elimination technique do not affect widely the logistic LR classification performance. However, the local outlier factor method LOF seemed to remove more efficiently the most isolated values based on their neighbor instances without worsening the classification performance; however, LOF offers a great disadvantage by becoming highly expensive while handling big datasets as in our case.

Table 5 Outlier elimination methods tested on a logistic regression base model

OTLIER ELIMINATION METHODS - Base model: Logistic Regression				
Normalization Method	Test accuracy (%)	N° iterations to converge	Time to converge [sec]	N° outliers removed
Standardized Data (no outliers removed)	69.70	319	78	-
Standard Deviation	69.29	17	4	3190
Isolation Forest	69.63	310	78	1171
Local Outlier Factor	**69.72**	**94**	**29**	**1171**
One-Class SVM	69.46	46	12	1171

Figure 21 presents a histogram of the removed instances by LOF, where most of the removed values belong to the most frequent classes corresponding to sandstone, shaly-sand, shale, marl, and limestone. Figure 21 also presents the boxplots of the 10% training set subsample prior to outlier removal and after applying LOF, where the main difference lies on the GR, DTC, RSHA, and SP logs.

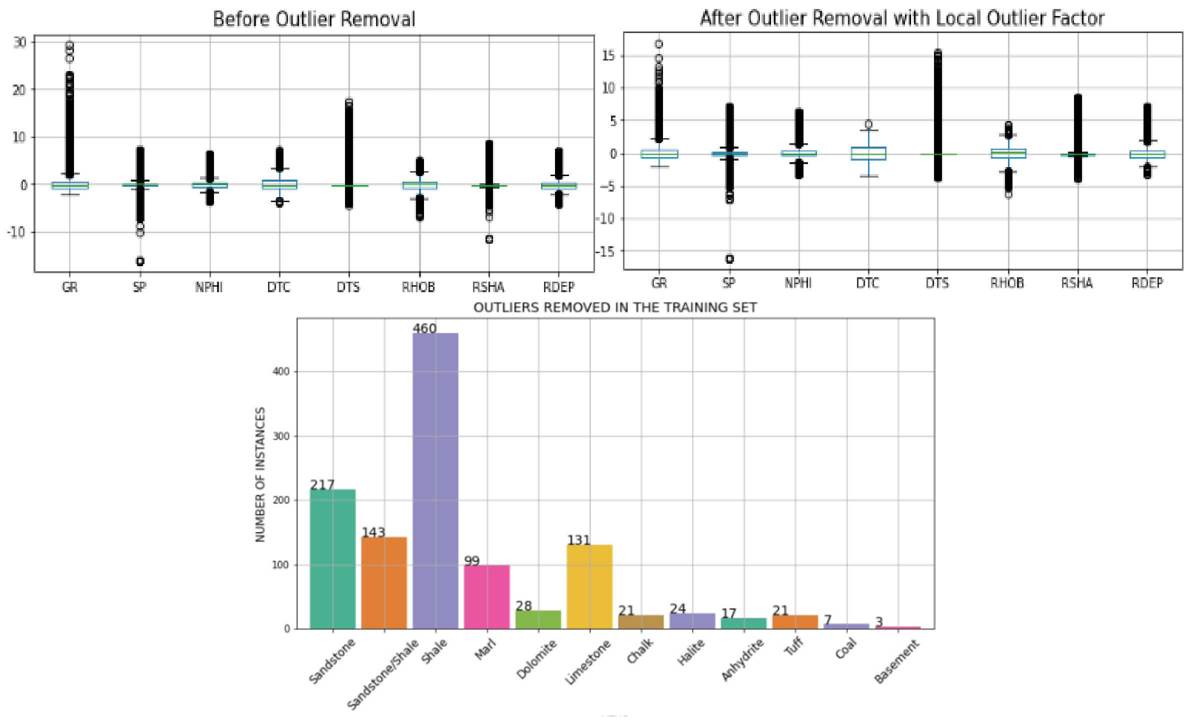


Figure 21 10% training data subsample boxplot before outlier removal (upper-left), 10% training data subsample boxplot after LOF outlier removal (upper-right), Subsample removed outliers' counts by lithofacie.

Finally, based on the previous analysis and regardless of the expensiveness LOF demands, it was applied to the complete training dataset removing a total number of instances equal to 10856, which compared to the initial number of instances held by the original training dataset, represents barely 1%.

3.2.2 Machine-learning-based data augmentation

Integrating well log data into seismic data is a core process to characterize reservoirs, process that becomes challenging if the available well log data presents missing sections. This issue has been profoundly investigated in the past years by using different techniques that include linear interpolation, local-based mean imputation, numerical rock models, and empirical relationships. For instance, even though the Gardner's and Castagna's empirical correlations may provide reasonable sonic-density and compressional-shear sonic relationships respectively; in most cases they do not provide a detailed relationship between such properties. In fact, empirical correlations and numerical rock models might tend to be sensitive to beforehand assumptions taken without considering the structural complexities and stratigraphic variations along the subsurface.

In this context, the FORCE datasets exposure offers an opportunity to approach this issue in a much more statistical-automated manner through the implementation of machine learning algorithms. Consequently, the present section presents a predictive, sequential, and multi-stage imputation approach to overcome the missing data issue as an attempt to optimize the final lithofacies classification task. This methodology is summarized on

Figure 22 and will be explained along the present section.

Firstly, a quick feature importance ranking is developed in order to understand which features play the most relevant role or contribution in the classification accuracy. This leaded along with petrophysical experience leaded us to identify that the most relevant features while classifying lithofacies by either machine learning and manual interpretation are the GR, NPHI, RHOB, DTS, and DTC logs.

```
ML-BASED FEATURE IMPUTATION ALGORITHM
Pre-requisites:
    1. feature_ranking: most relevant features ranked by missing data percentage from high
       to low.
    2. models: every possible machine-learning regressor to be evaluated against the others.

Input: training_set, test_set

for target_feature in feature_ranking:
    ** "Splitting training set on features and target"**
    features_i  = all variables other than target_feature
    target_i     = target _feature
    training_set_i = all training instances where target_i is present
    test_set_i = all test instances where target_i is present

    for model_i in models do:
        ** "Training and evaluating each machine-learning model"**
        fit the model to training_set_i
        predict target on test_set_i
    end
    ** "Imputing missing data before moving into the next target"**
    compare models' performances and best model section
    predict and impute missing instances of the target_feature on training_set and test_set
end

Output: Machine-learning feature imputed training_set and test_set
```

Figure 22 Machine-learning-based feature imputation algorithm

Second, based on the previous analysis we developed a prediction priority ranking for the five selected logs based on their missing data percentages in order to minimize the prediction error by using as much data as possible for training purposes in each case. In other words, we seek to sequentially predict each wireline log according completeness the other features have for training and prediction purposes, so the learning machines could get much more information from the other less sparse predictors.

Table 6 Feature prediction priority ranking to follow for ML imputation

Prediction Priority Ranking	Feature (Log)	Missing Data Percentage
1	DTS	85.1 %
2	NPHI	34.6 %
3	RHOB	13.8 %
4	DTC	6.9 %
5	GR	0.0 %

Third, based on the feature prediction priority ranking three ensemble regressors were used to train and test their performance on the open test set based on the root mean squared error (RMSE), mean absolute error (MAE), explained variance (EV), maximum error (ME) and R-squared factor (R^2). It is important to note that, as described on

Figure 22, the current methodology is a training-prediction multi-stage process where before entering each training-prediction substage for a particular target feature, the training dataset obtained during the previous training-prediction substage is splited into two smaller subsets for training and validation purposes. Afterwards, once each training process at each substage is completed, the regressors are tested on the open test set in order to select the best performing ML algorithm to finally update the datasets by imputing the missing values implementing the best performing ML regressor at each substage. This process aims to keep the actual reading for the treated features and only use machine learning to impute the missing values encountered along the mentioned variables.

The first prediction substage aims to predict the shear acoustic log, where Table 7 presents prediction results obtained by the evaluated regressors. The extreme gradient boosting XGB regressor appeared to achieve the highest performance when compared to light LGBM and categorical CAT gradient boosting algorithms. Further, even though XGB performed better, it faced difficulties to predict DTS values beyond 400 us/m, while the final ML imputed DTS distribution shown on Figure 23 seemed to be highly influenced by the predicted DTS values. This effect could be attributed to the amount of missing data the actual DTS log has, which involves almost 85.1% of the data instances from which we could expect to have many more shale and sand related readings given the North Sea geology nature in which the majoritarian lithologies are essentially those.

Table 7 Shear slowness DTS prediction results - Prediction substage 1

PREDICTION SUBSTAGE 1 – DTS PREDICTION							
Log	Model	Data	EV	ME	RMSE	MAE	R^2
DTS	XGBoost	Training	0.943	320.764	17.016	9.933	0.943
		Testing	**0.915**	**154.864**	**16.155**	**10.545**	**0.935**
	CatBoost	Training	0.949	288.091	16.098	9.393	0.949
		Testing	0.896	164.39	20.341	12.138	0.896
	LightBoost	Training	0.966	222.80	13.085	7.550	0.966
		Testing	0.917	170.59	18.142	11.347	0.917

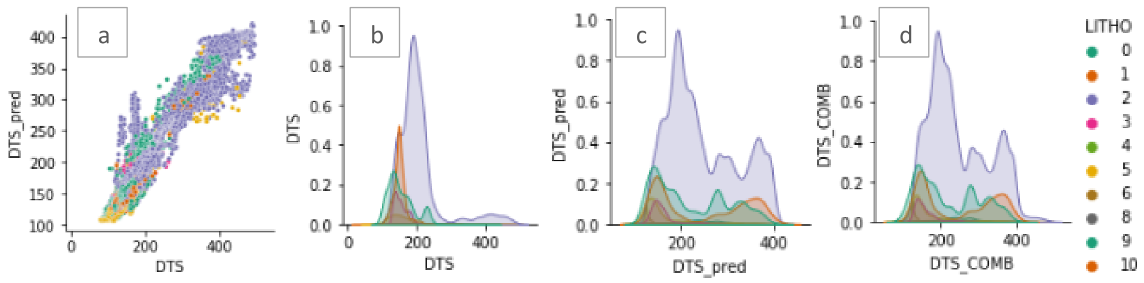


Figure 23 (a) Actual DTS vs. predicted DTS, (b) Actual DTS probability distributions by lithology, (c) Predicted DTS probability distributions by lithology, (d) Final ML imputed DTS probability distributions by lithology.

Once the DTS readings on the datasets are updated, the second prediction substage attempts to predict the neutron porosity NPHI missing values. Table 8 presents the metrics of the evaluated regressors used for predicting NPHI, where the LGBM performed slightly better than the other regressors on the training and test sets.

Table 8 Neutron Porosity NPHI prediction results - Prediction substage 2

PREDICTION SUBSTAGE 2 – NPHI PREDICTION							
Log	Model	Data	EV	ME	RMSE	MAE	R^2
NPHI	XGBoost	Training	0.822	0.598	0.055	0.039	0.823
		Testing	0.795	0.458	0.054	0.041	0.795
	CatBoost	Training	0.811	0.583	0.057	0.041	0.812
		Testing	0.789	0.473	0.055	0.041	0.789
	LightBoost	Training	0.857	0.568	0.049	0.035	0.857
		Testing	**0.803**	**0.486**	**0.053**	**0.039**	**0.802**

As visible on Figure 24, LGBM appeared to face difficulties to predict neutron porosity values above 0.6. Besides, even though the predicted NPHI distribution (Figure 24c) seemed to resemble the actual NPHI distribution (Figure 24a), the model seems to overestimate sandstones' porosities to values higher to 0.40, same which became less noticeable after imputing the predicted values into the missing readings.

Figure 24d depicts how the final distribution became more alike to the initial NPHI distribution after ML-imputation where slight overestimations may still be visible only for sandstones.

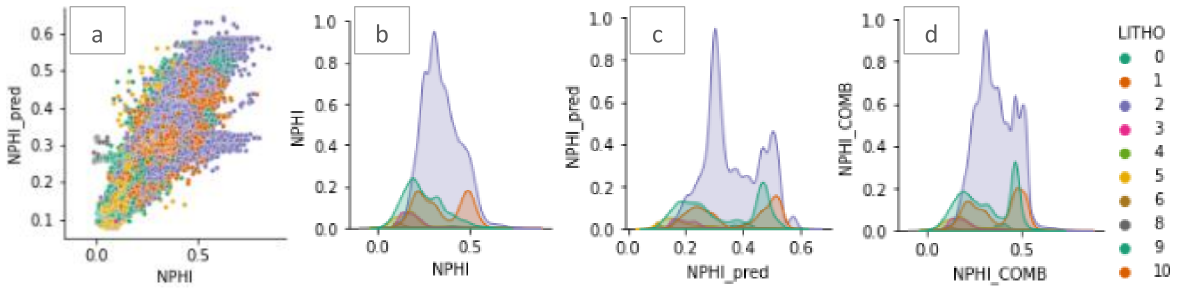


Figure 24 (a) Actual NPHI vs. predicted NPHI, (b) Actual NPHI probability distributions by lithology, (c) Predicted NPHI probability distributions by lithology, (d) Final ML imputed NPHI probability distributions by lithology.

Then, the third prediction substage after DTS and NPHI imputation attempts to predict missing bulk density values. Table 9 Bulk Density RHOB prediction results - Prediction substage 3presents the metrics for the RHOB prediction, where the categorical gradient boosting regressor seemed to predict NPHI with more confidence. In addition, Figure 25 demonstrate how similar the actual, the predicted, and the final ML-imputed NPHI distributions are, and hence how confident its prediction is.

Table 9 Bulk Density RHOB prediction results - Prediction substage 3

PREDICTION SUBSTAGE 3 – RHOB PREDICTION							
Log	Model	Data	EV	ME	RMSE	MAE	R^2
RHOB	XGBoost	Training	0.898	1.277	0.081	0.054	0.897
		Testing	0.854	1.063	0.930	0.063	0.854
	CatBoost	Training	0.938	1.26	0.629	0.042	0.938
		Testing	**0.871**	**0.973**	**0.087**	**0.060**	**0.871**
	LightBoost	Training	0.927	1.370	0.068	0.046	0.927
		Testing	0.866	0.958	0.089	0.060	0.865

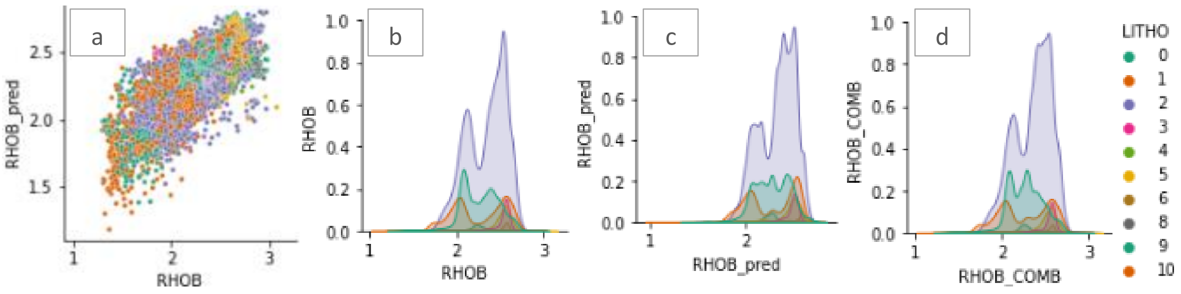


Figure 25 (a) Actual RHOB vs. predicted RHOB, (b) Actual RHOB probability distributions by lithology, (c) Predicted RHOB probability distributions by lithology, (d) Final ML imputed RHOB probability distributions by lithology.

Lastly, the fourth prediction substage involved the prediction of the compressional sonic DTC missing instances. Table 10 displays the metrics obtained while predicting DTC, where XGBoost regressor outperformed the other two algorithms on the open test dataset.

Table 10 Compressional Slowness DTC prediction results - Prediction substage 4

PREDICTION SUBSTAGE 4 – DTC PREDICTION							
Log	Model	Data	EV	ME	RMSE	MAE	R^2
DTC	XGBoost	Training	0.977	150.226	4.451	2.828	0.977
		Testing	**0.974**	**47.8123**	**4.422**	**3.172**	**0.974**
	CatBoost	Training	0.988	138.046	3.168	2.007	0.988
		Testing	0.975	49.690	4.263	3.056	0.975
	LightBoost	Training	0.986	98.843	3.493	2.249	0.986
		Testing	0.973	53.435	4.439	3.010	0.973

Figure 26 displays the correlation between the actual and predicted DTC, which seemed to have the same ranges, meaning that XGBoost was able to predict this property with high confidence as described by the regression metrics on Table 10. In addition, the confidence while predicting DTC can be observed on the similitude between the actual, predicted, and final ML-imputed DTC distributions.

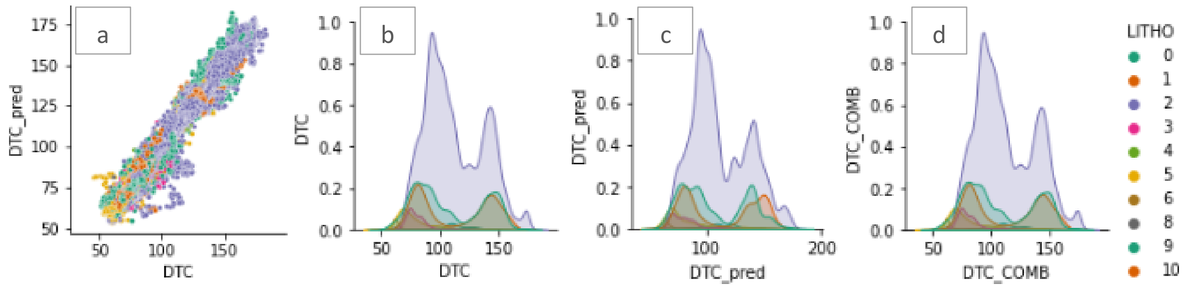


Figure 26 (a) Actual DTC vs. predicted DTC, (b) Actual DTC probability distributions by lithology, (c) Predicted DTC probability distributions by lithology, (d) Final ML imputed DTC probability distributions by lithology.

It is important to mention that the present ascendant-ranked feature imputation methodology based on target features presence percentages was selected against a descendent methodology inasmuch as the error for each predicted log increased importantly when the second method was tried out.

Finally, Figure 27 shows the actual, predicted, and machine learning imputed logs for well 35/9-8 corresponding to the test dataset. As visible, even though this well contains complete readings for the four treated logs, it serves to visualize and compare how similar the actual logs are in comparison to the predicted ones. In fact, based on the explained variance and

R^2 factors we could say that there is much more confidence while predicting the compressional slowness (DTC), shear slowness (DTS), and density (RHOB) logs than while predicting the neutron porosity (NPHI) log, meaning that much more of the variance held by target variable could be explained by the independent variables used during each training substage.

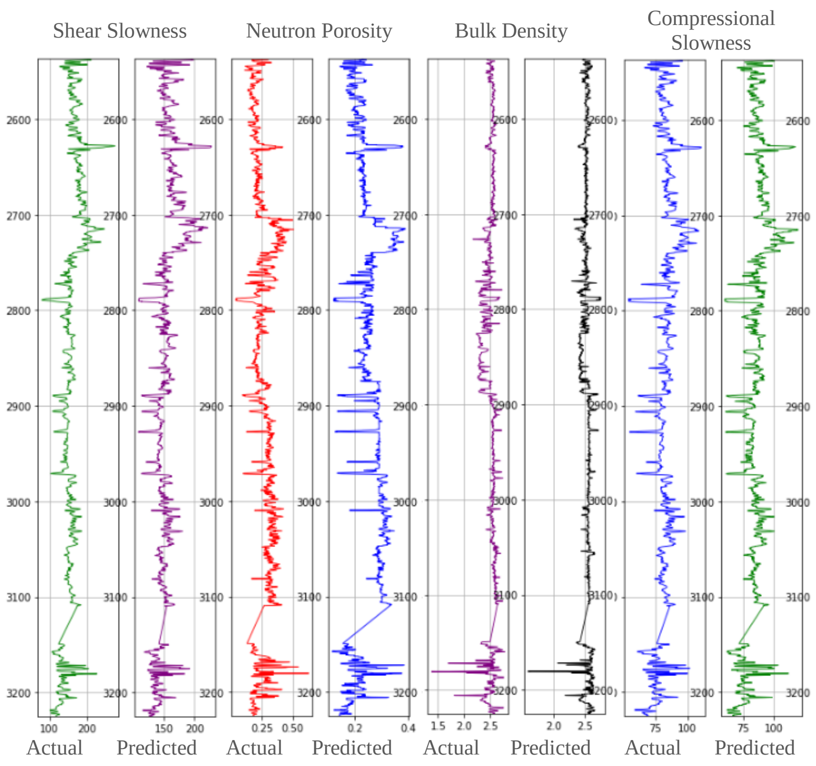


Figure 27 Actual and predicted DTS, NPHI, RHOB, and DTC logs (well 35/9-8).

In the other hand, well 34/5-1-S presented on Figure 28 shows how the highlighted small to medium size missing data gaps were effectively filled up by the most accurate machine learning model's predictions on each treated wireline log.

To conclude, the presented missing data imputation methodology was designed and adopted with the purpose of improving progressively the datasets quality and consequently the final classification performance. It should be noted this methodology attempt also to minimize as much as possible the prediction uncertainty, which might be mainly introduced while predicting extensive missing data gaps, by predicting each well log sequentially based on the data available to train each regressor during every training-prediction substage. Moreover, it is worth mentioning that the present methodology could be applied to any extent in order to predict any feature included into the datasets; however, due to timing and computational resources constrains, it was only applied to the four most relevant wireline logs.

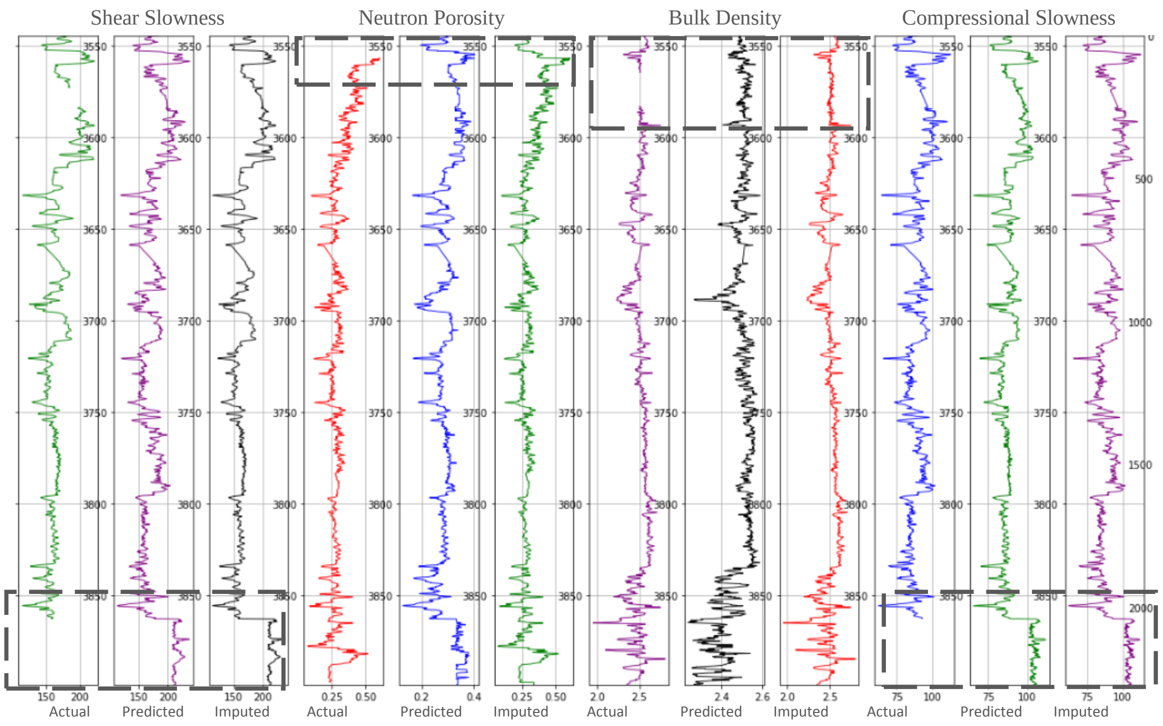


Figure 28 Actual and predicted DTS, NPHI, RHOB, and DTC logs (well 34/5-1S).

3.2.3 Feature Engineering

Furthermore, apart from the 23 initial pre-selected features during the data normalization analysis, four of which were imputed and improved during the augmentation section, seven more features were designed and included into the original datasets to be used as part of the training, validation, and prediction stages. These additional features are enlisted on Table 11.

Table 11 Additional features incorporated into datasets

N°	Feature Name	Key
1	Cluster by location	Cluster
2	Bulk Modulus	K
3	Shear Modulus	GM
4	Measured-vertical depth ratio	MD_TVD
5	Slowness Ratio	DT_R
6	Shear Impedance	AI
7	Compressional impedance	AI_P

Six out of the seven engineered features were computed straightforwardly based on the augmented wireline logs. However, deciding the optimal number of clusters to which to split up the dataset based on well location was a big question at first while implementing unsupervised learning, this leaded us to try to determine the number of clusters based on the elbow method, which in brief calculates the sum of the squared distances of each data point to the near cluster center, known as inertia, by using different number of clusters. The elbow plot on Figure 29 shows that three clusters was be optimal for our data and adding more clusters becomes marginal or useless.

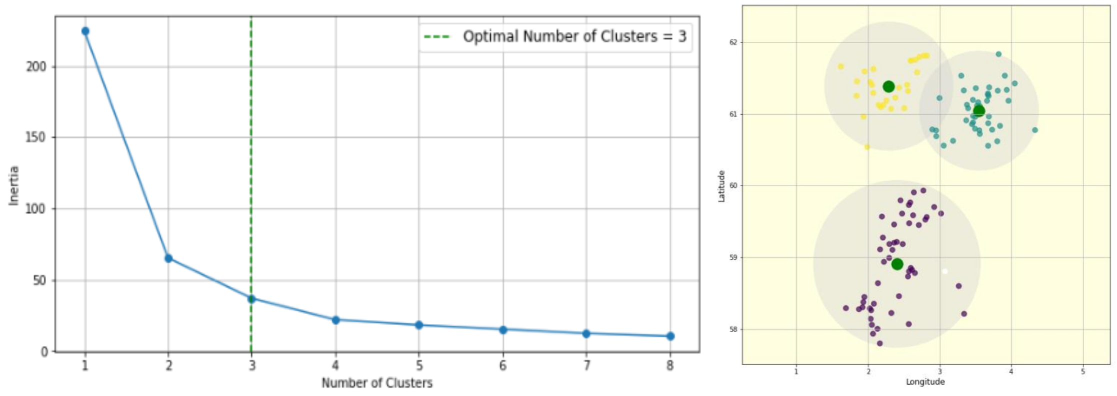


Figure 29 Optimal number of clusters based on elbow method (left), Clusters visualization (right)

Finally, Table 12 records how the previously analyzed logistic regression classifier's performance improved after machine-learning feature augmentation and features engineering were executed in comparison to the results obtained when median-imputed data was used for training. Along with this, based on the best standardization and outlier removal techniques that were found in previous analyses, the machine-learning imputed data was similarly treated in regards of this by implementing a standardization and a local outlier elimination techniques prior to enter the training and prediction stages.

Table 12 Logistic regression model's performance by using median-imputed data, machine learning-imputed data, and after including additional features.

LOGISTIC REGRESSION (Standardized data)				
Number of Features	Test accuracy (%)	N° iterations to converge	Time to converge [sec]	Features Comments
23	**69.7**	1126	256	Median-imputed
23	**70.7**	1127	255	+ ML Augmentation
30	**71.3**	1103	299	+ Additional Features

To conclude, it is important to note that the most significant performance improvements were obtained after the machine-learning feature augmentation process rather than from feature engineering. Nonetheless, even though the improvements might not appear highly significant while using a linear classifier, these might become higher after eliminating the non-informative features, carrying out hyper-parameter tuning, and by implementing more robust classifier types, factors that have not been addressed yet and will in the subsequent sections.

Chapter 4

4. LITHOLOGY PREDICTION BY MACHINE AND DEEP LEARNING

In this section, we initially explored the baseline construction philosophy and its importance to monitor model performance. Each baseline model was built and validated by implementing a cross validation technique on 10 stratified K-Folds of the training set. This technique splits the training dataset in 10 subsampled and tests every model on each of them, ensuring that each data subset keeps the same lithology class distributions that the original training set holds in order to generalize the performance and avoid a bias towards the most frequent classes.

Subsequently, considering the massive nature of the training dataset, the hyper-parameter tuning process for the most expensive models was executed in a smaller stratified subsample of the original set in order to reduce running time and save computational power. Refer to Appendix E where all the experimental process is extensively documented.

Furthermore, in the face of the efficient performance improvements previously seen on the logistic regression baseline model presented on Section 4 after data processing, the original readings on the training, open, and hidden datasets were replaced and complemented by the values obtained after ML feature augmentation, feature engineering, standardization, and outlier removal treatment, so that the other model could also experience a similar performance enhancement from this procedures. Refer Appendix B to see python code of all the functionalities needed to process the datasets prior to start the machine learning implementation and Appendix A to visualize the python code for every optimized model once the hyper-parameter and feature selection stages described in the current section, have been completed.

4.1 BASELINE MODEL OVERVIEW

Several baseline models were created and tested on 10 different stratified K-Folds of the training set as a cross validation technique. As shown on Figure 30, the top performing models while iteratively using 9 folds for training and 1 for testing seemed to be a random forest classifier. However, considering that every model was trained and tested by using only the training data with no regularizing their learning process, these results might be prone to overfitting.

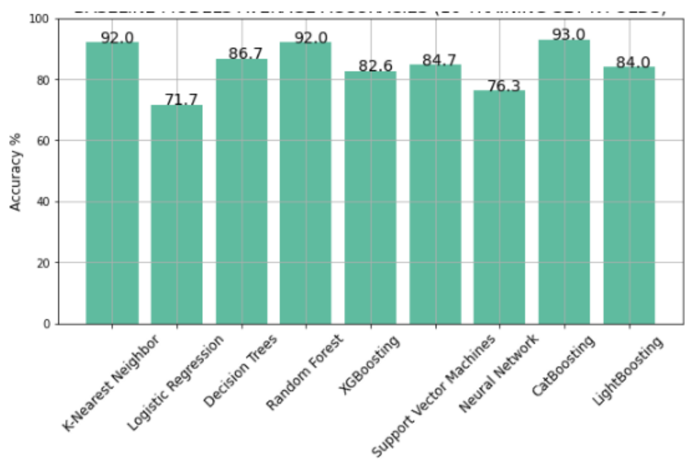


Figure 30 Base models average accuracies while iteratively training on 9 k-folds and testing on the 10th k-fold.

Therefore, each model has to be further analyzed, tuned, and then tested on the open and hidden datasets to have a consistent analysis and comparison between each other afterwards. The main objectives in order to optimize each model performance in the present section will involve an accurate hyper-parameters determination and a wise feature selection, considering that form the 30 available features for training, some may not be informative but they may incorporate noise and create confusion into the models. Table 13 presents all the processed features available for training the learning machines.

Table 13 Available Features for training the learning machines.

RDEP	DEPTH_MD	RHOB (augmented)
RMED	X_LOC	NPHI (augmented)
RSHA	Y_LOC	DTS (augmented)
RMIC	Z_LOC	Cluster (additional)
SP	BS	K (additional)
DCAL	CALI	GM (additional)
ROP	GROUP ENCODED	MD_TVD (additional)
DRHO	FORMATION ENCODED	DT_R (additional)
PEF	WELL_ENCODED	AI (additional)
GR	DTC (augmented)	AI_P (additional)

4.2 CONVENTIONAL MACHINE-LEARNING METHODS

4.2.1 Logistic Regression

In the preceding section we constructed a Logistic Regression base model, which performed with accuracies of 72, 71.3, and 73% on the training, open, and hidden sets, respectively after feature augmentation, future engineering, data standardization, outlier removal treatment, and by using the default model's hyper-parameters. Moreover, as stated previously, there is a genuine need to appropriately select the best model hyper-parameters and predictors to be used while training, validating, and testing in order to optimize the algorithm's performance.

Initially we attempted to reduce the number of the features through a recursive feature elimination process, which is normally intended to remove the least informative features that might slow down the training process, introduce noise, or create confusion into the model. This process did not provide much positive results for the current model since apparently 29 of the 30 original features seemed to be necessary to accomplish the highest accuracy on the training set as shown on Figure 31. However, the recursive feature selection process provided a better understanding on the predictors that play the most important role for the classification as shown in Figure 32a, where accordingly the first 11 features account for most of the variance of the training dataset and together accomplish an accuracy above 73%.

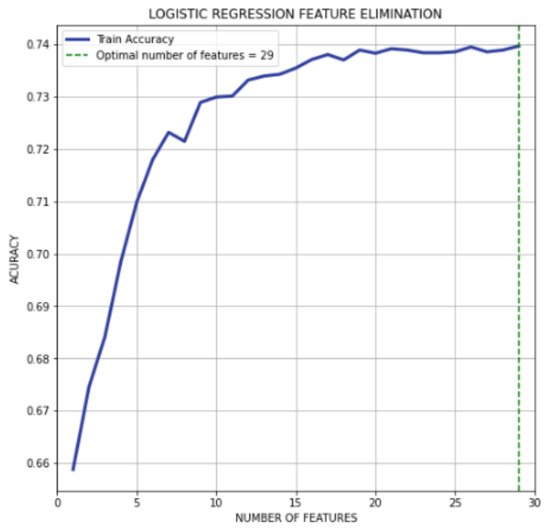


Figure 31 Logistic Regression Classifier: Recursive feature elimination by a logistic regression-based wrapper

In other words, this means that the 19 remaining features do not improve the training accuracy in more than 2% and hence could be removed without affecting largely the model performance. Besides, in order to double check our conjecture about the most influencing features, a forward sequential feature selection method was tested in order to validate these 11 features. Figure 32b confirmed that 73% of training accuracy could be achieved by keeping solely the 11 most informative predictors as we presumed.

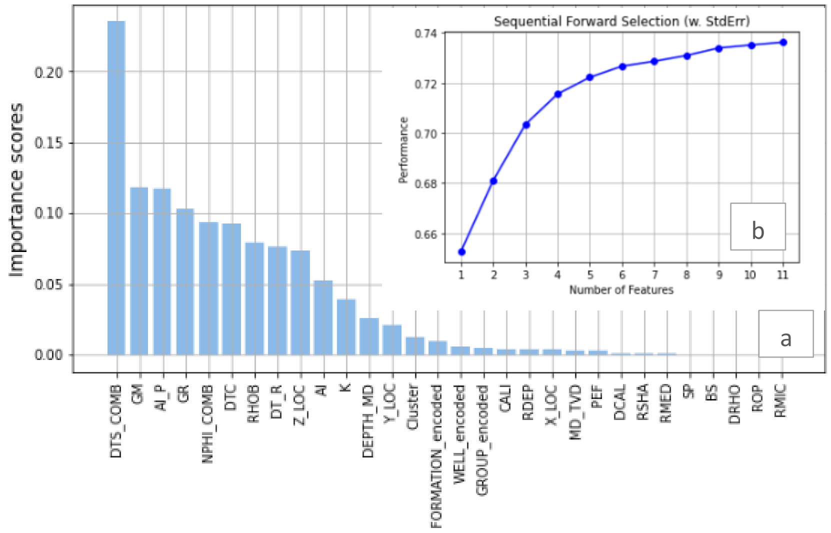


Figure 32 Logistic Regression Classifier: Permutation feature importance

Subsequently, by keeping the 11 previously selected features, a manually hyper-parameter tuning process was executed for the inverse regularization strength factor C, while the solver type and the maximum number of iteration where theoretically selected due to initial problems to make the model converge while using the default hyper-parameter values.

In addition, since any tuning process become normally expensive in terms of running time while dealing with large datasets, we performed this by using only a 10% stratified subsample of the original training set, which held the same class proportions present on the original dataset in order to make the sample statistically representative for our problem.

Afterwards, based on the scikit-learn documentation, SAGA and SAG solvers offer fast convergence when dealing with large and normalized datasets. In fact, as stated by (Defazio et al., 2014), SAGA is an improved version of SAG, which offers a better theoretical convergence rate and is adaptive to any inherent strong convexity of the problem. In consequence, a SAGA solver was selected for the current classification task while keeping the number of iterations to a high value of 4000 in order to let the model converge while manually evaluating different Inverse Regularization Strength (C) values on the open test set as validations set as shown in Figure 33.

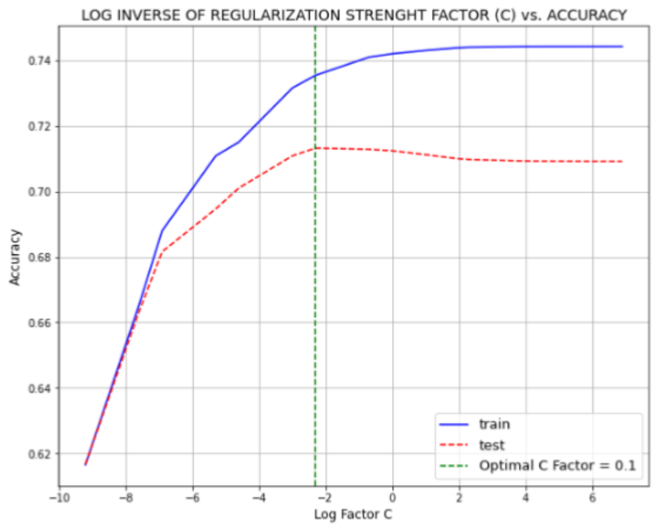


Figure 33 Logistic Regression Classifier: Different inverse regularization strength tested on the training and open test set (log C vs. accuracy)

The figure above represents how the training and validation accuracies change while the linear logistic regression model uses different inverse regularization values ranging from 10e-5 to 10e3. Note that the accuracies are plotted against the logarithm on the evaluated factor due to its investigation range; this leaded to find 0.1 as the optimal value for this parameter based on the validation accuracy. Thus, the selected optimal hyper-parameters that were implemented on the final model are summarized as follow on Table 14.

Table 14 Logistic Regression Classifier: Optimal hyper-parameters

Hyper-parameter	**Optimal value**
Inverse Regularization Parameter	0.1
Maximum Iterations Number	4000
Solver	‘saga’

To conclude, an end-model was created and trained on the 11 most informative features by using the optimal hyper-parameters previously selected. This provided accuracies of 74, 72, and 75% on the training, open test, hidden test sets, respectively. The results confirm our first guess about the non-linear relationship between features and the non-linear separation between most of the targeted lithology classes.

A class-detailed classification report for each dataset is presented on Table 15, where even though the open test set was used for fine tuning hyper-parameters, the hidden test set showed a better classification accuracy. This could be easily explained by the slight difference on the lithology distributions between the open and hidden test sets more notoriously on the limestone, marl, chalk, halite, and anhydrite lithology types. This apparently slight class distribution difference was enough to provide an extra improvement on the hidden set accuracy when compared to the open test set accuracy.

Table 15 Logistic Regression Classifier: Classification reports for the training, open test, and hidden test datasets.

LOGISTIC REGRESSION CLASSIFICATION REPORT									
Class	**Training Set**			**Open Set**			**Hidden Set**		
	Prec.	*Rec.*	*F1.*	*Prec.*	*Rec.*	*F1.*	*Prec.*	*Rec.*	*F1.*
Sandstone (0)	0.66	0.63	0.64	0.64	0.61	0.63	0.58	0.60	0.59
Sandstone/Shale (1)	0.51	0.15	0.23	0.30	0.23	0.26	0.34	0.13	0.19
Shale (2)	0.78	0.95	0.86	0.81	0.90	0.85	0.84	0.94	0.89
Marl (3)	0.44	0.15	0.23	0.15	0.01	0.02	0.22	0.18	0.19
Dolomite (4)	0.40	0.01	0.02	0.00	0.00	0.00	0.00	0.00	0.00
Limestone (5)	0.62	0.55	0.58	0.49	0.55	0.52	0.57	0.51	0.54
Chalk (6)	0.72	0.59	0.65	0.00	0.00	0.00	0.53	0.93	0.68
Halite (7)	0.98	0.99	0.98	0.00	0.00	0.00	0.99	0.96	0.97
Anhydrite (8)	0.88	0.67	0.76	0.00	0.00	0.00	0.88	0.31	0.46
Tuff (9)	0.54	0.16	0.24	0.66	0.15	0.25	0.10	0.03	0.04
Coal (10)	0.73	0.42	0.53	0.67	0.42	0.52	0.81	0.57	0.67
Basement (11)	0.96	0.22	0.36	-	-	-	-	-	-
Weighted Metric	0.71	0.74	0.70	0.68	0.72	0.69	0.71	0.75	0.72
Accuracy Score	0.74			0.72			0.75		
Matrix Score	-0.69			-0.75			-0.64		

Finally, the confusion matrixes normalized to the number of predictions per class are presented on Figure 34. In general, the logistic regression classifier showed the highest accurately while classifying shale, halite, and anhydrite; medium accuracies for limestone, tuff and coal; and the poor accuracies while handling similar composition lithologies. Besides, most of the misclassifications denote a tendency to the majority classes such as sandstone, shaly-sandstone and shale.

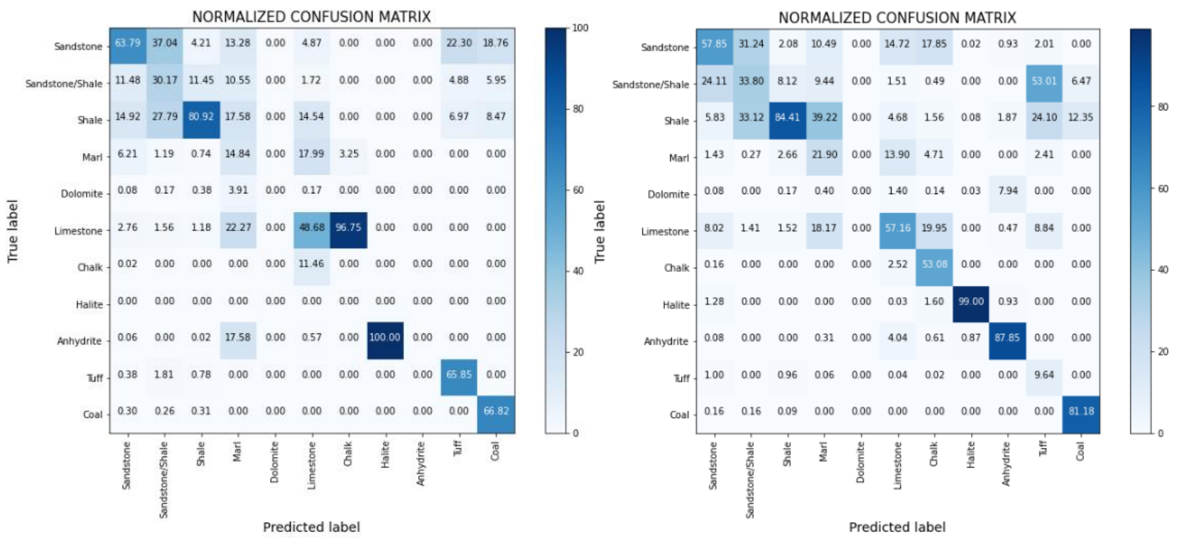


Figure 34 Logistic Regression Classifier: Classification confusion matrices for the open test set (left) and hidden test set (right).

The bias in the classification can be explained by the fact that linear separable algorithms as logistic regression are highly influenced by the target's probability distributions, which means that the minor present lithologies tend to be misclassified as any of the most frequent ones. This issue could corrected by equalizing the class distributions by any oversampling, undersampling, and synthetic sampling techniques; however, due to the extent of the current study, they were not implemented nor evaluated.

4.2.2 K-Nearest Neighbor

As previously discussed on section 4.1 the base non-parametric K-nearest neighbor model provided accuracies an accuracy about 92% when trained and tested on the training set by cross validating on 10-stratified k-folds. However, even though it showed promising results on the training data, the same did not occur when testing the base model on the open and hidden test sets, which provided classification accuracies of 72 and 74%, respectively.

In consequence, considering the high and medium-low accuracies obtained on the training and test sets respectively, a hyper-parameter optimization had to be executed to test for possible enhancements in performance. However, before undergoing into a hyper-parameter optimization, which is computationally expensive particularly when implementing KNN as previously we discussed on the theoretical background section, a feature dimensionality reduction was performed in order to be able to run the hyper-parameter in a less expensive manner.

Besides, there are plenty of model-based feature selection techniques, which may also become computational expensive when dealing with large massive datasets such as the case of recursive feature selection and permutation feature elimination. Therefore, a permutation feature selection was performed on a 10% stratified subsample of the training data in order to represent the label distribution existing on the original training dataset.

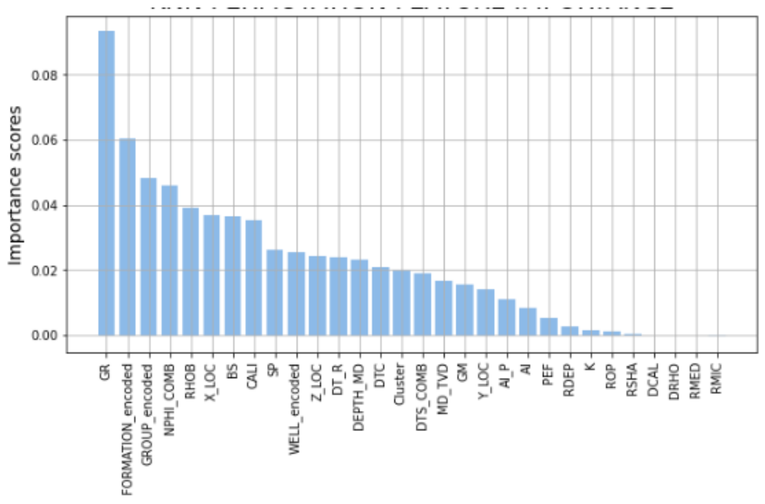


Figure 35 K-Nearest Neighbor Classifier: Permutation feature importance.

Figure 35 shows the features importance obtained by the permutation feature importance, where some predictors such as RMIC, RMED, DRHO, DCAL, RSHA, ROP, and K seem not to play a highly important role on the classification task; however, properly selecting the number of features that could provide the best results by only inspecting their importance becomes a bit difficult. In consequence, we used the open test set to measure the influence the number of features used during training has on the classification performance while keeping the same 10% stratified training subsample. Thus, based on the importance ranking provided by the permutation feature importance we trained and tested different KNN models by including sequentially one additional feature for training. Interestingly, as visible on Figure 36 the accuracy curves started to plateau while using just 6 to 10 features, and adding additional features only added slight improvement; however, the test accuracy showed a much more stable curve when more than 15 features were used.

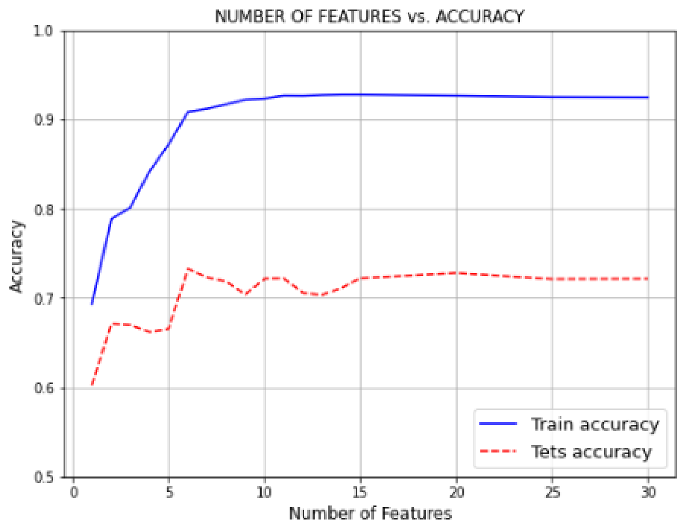


Figure 36 K-Nearest Neighbor Classifier: Impact the number of training features has on the classification accuracy.

In addition, based on the previous analysis a new default-parameter base model by only including the 15 most informative features was trained and tested on the open test and hidden test sets providing practically the same accuracies the initial KNN base model obtained while using the complete set of 30 features. In other words, by removing 15 of the less informative features the KNN classification accuracy did not get impacted while at the same time it reduced the running time KNN requires for training and predicting, and in consequence will help reducing the running time while optimizing hyper-parameters.

Moreover, a manual neighbors tuning optimization was performed in order to understand how its impact on the training and open set classification performance in order to select most optimal values that would improve model generalization. This investigation is documented on Figure 37, where we can observe that by using a number of neighbors lower than 25 the model performance on the open test set worsens while the training accuracy remains high, meaning that the model is unable to generalize well when a small number of neighbors is used.

In the other hand, by selecting a high number of neighbors, the test accuracy does not get any further improvement; thus, a number of neighbors bigger than 50 may be a good choice in order to generalized well on the unseen dataset since as shown the more number of neighbors used, the more computationally expensive the model becomes.

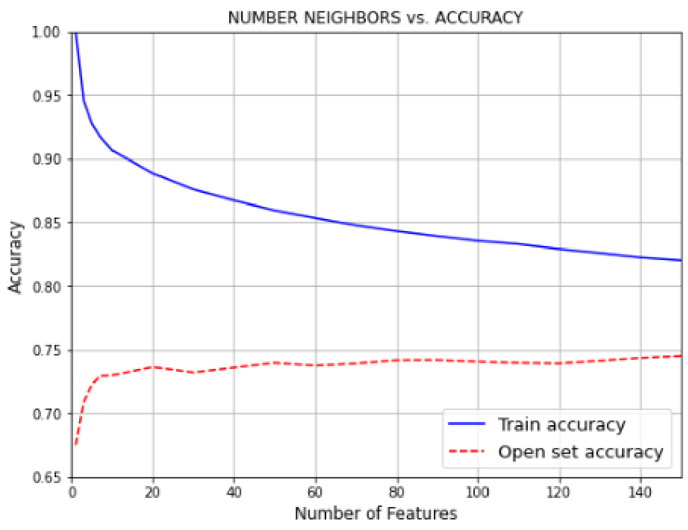


Figure 37 K-Nearest Neighbor Classifier: Number of neighbors vs. accuracy.

In addition, once the optimal number of neighbors was set on 80, a further grid hyper-parameter investigation on two additional relevant hyper-parameters such as the weights applied to each instance and the metric to compute the distance between data instances was performed. The optimal values found by the grid parameter search while implementing a 10 stratified k-fold cross validation as well as the optimal number of neighbors are summarized on Table 16.

Table 16 K-Nearest Neighbor Classifier: Optimal hyper-parameters.

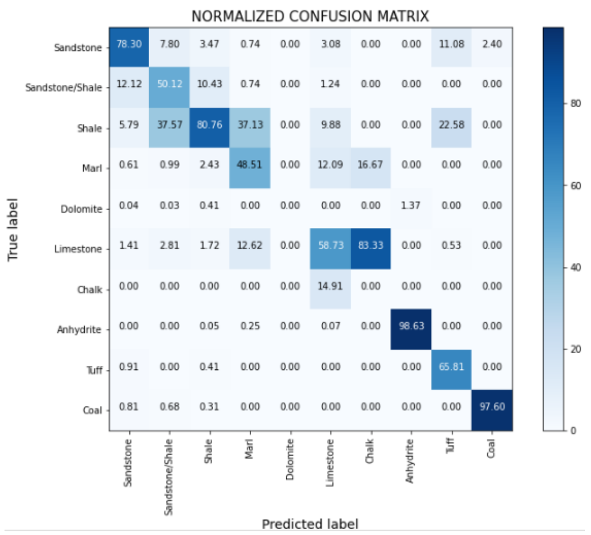
Hyper-parameter	Optimal value
Number of Neighbors	80
Weights	Manhattan
Metric	Distance

Finally, a final model based on the optimal hyper-parameters was trained and tested, providing accuracies of 78% on the open test and hidden test sets, which compared to the initial test accuracies show an important enhancement. It is important to mention that the open test set was used as validation set while finding out the optimal number of neighbors to be used; however, KNN showed consistent results when tested on unseen objects. A detailed classification report is presented on Table 17, where we could observe how KNN was able to perform consistently on both test datasets.

Table 17 K-Nearest Neighbor Classifier: Classification reports for the training, open test, and hidden test datasets.

K-NEAREST NEIGHBOR CLASSIFICATION REPORT									
Class	**Training Set**			**Open Set**			**Hidden Set**		
	Prec.	*Rec.*	*F1.*	*Prec.*	*Rec.*	*F1.*	*Prec.*	*Rec.*	*F1.*
Sandstone (0)	0.85	0.84	0.84	0.78	0.82	0.80	0.75	0.69	0.72
Sandstone/Shale (1)	0.79	0.69	0.74	0.50	0.24	0.33	0.47	0.27	0.34
Shale (2)	0.91	0.97	0.94	0.81	0.93	0.87	0.84	0.94	0.89
Marl (3)	0.85	0.73	0.79	0.49	0.06	0.11	0.66	0.29	0.40
Dolomite (4)	0.91	0.15	0.26	0.00	0.00	0.00	0.00	0.00	0.00
Limestone (5)	0.87	0.67	0.76	0.59	0.51	0.55	0.53	0.64	0.58
Chalk (6)	0.93	0.84	0.88	0.00	0.00	0.00	0.68	0.47	0.55
Halite (7)	0.97	1.00	0.98	-	-	-	0.97	1.00	0.99
Anhydrite (8)	0.92	0.82	0.87	0.99	0.58	0.73	0.93	0.49	0.64
Tuff (9)	0.84	0.86	0.85	0.66	0.50	0.57	0.51	0.46	0.49
Coal (10)	0.96	0.29	0.44	0.98	0.18	0.30	0.93	0.37	0.53
Basement (11)	1.00	0.41	0.58	-	-	-	-	-	-
Weighted Metric	0.88	0.89	0.88	0.74	0.78	0.74	0.76	0.78	0.76
Accuracy Score	0.89			0.78			0.78		
Matrix Score	-0.305			0.586			0.560		

In addition, the confusion matrices for the open test and hidden test sets are displayed on Figure 38, from which we could observe how KNN the most significant misclassifications occur between tuff and shale, chalk and marl, limestone an marl-shale, and shaly-sandstone and shale, while KNN was not even able to classify limestone on none of the test sets.



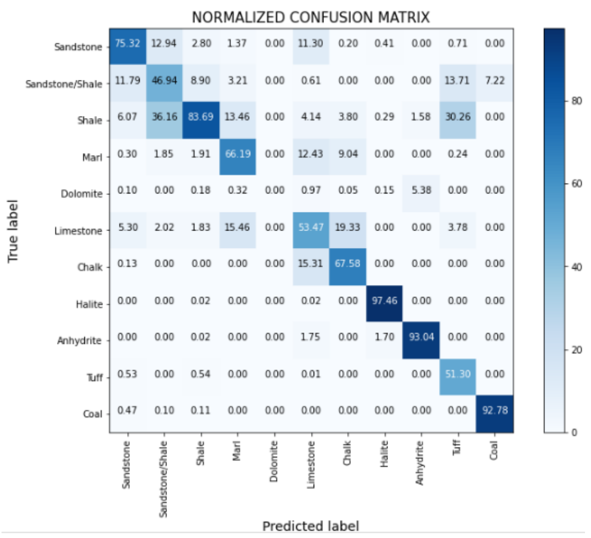


Figure 38 K-Nearest Neighbor Classifier: Classification confusion matrices for the open test set (left) and hidden test set (right).

4.2.3 Support Vector Machines

Support Vector Machines, SVM, implements separation hyper-planes to perform classification tasks. These hyper-planes achieve a good separation and the best generalization when the nearest training data point lies far from the decision plane. However, in several cases the data instances cannot be separated by a linear hyper-plane as we used while pretended while constructing our SVM base model, which provided relatively low initial classification performances of 74, 74, and 78% on the training, open test, and hidden test sets, respectively. Further, SVM requires storing the kernel matrix, which increases as the number of data instances increase, making SVM less feasible for massive datasets.

In consequence, a dimensionality reduction by implementing any model-based wrapper and a any type of grid hyper-parameter search are not suitable for SVM considering the massive number of data points contained on the training set. This leaded us to attempt to optimize manually the most crucial hyper-parameter needed for regularization purposes while only using a 10% stratified subsample of the training set that kept the class distributions in order to make it representative to the original training data. This subsample allowed to investigate the effect the regularization term C has on the SVM classification; in addition, a more expensive RBF kernel was also introduced as an attempt to translate the data into a much more complex dimension in which a much easier and accurate instance separation could be possible.

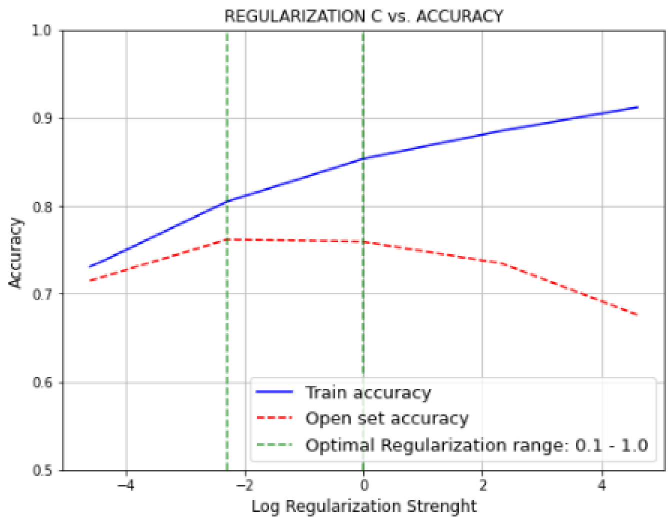


Figure 39 Support Vector Machines Classifier: Regularization vs. accuracy.

Figure 39 presents the effect C has on the training and open set accuracy, where the highest test accuracy reached a maximum value of 76% when C was equal to 0.1. It is important to note that the open test set was used to validate the hyper-parameter C in order to select the optimal value, which will be used later on the final model to predict on the hidden dataset. Moreover, due to the investigation range of the regularization term, which goes from 0.01 to 100, Figure 39 presents C in a logarithmic scale in order to be able to visualize the accuracies variability in relation to any change in C.

Based on the previous analysis, the optimal regularization term seemed to fall on between values of 0.1 and 1.0; thus, to allow a much wider variability and less penalized decision hyper-planes when testing SVM on unseen objected, an intermediate value of 0.5 was selected as optimal parameter.

Table 18 Support Vector Machines Classifier: Classification reports for the training, open test, and hidden test datasets.

SUPPORT VECTOR MACHINES CLASSIFICATION REPORT									
Class	**Training Set**			**Open Set**			**Hidden Set**		
	Prec.	*Rec.*	*F1.*	*Prec.*	*Rec.*	*F1.*	*Prec.*	*Rec.*	*F1.*
Sandstone (0)	0.78	0.80	0.79	0.80	0.78	0.79	0.71	0.73	0.72
Sandstone/Shale (1)	0.54	0.72	0.62	0.27	0.41	0.33	0.29	0.52	0.37
Shale (2)	0.96	0.87	0.91	0.91	0.82	0.86	0.95	0.84	0.89
Marl (3)	0.44	0.77	0.56	0.07	0.62	0.12	0.22	0.48	0.30
Dolomite (4)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Limestone (5)	0.63	0.78	0.70	0.52	0.45	0.49	0.65	0.57	0.61
Chalk (6)	0.70	0.89	0.78	0.00	0.00	0.00	0.46	0.71	0.56
Halite (7)	0.98	0.98	0.98	-	-	-	0.96	0.99	0.98
Anhydrite (8)	0.67	0.94	0.78	0.00	0.00	0.00	0.42	0.83	0.56
Tuff (9)	0.66	0.77	0.71	0.59	0.72	0.65	0.61	0.60	0.60
Coal (10)	0.40	0.87	0.54	0.36	0.92	0.52	0.56	0.88	0.68
Basement (11)	0.00	0.00	0.00	-	-	-	-	-	-
Weighted Metric	0.87	0.84	0.85	0.81	0.76	0.78	0.84	0.79	0.81
Accuracy Score	0.84			0.76			0.79		
Matrix Score	-0.425			-0.621			-0.536		

To conclude, a final model based on the optimal regularization hyper-parameter and a radial basis function kernel RBF was constructed, trained, and tested providing accuracies of 84, 76, and 79% on the training, open test, and hidden test sets, respectively. A class-detailed classification report is presented on Table 18 as well as the confusion matrices normalized by the total number of predictions per class.

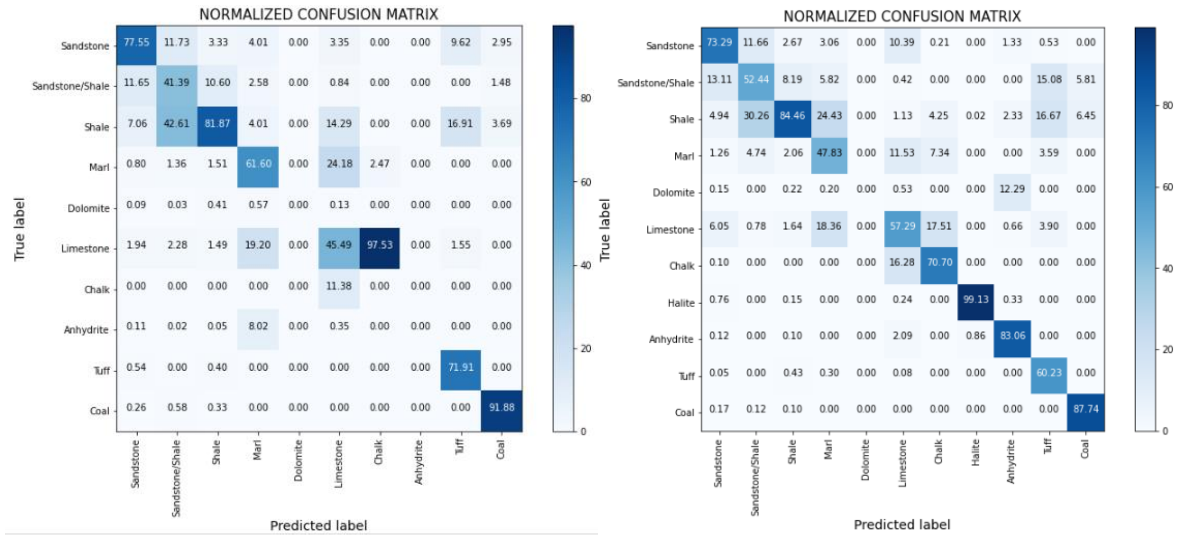


Figure 40 Support Vector Machines Classifier: Classification confusion matrices for the open test set (left) and hidden test set (right).

In general, SVM appeared not being able to distinguish between sandstones and shaly-sandstones, shale and sandstones, limestone and marl, tuff and shale, while it seemed to have high performances when classifying halite and shale. This suggests that SVM classification presents a great tendency towards the majority classes; however, encouraging the model to classify better the minority classes by weighting them through the weight parameter did not provide better results but worse. Thus, various over, under, and synthetic sampling techniques might be possible solutions to overcome SVM limitations regarding class imbalance, same which due to computational power limitation and the extent of the present study were not analyzed.

4.2.4 Decision Trees

The base model we initially constructed provided accuracies of 93, 62, and 63% for the training, open test, and hidden test sets, respectively. However, it is important to note that this accuracy was obtained only by training and testing the model on 10 stratified k-folds of the training data without manipulating any regularization term into the model. In other words, this great difference between accuracies on the training and test sets is a clear show of overfitting, which will be corrected by implementing a technique called decision tree pruning.

Cost complexity pruning is a machine-learning technique that aims to reduce the size of a decision tree by removing redundant branches that might cause overfitting in the model, so in brief it would counteract a poor model generalization. A common and suggested approach is to first decrease the maximum depth for a decision tree before undergoing into a pruning process; in consequence, we established new maximum depth equal to 15 for the base model obtaining new accuracies of 93, 62, and 61% for the training, open, and hidden sets, respectively.

Later on, the cost complexity parameter, ccp_alpha and the impurities at each level of the tree are calculated. In general, ccp_alpha influences the tree in the number of nodes a tree ends up growing. In other words, we will try to find the best ccp_alpha parameter that would restrict the tree growth up to an optimal number of nodes.

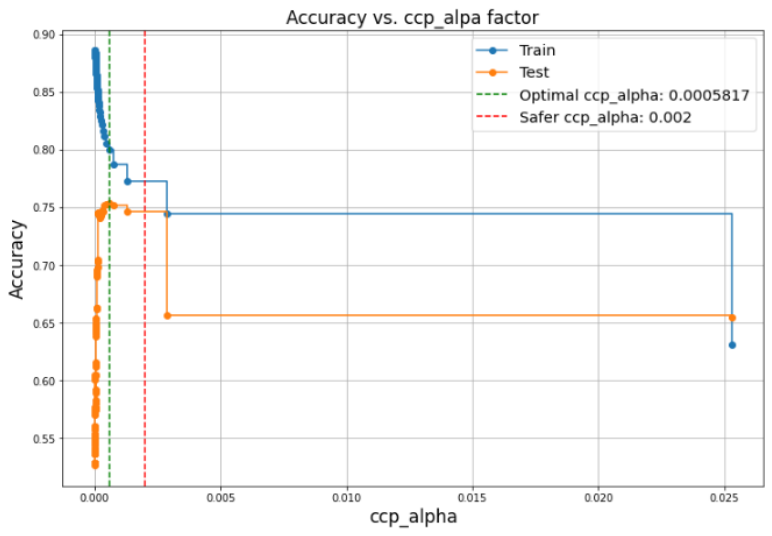


Figure 41 Decision Tree Classifier: Cost complexity factor ccp_alpha vs. accuracy on the training and open test datasets.

Figure 41 shows how the training and open test set accuracy vary accordingly to the value the ccp_alpha factor takes. The plot provides an idea that the optimal ccp_alpha factor should be in order to get the highest performance when testing on the open test set used as a validation set for the current pruning procedure. The highest performance on the test set was obtained by using a ccp_alpha equal to 0.000587; however, using this value might still be too specific in order to generalize the model performance on unseen objects. This analysis leaded us to define the optimal ccp_alpha could be any value between 0.000587 and 0.003. In consequence, with this in mind we opted for a safer ccp_alpha value of 0.002 for training and testing final model.

Table 19 provides a detailed classification performance acquired by pruned decision tree, in which we could observe that although the pruned tree provided accuracies of 76, 75, and 75% on the training, open test, and hidden test sets, respectively, it was unable to predict the least frequent classes such as chalk, halite, anhydrite, tuff, coal, and the crystalline basement.

Table 19 Decision Tree Classifier: Classification reports for the training, open test, and hidden test datasets.

DECISION TREE CLASSIFICATION REPORT									
Class	**Training Set**			**Open Set**			**Hidden Set**		
	Prec.	*Rec.*	*F1.*	*Prec.*	*Rec.*	*F1.*	*Prec.*	*Rec.*	*F1.*
Sandstone (0)	0.68	0.67	0.67	0.79	0.72	0.75	0.60	0.61	0.60
Sandstone/Shale (1)	0.75	0.21	0.33	0.50	0.02	0.03	0.78	0.11	0.19
Shale (2)	0.77	0.96	0.86	0.74	0.99	0.85	0.77	0.98	0.86
Marl (3)	0.64	0.18	0.28	0.74	0.02	0.04	0.48	0.03	0.06
Dolomite (4)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Limestone (5)	0.80	0.47	0.59	0.63	0.29	0.40	0.57	0.56	0.56
Chalk (6)	0.79	0.63	0.70	0.00	0.00	0.00	0.00	0.00	0.00
Halite (7)	0.77	1.00	0.87	0.00	0.00	0.00	0.87	1.00	0.93
Anhydrite (8)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Tuff (9)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Coal (10)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Basement (11)	0.00	0.00	0.00	-	-	-	-	-	-
Weighted Metric	0.74	0.76	0.72	0.70	0.75	0.67	0.70	0.75	0.69
Accuracy Score	0.76			0.75			0.75		
Matrix Score	-0.663			-0.690			0.665		

In addition, the confusion matrices normalized by the total number of predictions per class is presented on Figure 42.

The classification report and the confusion matrices revealed that the decision trees model was unable to predict classes such as coal, tuff, chalk, and dolomite. The imbalance on the prediction might be explained by the cost complexity pruning process, which is a great technique to raise the overall model accuracy at the cost of not capturing in detail least represented classes into the datasets. In other words, the cost complexity pruning process improved the classifier's performance from 62% to 75% on the test sets mainly by improving significantly the classification on the most frequent classes but without improving the classification for the minority classes.

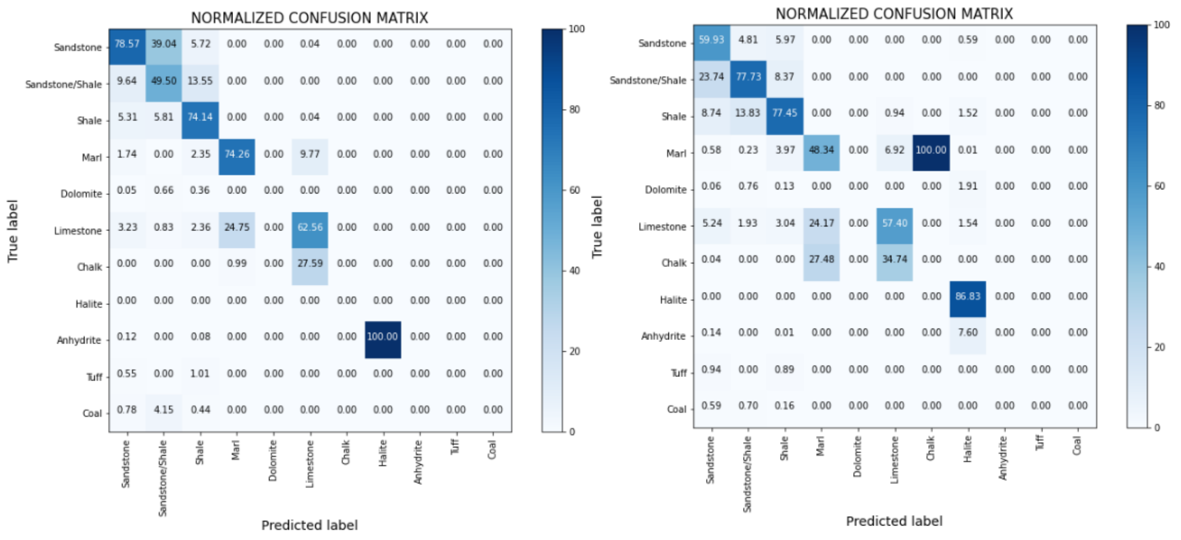


Figure 42 Decision Tree Classifier: Classification confusion matrices for the open test set (left) and hidden test set (right).

4.3 ENSEMBLE MACHINE-LEARNING METHODS

4.3.1 Random Forest

Beforehand we constructed a Random Forest base model that performed with an accuracy of 92% while training and validating the model on 10 stratified k-folds on the preprocessed training set. This provided accuracies of 78% and 79% on the open test and hidden test sets, respectively. In addition, considering the base model did not consider any regularization technique, the current section aimed to optimize the base model by performing an accurate features and hyper-parameters selection.

4.3.1.1 Recursive Feature Elimination

Initially in order to improve the model performance a feature dimension reduction was attempted to remove the least informative features that might slow down the training process, introduce noise, or create confusion into the model. To do so a recursive feature elimination wrapper was constructed and tested on a 10% stratified subsample of the training data in order to train the model on a representative sample, handle the imbalance between lithology classes, and avoid overfitting. Besides, it is important to mention that this approach was adopted since random forest classifier available on the scikit-learn library only supports CPU but not GPUs.

```
Optimal number of features: 27
```

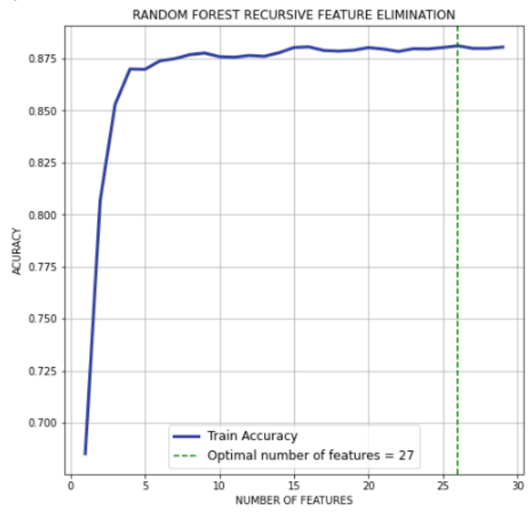


Figure 43 Random Forest Classifier: Recursive feature elimination wrapper results

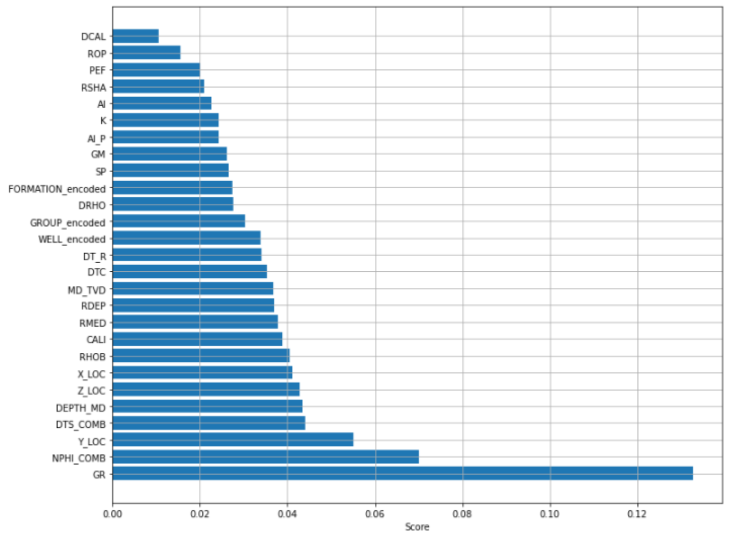


Figure 44 Random Forest Classifier: Feature importance given by the RFE wrapper.

The recursive feature selection, documented on Figure 43, indicated 27 as the optimal number of features in order to attain the highest training accuracy; however, we can also appreciate how most of the accuracy is achieved by only the initial 10 features and the 17 subsequent only contribute a slight improvement in the accuracy. In addition, RFE wrapper also provided the apparent features importance (Figure 44), in which GR, NPHI, DTS, RHOB, and some metadata features related to instances location seemed to influence the random forest output the most.

4.3.1.2 Hyper-parameter Tuning

Subsequently, after we decided to reduce the number of features up to 27 in order to look for the maximum possible accuracy, a hyper-parameter optimization process was performed based on a randomized parameter search technique. The parameters' evaluation ranges are enlisted on**¡Error! No se encuentra el origen de la referencia.**.

Table 20 Random Forest Classifier: Hyper-parameter ranges defined for tuning

Hyper-parameter	**Value ranges**
n_estimators	[from 100 to 500 in steps of 50]
max_features	['sqrt', 'auto']
max_depth	[form 1 to 50 in steps of 2]
bootstrap	[True, False]

The hyper-parameter grid search was executed for 25 iterations while cross validating the training with 10 stratified folds in order to avoid overfitting the training data, the better hyper-parameters are enlisted on Table 21.

Table 21 Random Forest Classifier: Optimal Hyper-parameter

Hyper-parameter	**Optimal value**
n_estimators	350
max_features	'sqrt'
max_depth	45
bootstrap	False

Lastly, a new model was trained by using the 27 most informative predictors (See Figure 44) and the optimal hyper-parameters. This final model provided accuracies of 98, 78, and 80% on the training, open test, and hidden test, respectively. The detailed classification reports by class can be visualized on Table 22.

Additionally, in order to help visualize the classification results Random Forest obtained, the normalized confusion matrices are displayed on Figure 45.

Table 22 Random Forest Classifier: Classification reports for the training, open test, and hidden test datasets

RANDOM FOREST CLASSIFICATION REPORT									
Class	**Training Set**			**Open Set**			**Hidden Set**		
	Prec.	*Rec.*	*F1.*	*Prec.*	*Rec.*	*F1.*	*Prec.*	*Rec.*	*F1.*
Sandstone (0)	0.97	0.97	0.97	0.79	0.85	0.82	0.72	0.80	0.76
Sandstone/Shale (1)	0.96	0.95	0.95	0.50	0.27	0.35	0.57	0.26	0.36
Shale (2)	0.98	0.99	0.99	0.83	0.92	0.87	0.85	0.96	0.90
Marl (3)	0.97	0.96	0.97	0.48	0.01	0.03	0.43	0.25	0.31
Dolomite (4)	0.95	0.77	0.85	0.00	0.00	0.00	0.00	0.00	0.00
Limestone (5)	0.97	0.91	0.94	0.42	0.57	0.48	0.60	0.61	0.60
Chalk (6)	0.99	0.99	0.99	0.00	0.00	0.00	0.56	0.34	0.42
Halite (7)	1.00	1.00	1.00	0.00	0.00	0.00	0.99	0.98	0.99
Anhydrite (8)	0.97	0.98	0.98	0.81	0.41	0.55	0.72	0.76	0.74
Tuff (9)	0.98	0.98	0.98	0.84	0.53	0.65	0.66	0.26	0.38
Coal (10)	0.95	0.88	0.91	0.79	0.85	0.82	0.86	0.64	0.74
Basement (11)	1.00	1.00	1.00	-	-	-	-	-	-
Weighted Metric	0.98	0.98	0.98	0.75	0.78	0.75	0.77	0.80	0.78
Accuracy Score	0.96			0.78			0.80		
Matrix Score	-0.061			-0.582			-0.497		

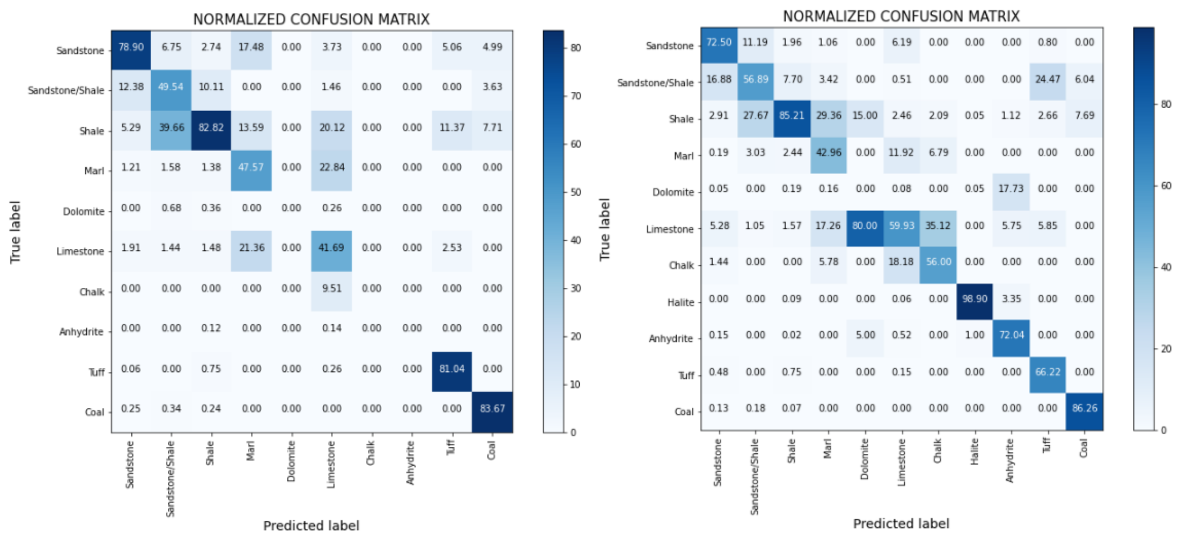


Figure 45 Random Forest Classifier: Classification confusion matrices for the open test set (left) and hidden test set (right).

Finally, from both the classification reports and the confusion matrices we could observe how the random forest perform quite well while predicting classes that have no conflict with others; however, when it comes to similar lithologies it is prone to make many more mistakes as the case of dolomite and chalk. In addition, it is important to note that so far random forest is the only model capable of predicting limestone, something that did previous models were not able to do so.

4.3.2 Categorical Gradient Boosting

Categorical Gradient Boosting, CatBoost for short, is a recently developed machine-learning algorithm that gets its name derived from the terms Category and Boosting. ‘Cat’ references the fact that it handles categorical features or predictors by itself without necessity of encoding categorical data separately, which is widely required by other machine learning techniques as part of the pre-processing stage. ‘Boost’ refers to its functionality based on gradient boosting algorithm covered in the preceding sections (Ghori et al., 2019).

In addition, CatBoost is compatible with scikit-learn tool kit, and supports training on either CPUs and GPUs. As a first attempt, a hyper-parameter random grid search technique was executed considering the most relevant parameters as shown on Table 23.

Table 23 Categorical Boosting Classifier: Random search grid for CatBoost classifier

Hyper-parameter	Value ranges
depth	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
iterations	[100, 250, 500, 1000]
learning_rate	[0.001, 0.01, 0.03, 0.1, 0.2, 0.3]
l2_leaf_reg	[1, 3, 5, 10, 100]
border_count	[1, 3, 5, 10, 100]
random_strenght	[1, 10, 100, 1000]
grow_policy	['SymmetricTree', 'Lossguide', 'Depthwise']

The random search was executed for 100 epochs or iterations by cross validating each set of parameters on 3 stratified k-folds of the training set. This leaded us to find the values summarized on Table 24 as the optimal ones according to the random search approach. However, once a new model was fitted and tested by using these hyper-parameters, it provided poor accuracies of 71% and 75%, on the open test and hidden test sets respectively.

Besides, considering the high accuracy of 90% obtained on the training set and the considerably low accuracy on the test sets, which is an indicator of overfitting, we decided to implement a manual tuning process as a way to take advantage of the fast training that CatBoost compatibility with GPUs offers. Further, prior to manually attempt to tune the

CatBoost hyper-parameters, a recursive feature selection wrap was first run in order to reduce the possible less informative and nosy predictors held by the datasets.

Table 24 Categorical Boosting Classifier: Optimal hyper-parameters obtained by random search grid approach

Hyper-parameter	Pseudo-optimal value
depth	9
iterations	500
learning_rate	0.2
l2_leaf_reg	3
border_count	100
random_strenght	1.0
grow_policy	‘'Depthwise'

4.3.2.1 Recursive Feature Elimination

Recursive Feature Elimination is an effective feature selection methodology that allows machine-learning algorithms to run more efficiently and effectively. The training data set was treated for missing values, difference in feature scales, and outliers as previously explained with the difference that the categorical variables were not encoded since CatBoost handles them by itself as an attempt to avoid data leakage between the training and test sets while using feature encoders. Data leakage normally leads to conditioned predictions by proposing a new tree ordering principle which is profoundly described in (Prokhorenkova et al., 2019).

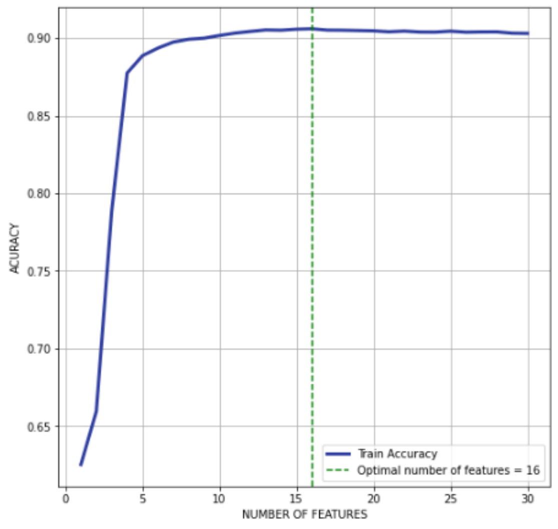


Figure 46 Categorical Boosting Classifier: Recursive feature elimination wrapper results

Later, the preprocessed trainings data went into a recursive feature elimination wrap by cross validating the model with 10 stratified k-folds of the training dataset, this process specified 16 as the optimal number of features for the present model as shown in Figure 46. This suggested that we could remove almost 50% of the training considered to be uninformative for the CatBoost classifier.

In addition, the selected features by the RFE wrapper and their respective importance is depicted in Figure 47. Considering the RFE is a wrapper-type feature selection methodology, which might take any machine-learning model as core for evaluation, the importance scores shown below are fully dependent on the stochastic nature of the CatBoost algorithm.

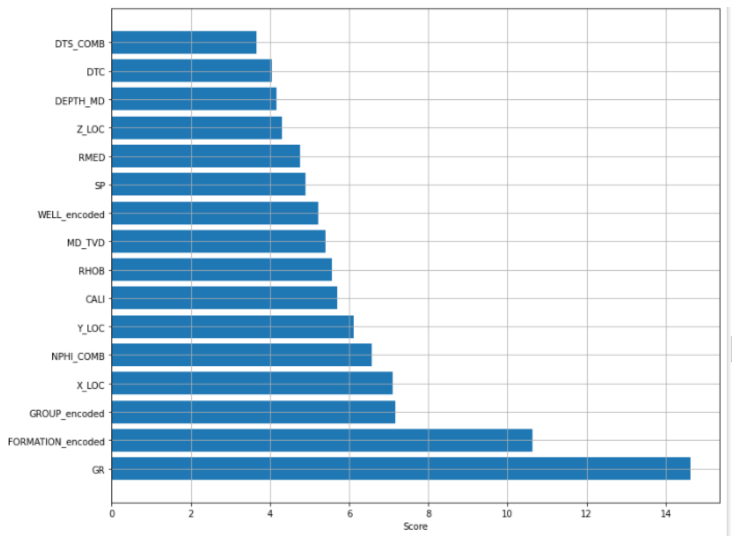


Figure 47 Categorical Boosting Classifier: Feature importance given by the RFE wrapper

From a general perspective, we can observe how the previously machine-learning imputed logs DTS, NPHI, DTC, and RHOB are included as the 16 most informative features for a CatBoost model and how important and decisive the metadata features as FORMATION, GROUP, and LOCATION are as well.

4.3.2.2 Hyper-parameter Tuning

During this stage the training performance was compared with the open set performance which served as validation set for the current analysis. Initially, in order to prevent for under or overfitting the number of trees or iterations had to be set to a large value of 1000. Next, the learning rate was investigated by incorporating cross validation, the open set as validation set, and a 25-round early stopping as callback to prevent for overfitting.

Every change accomplish on the accuracy by changes on the learning rate was documented and compared on the open set (validation set) in order to select the best possible hyper-parameter that generalizes well on unset objects. Figure 48 helps visualize how the train and test accuracies evolve by using different learning rates ranging from 0.001 to 0.5 where the optimal learning rate was found to be 0.1. From this figure, it is also visible how the model overfits after the learning rates exceed a value of 0.2.

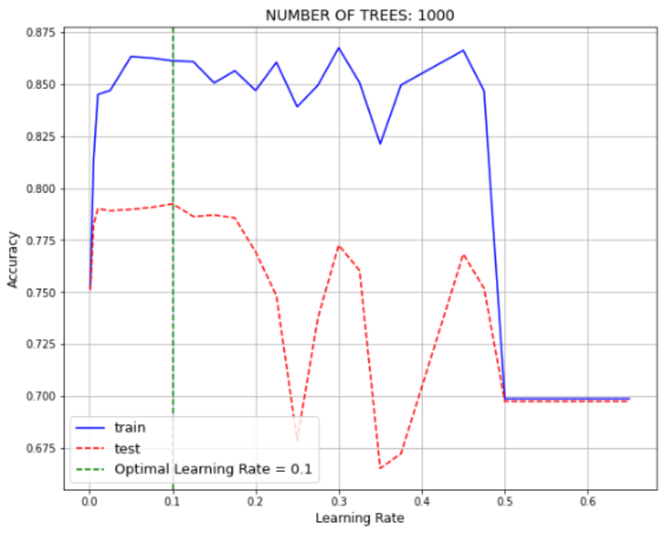
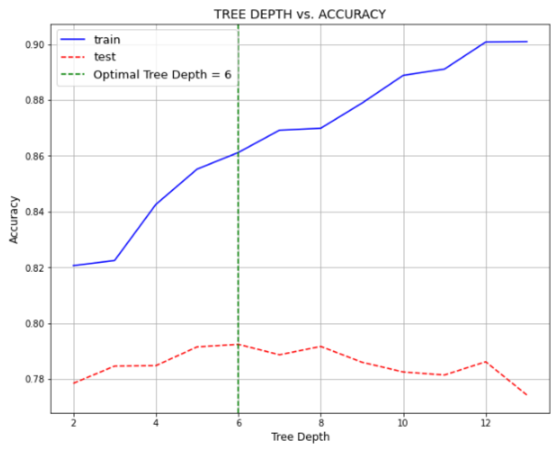


Figure 48 Categorical Boosting Classifier: Learning rate vs. accuracy

Subsequently, considering the massive size of the training data and the limited RAM memory Google Colab provides, a constrained tree depth range from 2 up to 14 was tested and validated on the training and validation sets, respectively. As result of this procedure, a depth of 6 was selected as the most accurate based on the validation set performance. This process is depicted on Figure 49a where it is visible how the model starts overfitting as the tree depth exceeds values over 6 harming in this way the validation set accuracy.

Besides, the coefficient at the L2 regularization term of the cost function was investigated within values equally spaced from 25 to 500 (Figure 49b). An L2 factor equal to 300 showed to give the best accuracy on the validation set and hence was selected for the final model.



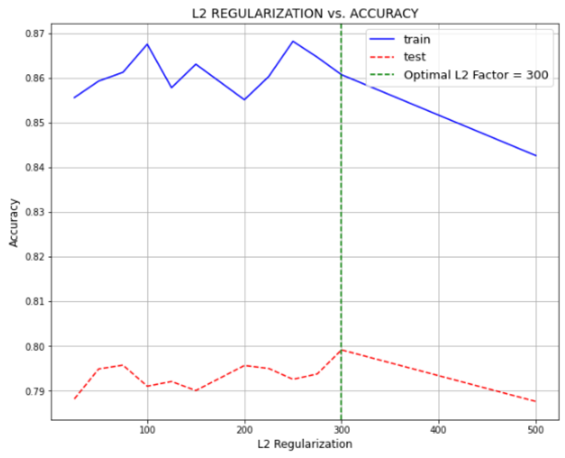
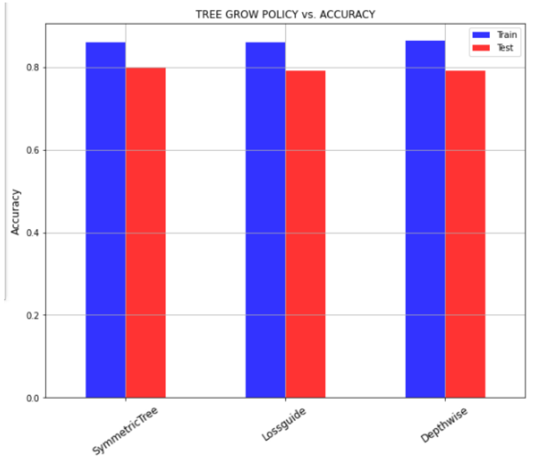


Figure 49 Categorical Boosting Classifier: Tree depth vs. accuracy (left) and L2 regularization term vs. accuracy (right).

Lastly, the tree growing policy and the number of splits for numerical features, also known as border count, parameters were also investigated; however, no other value than the default ones gave better results. These attempts are depicted on Figure 50.



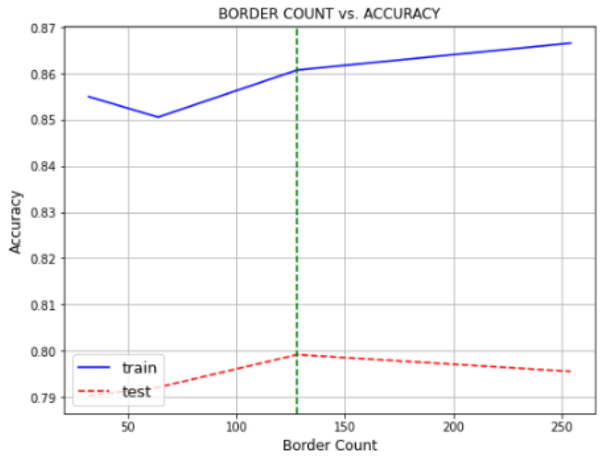


Figure 50 Categorical Boosting Classifier: Tree growing policy vs. accuracy (left) and Border count vs. accuracy (right).

The border count parameter, which mainly depends on the processing unit and has a direct impact in the training speed on a GPU, was investigated in the range from 32 to 254 as recommended by the CatBoost webpage. The optimal parameter was kept on the default value of 128 as there was no other possible value able to beat its influence on the open set performance. Furthermore, even though the Lossguide and Depthwise tree growing policies performed reasonably on the open set, the default Symmetric tree growing policies still provided a best performance. The optimal hyper-parameters found via the manual tuning process can be found enlisted on Table 25 below.

Table 25 CatBoost classifier: Manually tuned hyper-parameters

Hyper-parameter	Optimal Value
iterations	1000
learning_rate	0.1
depth	6
l2_leaf_reg	300
border_count	128 (default)
grow_policy	'Symmetric' (default)

Lastly, a new CatBoost classifier was fitted and tested based on the manually tuned hyper-parameters, this provided prediction accuracies of 86, 80, and 81% on the training, open test, and hidden test sets, respectively. Table 26 represents the detailed classification reports for the training, open test, and hidden test data, where although the open test data was used as validation set while tuning hyper-parameters, the model was still able to generalize well and provide comparable results, and even slightly better results, on the hidden dataset.

Table 26 Categorical Boosting Classifier: Classification reports for the training, open test, and hidden test datasets.

CATEGORICAL GRADIENT BOOSTING CLASSIFICATION REPORT									
Class	**Training Set**			**Open Set**			**Hidden Set**		
	Prec.	*Rec.*	*F1.*	*Prec.*	*Rec.*	*F1.*	*Prec.*	*Rec.*	*F1.*
Sandstone (0)	0.83	0.81	0.82	0.83	0.83	0.83	0.75	0.79	0.77
Sandstone/Shale (1)	0.77	0.58	0.66	0.61	0.29	0.40	0.67	0.43	0.52
Shale (2)	0.89	0.96	0.92	0.83	0.96	0.89	0.88	0.95	0.91
Marl (3)	0.79	0.60	0.68	0.69	0.08	0.14	0.30	0.29	0.30
Dolomite (4)	0.64	0.03	0.05	0.00	0.00	0.00	0.00	0.00	0.00
Limestone (5)	0.82	0.63	0.71	0.44	0.55	0.49	0.63	0.61	0.62
Chalk (6)	0.88	0.85	0.87	0.91	0.02	0.03	0.70	0.44	0.54
Halite (7)	0.98	0.99	0.98	-	-	-	0.99	1.00	0.99
Anhydrite (8)	0.86	0.81	0.83	0.00	0.00	0.00	0.80	0.77	0.78
Tuff (9)	0.78	0.79	0.79	0.75	0.71	0.73	0.59	0.56	0.57
Coal (10)	0.86	0.45	0.59	0.87	0.45	0.60	0.83	0.61	0.71
Basement (11)	0.00	0.00	0.00	-	-	-	-	-	-
Weighted Metric	0.85	0.86	0.85	0.78	0.80	0.77	0.80	0.81	0.80
Accuracy Score	0.86			0.80			0.81		
Matrix Score	-0.36			-0.52			-0.45		

From the classification reports and confusions matrices we can observe that CatBoost was able to classify with medium and high accuracies most of the lithologies, buts it was unable to classify dolomite and anhydrite in particular. Besides, most of the misclassifications are predominant in tuff and dolomite which are misclassified as shaly-sandstone and shale.

Finally, even though CatBoost handles much better but not perfectly data imbalance in an algorithm-level way, there is still a bias in the predictions towards the most frequent classes, yet CatBoost achieved better results than any stand-alone model previously analyzed. Hence, this less visible skew in the prediction distributions towards the most frequent classes while implementing ensemble models has been documented broadly in multiple classification problems, and seems to be worsened as the number of classes increases.

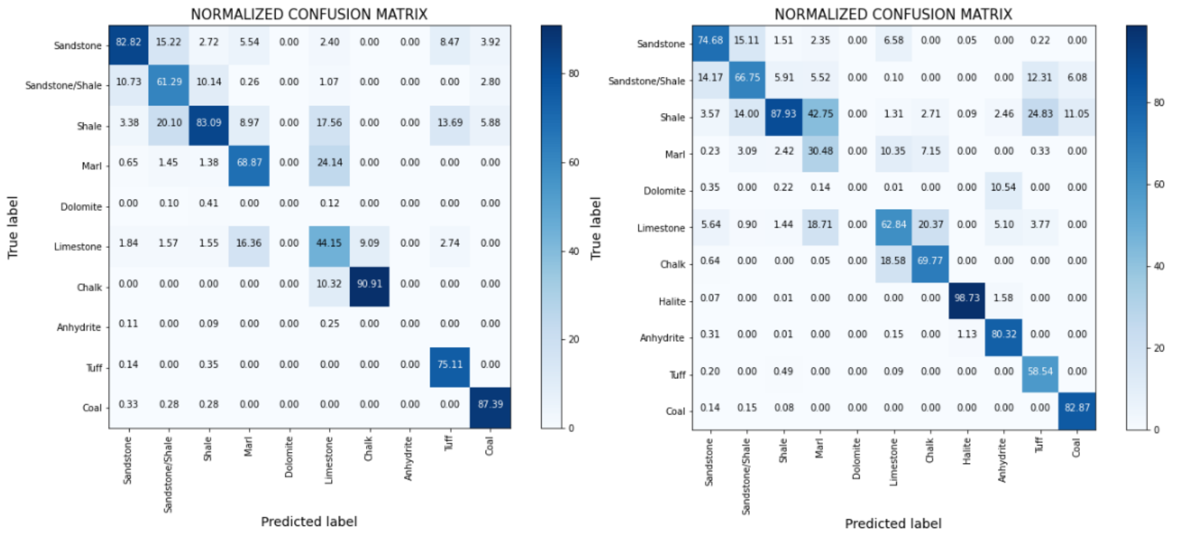


Figure 51 Categorical Boosting Classifier: Classification confusion matrices for the open test set (left) and hidden test set (right).

4.3.2.3 Categorical Gradient Boosting Interpretability

Although ensemble machine-learning algorithms are some of the most robust methods used for classification tasks, their interpretation involves high complexity. This complexity gets higher as the number of classes to be predicted raise meaning that even the most popular feature importance techniques become inconsistent and unable to provide a clear significance for each predictor in relation to each class involved in the prediction task. In order to address this issue the open source SHAP python library was used to get an insight of the individual contribution of each feature into the predictions in a consistent manner by taking into account feature missingness.

Figure 52 briefly summarizes how important each feature is for every predicted lithology class in the form of a bar plot. Of course, this only helps to get a relative but accurate feature importance based on the training set but without representing each feature impact on the model's output range and distribution. This SHAP values bar chat is not comparable to the recursive feature importance plot previously showed due to the difference in the way each is computed; however, they have a general agreement on the top most important features for the CatBoost machine-learning model.

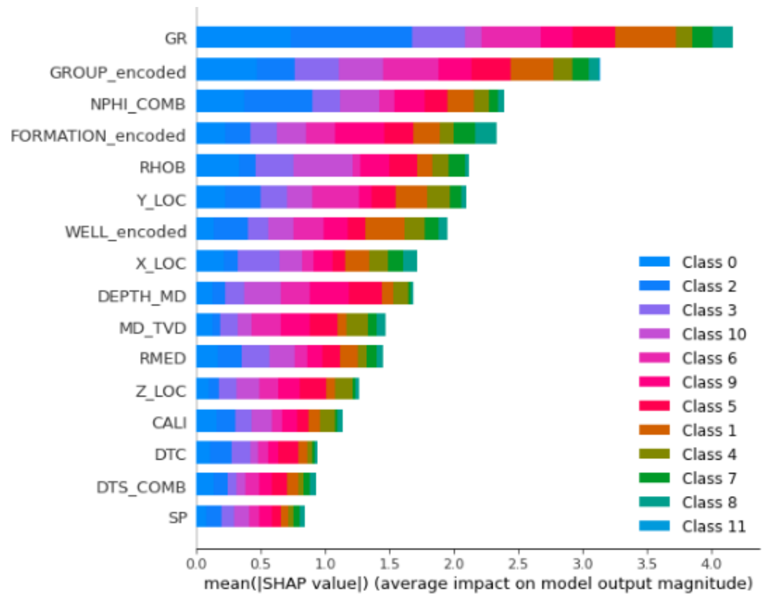


Figure 52 Categorical Boosting Classifier: SHAP values for each target lithology class

In addition, we could analyze each feature influence on the model's output for each lithology class, but for simplification for the current section, we only focus on some examples of the less accurately classified lithologies such as shaly-sandstone and dolomite, which were the ones the CatBoost model misclassified the most. Refer to Appendix I to find SHAP values impact for all lithology types.

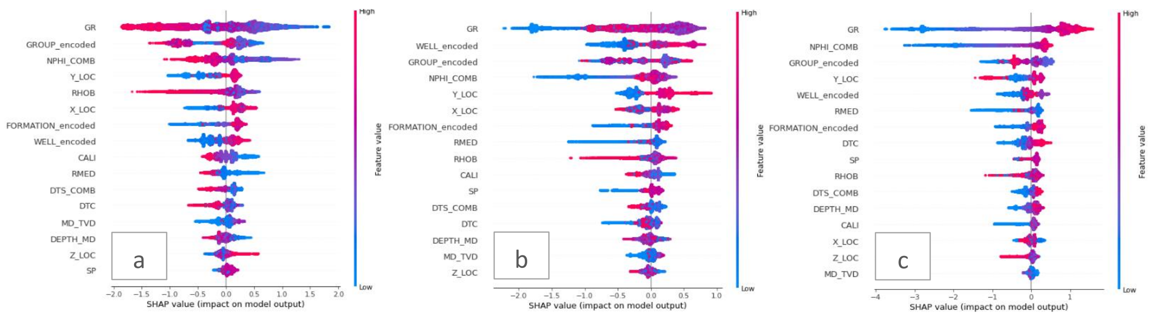


Figure 53 Categorical Boosting Classifier: (a) SHAP values impact while predicting sandstone, (b) SHAP values impact while predicting shaly-sandstone, (c) SHAP values impact while predicting shale.

Figure 53, describes how each feature influences the classification output for sandstones, shaly-sandstones, and shale, from here we can observe that for an instance to be classified as a shale or as a sandy-shale, the GR, NPHI, Y_LOC, X_LOC, GROUP, and FORMATION features play the most important role. Besides, we can appreciate that there is not a well-defined boundary for most of the mentioned features to distinguish between sandstones from the shaly-sandstones, as there is to differentiate shale from the other two classes.

For instance, a high GR is more likely to help the CatBoost to classify such instance as a shale as seen on Figure 53c, and a low-medium GR is needed to classify a data instance as sandstone as seen on Figure 53a. However, there is not such boundary properly defined to predict a data instance as a shaly-sandstone, since as we can observe on Figure 53b, either a high or medium value is needed to do so. Thus, a medium GR values easily create confusion while training the classifier to distinguish between sandstones and shaly-sandstones. In consequence, this lack of a well-defined feature boundary to distinguish these two classes are the reason why, the CatBoost classifier does better while distinguishing shale from other classes than when sorting out shaly-sandstones from sandstones (See Table 26).

Moreover, following the same logic we could explain the CatBoost incapability to properly classify lithologies that share similar composition and properties such as the case of dolomites, limestone, and chalk . The poorest classification between these three classes was encountered on dolomites (See Table 26); fact that may be explained by the almost null presence of dolomites on the training set which could have made the CatBoost model unable to learn how to classify them in a considerable good manner (See Figure 14).

4.3.3 Light Gradient Boosting

Light Gradient Boosting algorithm, LightGBM for short, is a highly efficient gradient boosting decision tree that from a general perspective exclude a significant portion of the data instances with small gradients during the estimation of the information gain. This implies having an algorithm with almost the same efficiency but several times faster during the training process in comparison with conventional Gradient Boosting Decision Trees (GBDT) machine-learning models (Ke et al., 2017).

The LightGBM base model performance we constructed initially by using the complete set of 30 features, did not consider any regularization term or technique, providing accuracies of 84, 72, and 65% on the training, open test, and hidden test sets, respectively, which means that the model was unable to generalize its performance on unseen objects. Consequently, in the current section we attempted to optimize the LightGBM model's performance by fist running a recursive feature elimination wrapper and then undergoing into a manual hyper-parameter tuning process. In addition, it is worth to mention that LightBoost library offers compatibility with either CPUs or GPUs, which made possible optimizing the model's hyper-parameters manually.

4.3.3.1 Recursive Feature Elimination

A recursive feature elimination wrapper was executed in order to study the possibility of reducing the dataset magnitude without affecting the model performance. This process accompanied by a 10 stratified k-fold cross validation achieved to determine 24 as the optimal set of predictors that reached the highest training accuracy as shown on Figure 54.

The features importance obtained by the RFE wrapper are depicted on Figure 55. As visible, the four features we improved by the ML imputation technique are still considered to be highly relevant for LightGBM as well as the additional features we created. However, surprisingly LightGBM provided a high importance to variables such as ROP and DRHO, same that have not been considered highly relevant in other machine-learning algorithms.

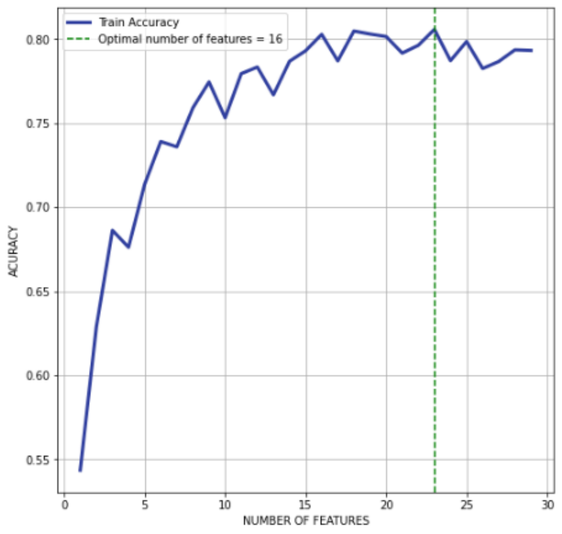


Figure 54 Light Boosting Classifier: Recursive feature elimination wrapper

4.3.3.2 Hyper-parameter tuning

While many other popular Gradient Boosting Decision Trees algorithms base their functionality on a depth-wise growing policy, LightGBM uses leaf-wise growing policy which normally help the algorithm to converge much faster; however, this might also help to overfit the model if wrong hyper-parameters are selected. Further, based on the extensive number of hyper-parameter handled by LightGBM, it became time demanding to tune the complete set of hyper-parameters by implementing either a random or a grid parameter search techniques.

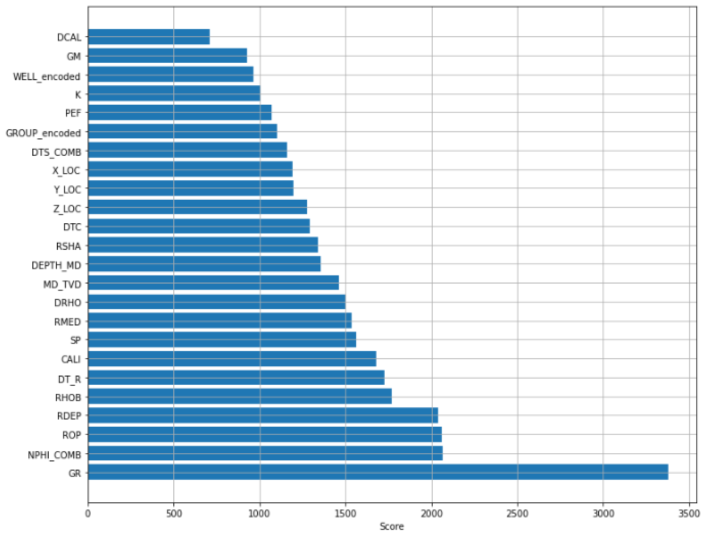


Figure 55 Light Boosting Classifier: Feature importance given by the RFE wrapper

Accordingly, a manual tuning process for the three most important regularization parameters was executed in order to establish the best set of values that outperform the default values. These highly influencing regularization hyper-parameters were evaluated individually and sequentially by inspecting the training and validation accuracies while the number of estimators was set to constant value of 1000 to prevent for overfitting or underfitting.

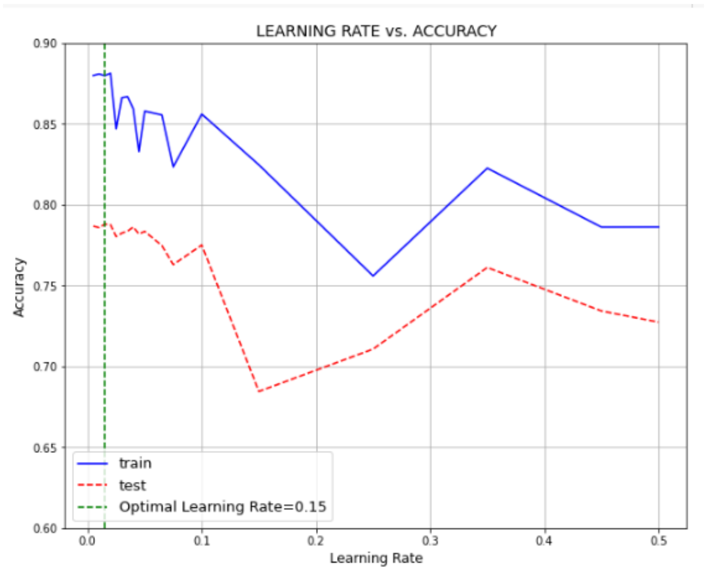
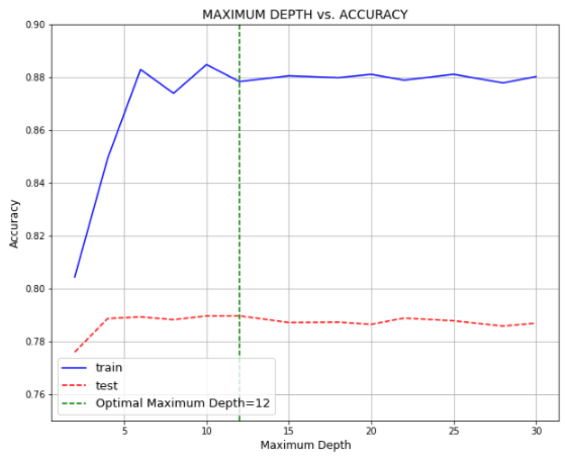


Figure 56 Light Boosting Classifier: Learning rate vs. accuracy

First, the learning rate was investigated in the range from 0.005 to 0.5 by the aid of the open test dataset as validation set, a 10 K-Fold cross validation technique on the training data, and 25-round early stopping callback to stop the training process if no improvement on the multiclass probability objective function was obtained. This process is documented on Figure 56 where the optimal learning rate was found to be 0.015; in addition, the figure also represents how the training and validation accuracies worsen dramatically as the learning rate exceeds a value of 0.1. In other words, training accuracy fall means that LightGBM was unable to converge and optimize the objective function at learning rates higher than 0.1.

Second, maximum depth is a parameter that not only controls the distance or steps between the root node and the leaf node, but also has a high influence on the model training time. In this context, several maximum depths ranging from 2 to 30 were studied and validated on the training and validation data. Besides, the regularization factor L2 was also looked at in the range from 1 to 300 in order to prevent for overfitting.



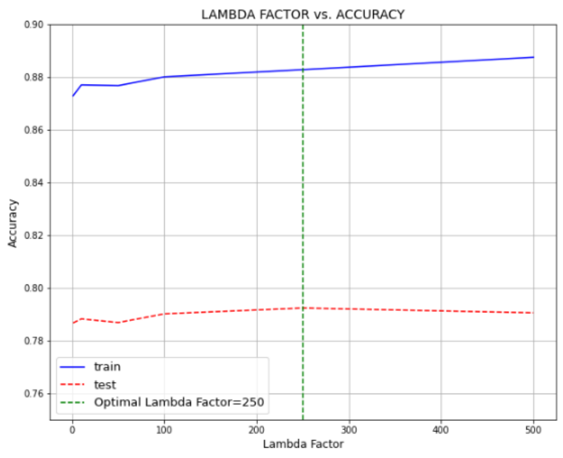


Figure 57 Light Boosting Classifier: Maximum tree depth vs. accuracy (left) and Regularization lambda L2 vs. accuracy (right).

As result, a maximum depth of 12 and regularization factor of 250 were selected as optimal values based on slight improvements on the validation accuracy which showed practically constant values along the studied parameter ranges as shown on Figure 57. In addition. It is important to note that although the last two studied parameters seemed not to have a high impact on the model accuracy, their definition would help LightGBM generalize better on unseen objects.

Table 27 Light Boosting Classifier: Manually tuned hyper-parameters

Hyper-parameter	Optimal Value
iterations	1000
learning_rate	0.015
max_depth	12
Reg_lambda	250

Finally, a new LigthGMB model was trained and test by using the optimal hyper-parameters found via the manual tuning process (See Table 27), obtaining classification accuracies of 88, 79, and 80% on the training, open test, and hidden test sets, respectively.

A detailed classification report for each predicted class and the confusion matrices normalized by the number of predictions per class are presented on Table 28and Figure 58, respectively.

Table 28 Light Boosting Classifier: Classification reports for the training, open test, and hidden test datasets.

LIGHT GRADIENT BOOSTING CLASSIFICATION REPORT									
Class	**Training Set**			**Open Set**			**Hidden Set**		
	Prec.	*Rec.*	*F1.*	*Prec.*	*Rec.*	*F1.*	*Prec.*	*Rec.*	*F1.*
Sandstone (0)	0.85	0.84	0.85	0.82	0.82	0.82	0.72	0.83	0.77
Sandstone/Shale (1)	0.80	0.64	0.71	0.55	0.31	0.40	0.65	0.38	0.48
Shale (2)	0.91	0.97	0.94	0.83	0.95	0.89	0.87	0.95	0.91
Marl (3)	0.84	0.76	0.80	0.63	0.15	0.24	0.22	0.21	0.21
Dolomite (4)	0.66	0.16	0.26	0.00	0.00	0.00	0.00	0.00	0.00
Limestone (5)	0.86	0.68	0.76	0.49	0.54	0.51	0.61	0.52	0.56
Chalk (6)	0.92	0.93	0.93	0.00	0.00	0.00	0.68	0.58	0.62
Halite (7)	0.99	0.99	0.99	-	-	-	0.99	1.00	0.99
Anhydrite (8)	0.93	0.91	0.92	1.00	0.06	0.12	0.90	0.10	0.18
Tuff (9)	0.89	0.85	0.87	0.65	0.44	0.52	0.63	0.31	0.41
Coal (10)	0.83	0.65	0.73	0.76	0.50	0.61	0.82	0.54	0.65
Basement (11)	0.00	0.00	0.00	-	-	-	-	-	-
Weighted Metric	0.88	0.88	0.88	0.77	0.79	0.77	0.79	0.80	0.79
Accuracy Score	0.88			0.79			0.80		
Matrix Score	-0.304			-0.533			-0.477		

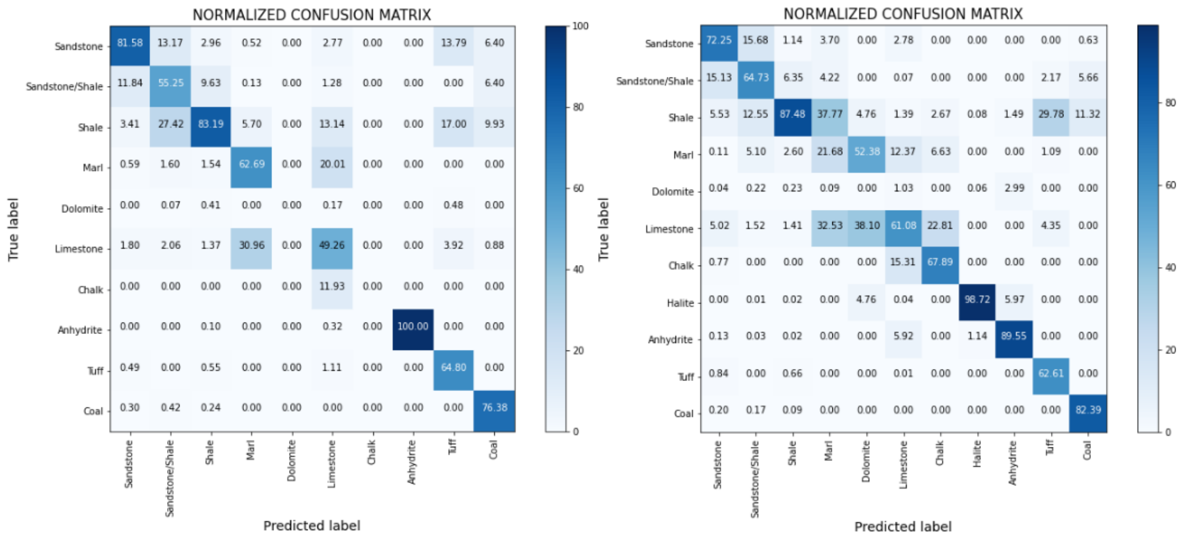


Figure 58 Light Boosting Classifier: Classification confusion matrices for the open test set (left) and hidden test set (right).

LightGBM seems to achieve high-medium level accuracies when there is no conflict between similar lithologies. In other words, when predicting similar classes, LightGBM seems to have struggled and made many more classification mistakes, particularly for dolomite for which the model could not make any right prediction at all. In addition, all the predictions seemed

]

to be consistent and comparable between the open and hidden test sets; however, there are still some minor differences on the accuracies attained on each dataset. This apparent difference in performance is more linked to the test sets' labels distributions than to the model itself (See Figure 14). For instance, chalk is much better classified in the hidden test set than in the open test set, which is greatly due to the difference on the lithologies presence on each set from which the LightGBM model could have made either right or wrong prediction.

4.3.3.3 Light Gradient Boosting Interpretability

As discussed previously, understanding why a model makes certain predictions can be a crucial task in regression and classification problems, overall when accuracy and interpretability are discussed together considering that the best performing machine-learning algorithms are also the most complex ones such as the case of ensemble and deep learning models (Lundberg and Lee, 2017). To address this issue the open source python library SHAP, short for Shapley Additive exPlanations, was used to dig deeper into LightGBM interpretability.

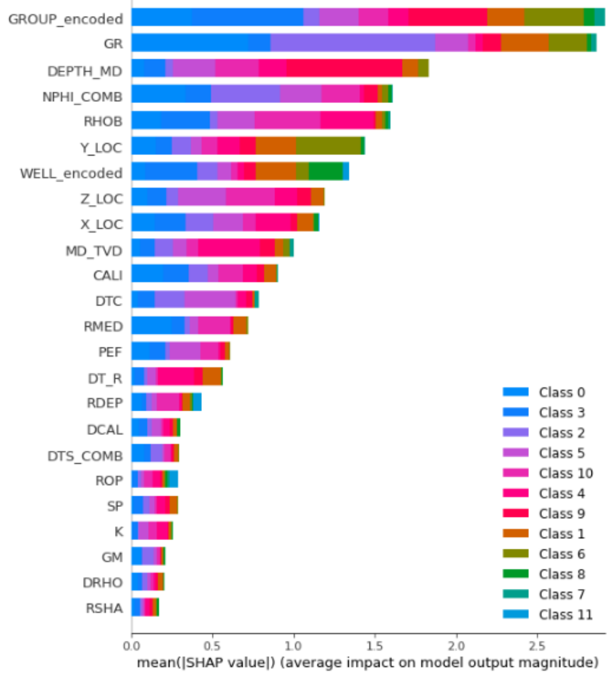


Figure 59 Light Boosting Classifier: SHAP values for each target lithology class

Figure 59 depicts how important the metadata features such as GROUP, DEPTH, Y_LOC, Z_LOC, and Y_LOC are for classifying lithofacies while implementing LightGBM. In addition, the SHAP impact values showed a great disagreement on some features considered as important when compared to the RFE wrapper feature importance previously obtained. However, the NPHI and RHOB machine-learning imputed features are still at the top of the most relevant features helping LightGBM perform well.

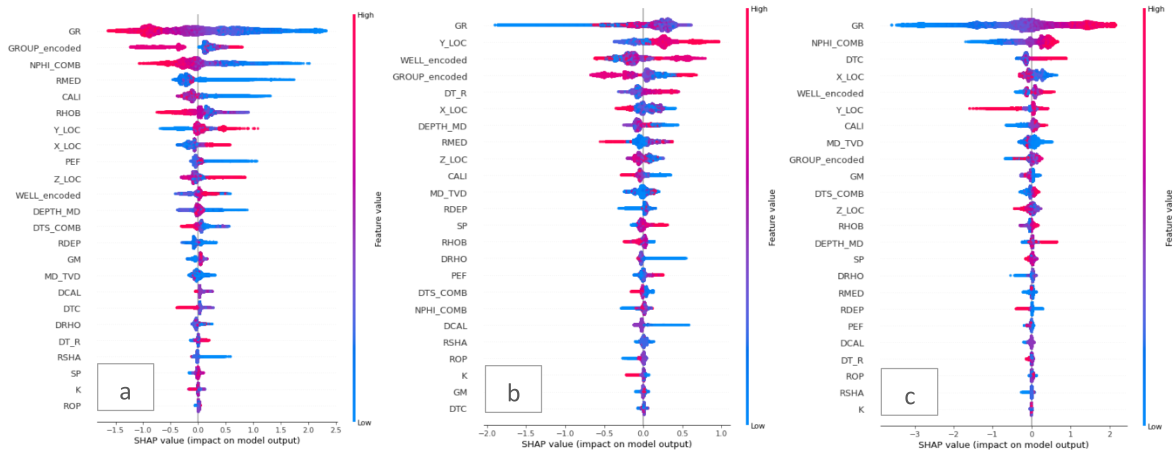


Figure 60 Light Boosting Classifier: (a) SHAP values impact while predicting sandstone, (b) SHAP values impact while predicting shaly-sandstone, (c) SHAP values impact while predicting shale.

Moreover, Figure 59 also reveals how sandstone, shaly-sandstone, and shale classifications are mainly impacted by the GR and NPHI logs, but when it comes to other similar lithologies such as limestone, dolomite, and chalk, LightGBM needed other additional features such as RHOB and the acoustic logs to take part of the classification task.

In addition, by following the same logic we used to interpret CatBoost, the LightGBM classification performance for similar classes rely mainly on the GR, and NPHI readings along with some other metadata features, where medium-size features readings created great confusion while defining proper boundaries capable of separating these lithologies. For instance, GR and NPHI have quite well defined boundaries that help LightGBM distinguish between shale and sandstone, low GR and low NPHI for sandstones and high GR and high NPHI for shale. However, when these boundaries fade away as while classifying shaly-sandstones, GR, and NPHI become less informative and less useful to accomplish the prediction task for this particular class as shown on Figure 60. Refer to Appendix J to find SHAP values impact for all lithology types.

Therefore, the complexity that classifying similar lithologies involves along with the low amount of training samples available for some these classes such as dolomite leaded

LightGBM to perform partially well in general, but poorly particularly while classifying shaly-sandstones and dolomites, where the lowest performances were found (See Table 28).

4.3.4 Extreme Gradient Boosting

Extreme Gradient Boosting, XGBoost in short, is a highly robust, powerful, efficient, scalable, and widely used Gradient Boosting Decision Trees machine-learning model consider to lead the forefront when it comes to classification tasks. XGBoost is an almost perfect blend of software and hardware capabilities designed to enhance the pre-existing boosting techniques in terms of training time and efficacy. It introduced two additional techniques that help the model prevent overfitting. The first technique known as columns or feature subsampling, originally part of random forest, which helps to train each independent learner more efficiently on a different subset of features. The second technique is known as shrinkage that, similarly to a learning rate in stochastic optimization, reduces the influence of each individual tree by scaling the output weights after each step of the tree boosting optimization (Chen and Guestrin, 2016).

Even though during the initial part of section 5, we created a base XGBoost model that did not consider any regularization technique to prevent for under or overfitting, it was still able to achieve good and pseudo-balanced performance results when dealing with unseen objects, 79 and 80% in the open test and hidden test sets, respectively. However, based on the great results XGBoost has obtained along several data science competitions for both classification and regression task, we believed that a proper hyper-parameter selection could improve its performance.

Consequently, the current section presents a dimensionality reduction process through RFE accompanied by a manual hyper-parameter selection by taking advantage of the quick and parallelized learning process offered by the XGBoost's compatibility with GPUs, same that allowed to process and exploit profoundly the complete datasets.

4.3.4.1 Recursive Feature Elimination

Initially, in order to be consistent with the previously analyzed boosting algorithms a costly Recursive Feature Elimination wrapper was executed to filter out the less informative predictors that could lead to confusion during the training stage so that higher performances could be achieved in shorter training times.

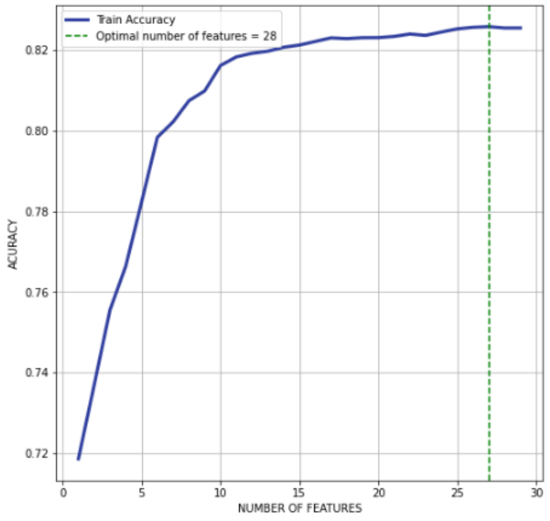


Figure 61 Extreme Boosting Classifier: Recursive feature elimination wrapper

The top training accuracy seemed to go beyond 82% while only using 10 training features; however small improvements are achieved by including 18 more features (See Figure 61). For the context of the current model, which can be run on GPUs, we kept 28 as the optimal number of features in order to optimize as much a possible the classification accuracy.

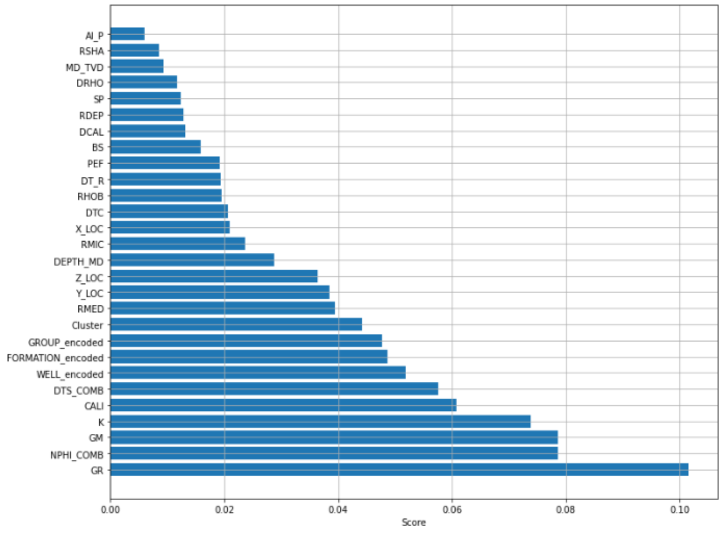


Figure 62 Extreme Boosting Classifier: Feature importance given by the RFE wrapper

In addition, the apparent importance each feature has on XGBoost is described on Figure 62, which in comparison to LightBoost and CatBoost confers more weight to some of the 7 additional features created on Section 4, such as bulk modulus K, Shear modulus GM, Cluster, and Slowness ratio DT_R.

4.3.4.2 Hyper-parameter Tuning

A manual hyper-parameter tweaking was focus on the most relevant parameters consider to be the learning rate and the tree depth. Each parameter evaluated in the current section used a 10 stratified K-Fold cross validation, a 25-round early stopping callback, and the open set as validation set, while the number of trees was set to a value of 1000 in order to prevent underfitting the training data.

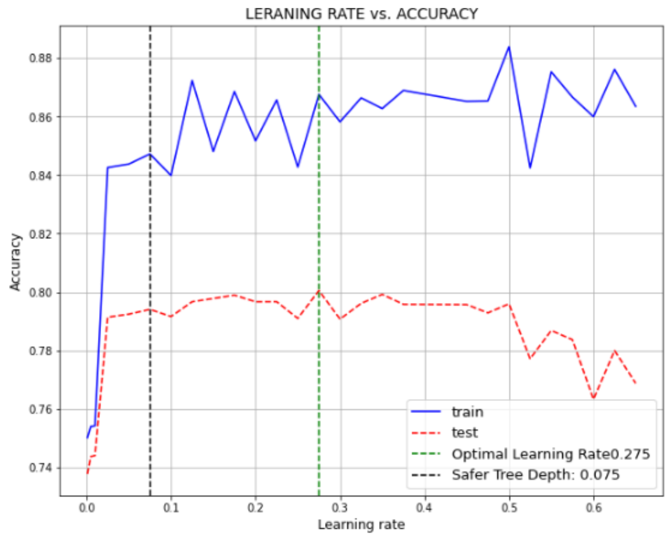


Figure 63 Extreme Boosting Classifier: Learning rate vs. accuracy

First, the learning rate was investigated in ranges from 0.001 to 0.65 by incorporating cross validation, the open test set as validation set, and a 25-round early stopping as callback to reduce overfitting. The results of the learning rate investigation are documented on Figure 63 where the optimal learning rate according to model's best performance on the open test data was found to be 0.35; however, as visible on the plot there are great fluctuations on the test accuracies while using learning rates from 0.20 to 0.35. This leaded us to think that any slight performance improvements on these sections may have been obtained by chance and not by the model's capability to generalize accurately its performance. Thus, selecting a lees greedy and more stable learning rates between 0.01 and 0.25 might be safer and more accurate while dealing with unseen objects.

Based on the previous reasoning we opted to set the optimal learning rate to 0.075 before undergoing into the next hyper-parameter analysis. Next, an estimator depth range from 2 to 15 was looked at and documented as shown in Figure 64 were XGBoost seemed to generalize better on the open test data when a tree depth equal to 4 is selected.

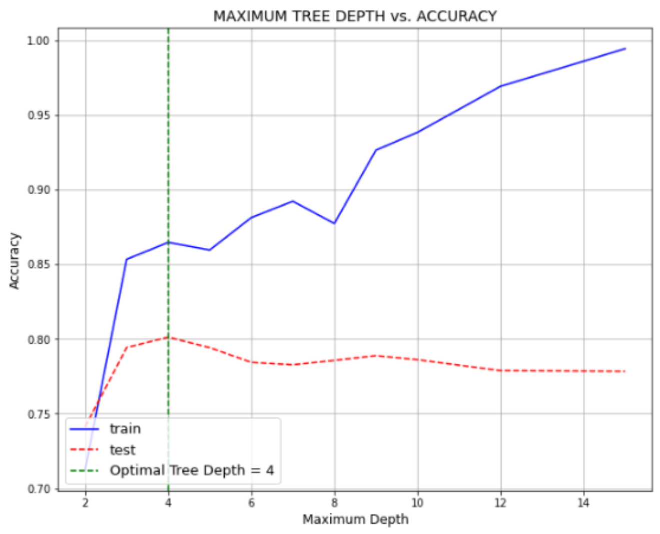


Figure 64 Extreme Boosting Classifier: Tree depth vs. accuracy

Finally, based on the optimal hyper-parameters found by the manual tuning process enlisted on Table 29, a new XGBoost model was fitted and tested obtaining training, open test, and hidden test accuracies of 88, 80, and 82%, respectively.

Table 29 Extreme Boosting Classifier: Manually tuned hyper-parameters

Hyper-parameter	Optimal Value
n_estimators	1000
learning_rate	0.075
max_depth	4
reg_lambda	1500
subsample	1 (default)
colsample_bytree	1(default)

A detailed prediction report separated by predicted classes and a confusion matric normalized by the number of prediction per class are evidenced on Table 30 and Figure 65 where the most remarkable observation is that XGBoost achieved better classification performance that the other Gradient Boosting tree based model especially for mixed mineral lithologies such as shaly-sandstones, limestone, and chalk.

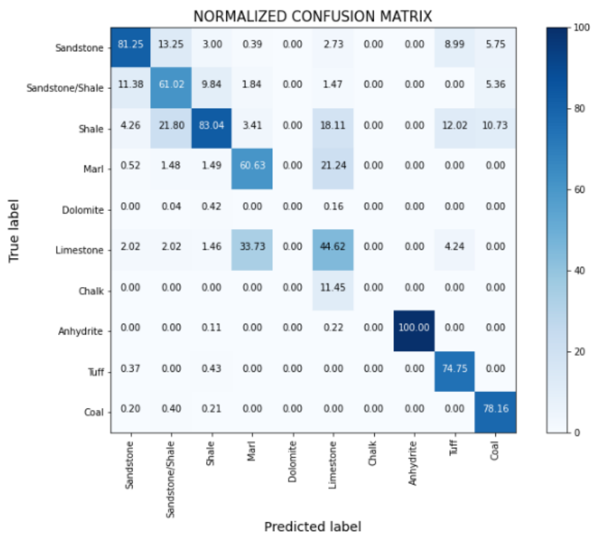
Table 30 Extreme Boosting Classifier: Classification reports for the training, open test, and hidden test datasets

EXTREME GRADIENT BOOSTING CLASSIFICATION REPORT									
Class	**Training Set**			**Open Set**			**Hidden Set**		
	Prec.	*Rec.*	*F1.*	*Prec.*	*Rec.*	*F1.*	*Prec.*	*Rec.*	*F1.*
Sandstone (0)	0.83	0.82	0.83	0.81	0.82	0.82	0.75	0.83	0.78
Sandstone/Shale (1)	0.77	0.58	0.66	0.61	0.30	0.40	0.64	0.46	0.53
Shale (2)	0.89	0.96	0.93	0.83	0.95	0.89	0.89	0.94	0.92
Marl (3)	0.80	0.67	0.73	0.61	0.14	0.23	0.31	0.30	0.31
Dolomite (4)	0.58	0.11	0.19	0.00	0.00	0.00	0.41	0.10	0.16
Limestone (5)	0.83	0.65	0.73	0.45	0.51	0.47	0.73	0.59	0.66
Chalk (6)	0.90	0.90	0.90	0.00	0.00	0.00	0.81	0.76	0.79
Halite (7)	0.99	0.99	0.99	-	-	-	0.99	0.99	0.99
Anhydrite (8)	0.91	0.88	0.90	1.00	0.04	0.08	0.73	0.65	0.69
Tuff (9)	0.82	0.86	0.84	0.75	0.59	0.66	0.64	0.54	0.59
Coal (10)	0.82	0.57	0.67	0.78	0.59	0.67	0.81	0.70	0.75
Basement (11)	1.00	0.17	0.28	-	-	-	-	-	-
Weighted Metric	0.87	0.87	0.87	0.77	0.80	0.77	0.82	0.83	0.82
Accuracy Score	0.87			0.80			0.83		
Matrix Score	-0.352			-0.531			-0.433		

Moreover, there is still a visible bias in the prediction obtained by XGBoost towards the majority classes although XGBoost has built-in functions to decrease the impact class imbalance has on classifications. Finally, even though XGBoost presented difficulties while properly differentiating between dolomite, chalk, and limestone, it was still able to classify with high accuracy sandy and shaly lithologies, which are normally the most relevant for oil and gas conventional reservoirs. These and more details will be discussed on the model comparison section of the current study.

4.3.4.3 XGBoost Interpretability

Decision trees-based machine-learning algorithms have been consider black-box models so far due to the complexity they involve; in consequence, endowing these kind of models with some interpretability is a major task before and after their execution. By doing this we might provide some insight into how a model could be improved while at the same time we could support a profound understanding on the process being modelled. SHAP values assign a unique additive feature importance for a particular prediction, which serves to understand how important and impactful a predictor is to a particular outcome obtained by the trained model (Lundberg and Lee, 2017). The Shapley Additive exPlanations technique, SHAP, implemented on python provided the following color-bar chart in which we could explore the contribution of each feature to the model final prediction Figure 66.



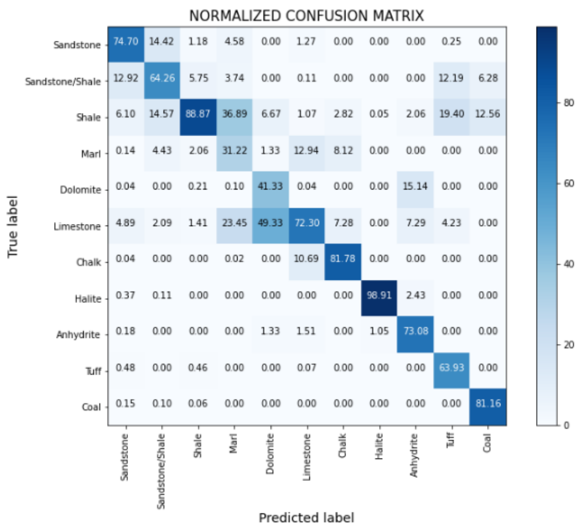


Figure 65 Extreme Boosting Classifier: Classification confusion matrices normalized by the number of predictions by class (a) Open test set, (b) Hidden test set.

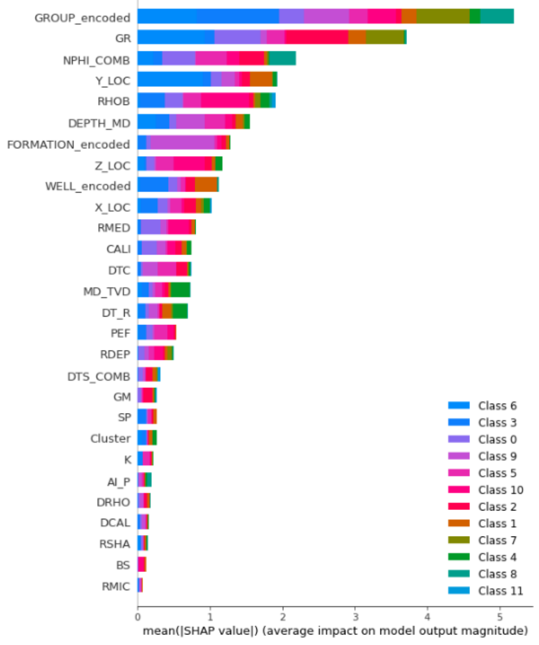


Figure 66 Extreme Boosting Classifier: SHAP values for each target lithology class

Figure 66 briefly shows a relative but accurate manner of representing the feature impact on the XGBoost output, which in comparison to the recursive feature elimination process taken beforehand rested importance to the shear GM, bulk modulus GM, CALI, and Cluster features, while providing even more importance to some of the machine-learning preprocessed features such as NPHI and RHOB.

In addition, we could attempt to analyze the importance each feature had for the prediction for each particular lithology class, however, as a matter of simplicity, we will only take some representative examples of the classes that XGBoost misclassified the most such as the case of dolomite, shaly-sandstone, and marl. Refer to Appendix K to find SHAP values impact for all lithology types.

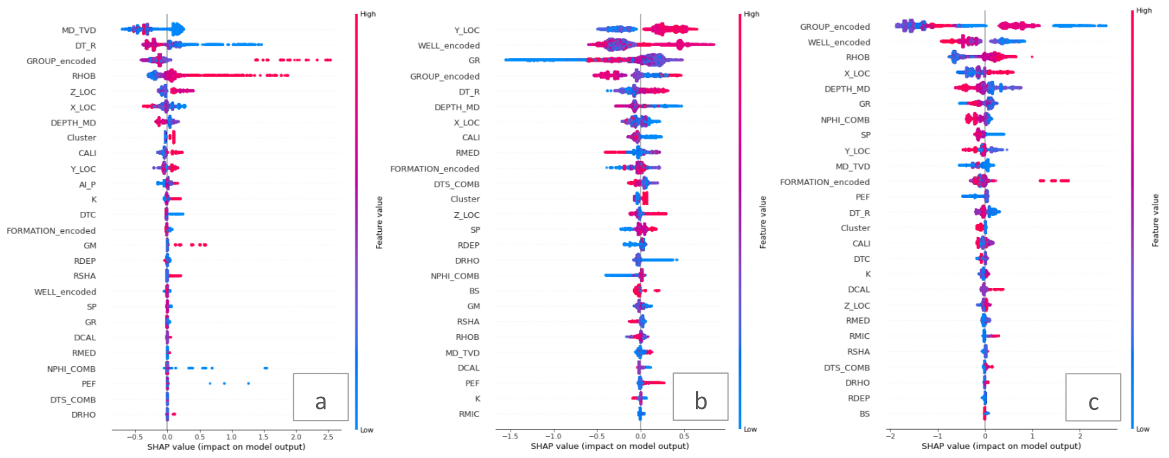


Figure 67 Extreme Boosting Classifier: (a) SHAP values impact while predicting dolomite, (b) SHAP values impact while predicting shaly-sandstone, (c) SHAP values impact while predicting marl.

As shown on Table 30, it seemed that dolomite is the hardest lithology to be predicted in almost any gradient boosting model including XGBoost, which we presume to be linked to the low amount of dolomite samples available for training, which accompanied by its inherent similarity to limestone and chalk could have made XGBoost unable to properly learn how to classify this lithology type. In addition, Figure 67a showed that the current XGBoost model only considers a low number of features like MD_TVD ratio, slowness ratio DT_R, GROUP, and RHOB as the ones that positively contribute to classify a particular instance as dolomite. In other words, it means that the misclassification might have been caused by the lack of enough dolomite samples or by the poor relationship between the variables and the target in such specific case.

Moreover, in the case of shaly-sandstone, which normally may be confused with either sandstones or shale, we can observe on Figure 67b how the instance Y_LOC impacts more that the GR reading which in generally speaking helped XGBoost to get better results while classifying such lithology class when compared to the results obtained by the other gradient boosting tree based algorithms. Hoverer, the unclear boundary on the GR to separate sandstone form shale and shaly-sandstones still plays an important role to properly distinguish between these lithologies.

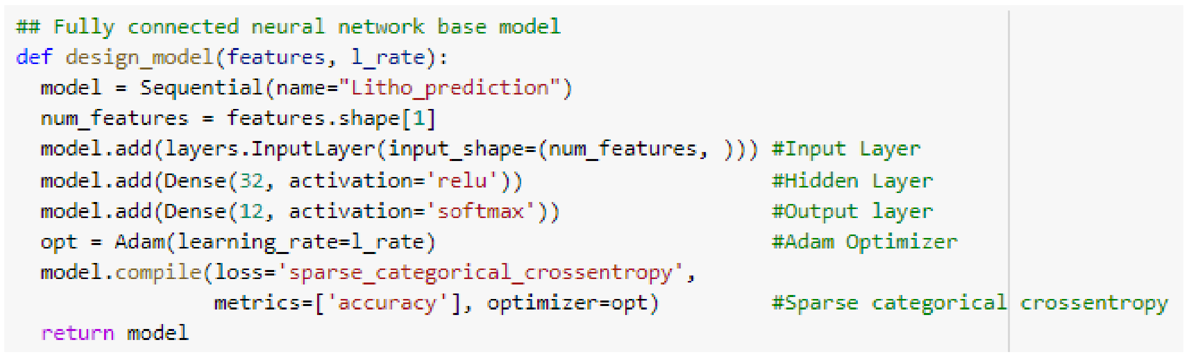
Finally, marl prediction relies mainly on metadata features such as GROUP, X_LOC, DEPTH_MD, and minorly on other reading such as RHOB, GR, NPHI, and SP to mention some (See Figure 67c). Apparently, the possible reasons why XGBoost struggled to classify marl, which is a mix of clay and calcium carbonate, was that it could easily be confused with shaly sediments or any type of limestone such as sandy-shale, shale, dolomite, limestone and chalk. In other words, marl encompasses a wide spectrum of analogous classes that hindered its proper classification.

4.4 Deep Learning – Neural Network

The methodology to analyze neural network performance on the lithofacies classification problems relies on three major steps including a one-hidden sequential fully connected base model, feature selection process, and finally a Bayesian hyper-parameter optimization by using scikit-optimize library skopt.

4.4.1 One-hidden Layer Base Model

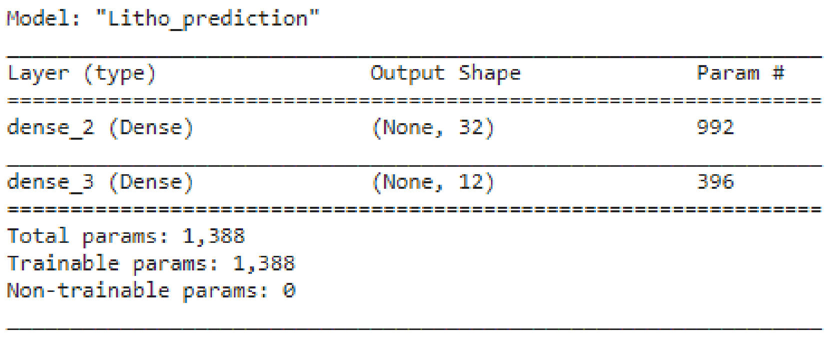
A fully connected sequential model was constructed as a baseline to test how a neural network performs to classify lithofacies. The NN structure consisted of 1 input layer, 1 hidden layer with 32 neurons, a RELU activation function, and 1 output layer using a softmax activation function. Besides, an Adam optimizer and a sparse categorical cross-entropy loss function to save memory and time were included into the base neural network. Finally, a standard learning rate of 0.01 was used to back propagate and minimize the loss function. Refer to Section 2 and Figure 13 to see how neural network weight optimization works, or to Nielsen, (2015) for detailed information about gradient descent and back propagation. The structure of the neural network is summarized on Figure 68.



```
## Fully connected neural network base model
def design_model(features, l_rate):
  model = Sequential(name="Litho_prediction")
  num_features = features.shape[1]
  model.add(layers.InputLayer(input_shape=(num_features, ))) #Input Layer
  model.add(Dense(32, activation='relu'))                    #Hidden Layer
  model.add(Dense(12, activation='softmax'))                 #Output layer
  opt = Adam(learning_rate=l_rate)                           #Adam Optimizer
  model.compile(loss='sparse_categorical_crossentropy',
                metrics=['accuracy'], optimizer=opt)         #Sparse categorical crossentropy
  return model
```

Figure 68 Neural Network: Base model structure

The neural network base model contains 1388 trainable parameters between weights and biases, 992 of which belong to the hidden layer and 396 to the output layer. It is important to note that the number of trainable parameter in any hidden or output later is equal to the sum of the number weights plus the number of biases. The number of weights is equal to the number of neurons times the number of predictors or features contained in the training data, and the number of biases corresponds to a one dimensional array equal to the number of neurons present in a particular layer (See Figure 69)



```
Model: "Litho_prediction"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 32)                992
_________________________________________________________________
dense_3 (Dense)              (None, 12)                396
=================================================================
Total params: 1,388
Trainable params: 1,388
Non-trainable params: 0
_________________________________________________________________
```

Figure 69 Neural Network: Base model number of trainable parameters and output shape in each layer.

In addition, prior to start the training stage a 40-epoch early stopping was created to monitor the training process while cross validating the training data to open test dataset, then the model was trained based on the original 30 features obtained after the data processing stage. The training evolution of the base neural network is shown in Figure 70.

Consequently, the base model performed with accuracies of 76, 73, and 73% on the training, open test, and hidden test sets, respectively. Moreover, as visible on Figure 70 the base model showed a highly unstable learning process, and the loss function did not decrease either but increase over each iteration. This seemed to be caused by a well-known problem know as exploding gradient problem, which could be defined as an error in the direction and the magnitude of the learning step while training a neural network, which consequently derives in an unstable gradient problem.

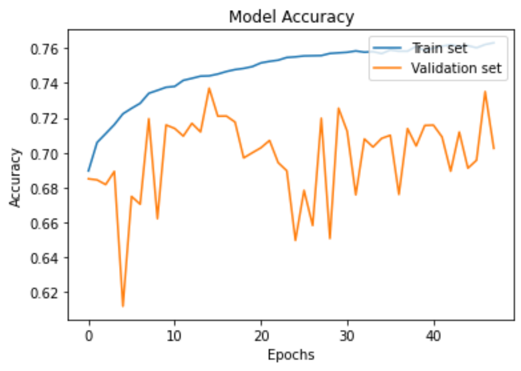


Figure 70 Neural Network: 30-feature-based baseline model training history

As consequence of the gradient descent problem, we decided to try out several approaches in order to stabilize the gradient descent. The main changes we included into the base model structure were, a random normal weight initialization, a zeros bias initialization, and a momentum based stochastic gradient descent optimizer SDG. Figure 71 documents a much more stable learning history and how the loss gets minimized after each iteration once these changes were effectuated.

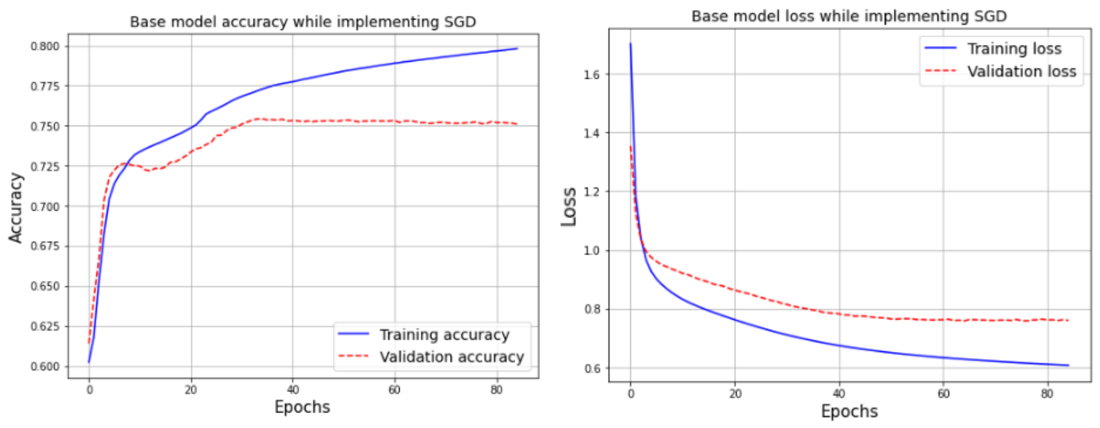


Figure 71 Neural Network: Stochastic Gradient Descent-based neural network base model accuracy history (left) and loss function history (right).

The new stabilized base model based on a stochastic gradient descent SGD provided much better accuracies compared when an Adam optimizer was used of about 79, 75, and 74% on the training, open test, and hidden test sets, respectively, and most relevantly it was able to make the loss function get minimized. It is important to note that the current base model was trained using the original 30 features we got after processing the data, so removing possible noisy predictors was a must before undergoing into hyper-parameters optimization.

4.4.2 Feature Importance Investigation

Our attempt to select the most relevant features for the neural network was based on a ranking of all the features according to the importance given by the extreme gradient boosting gradient model considering it provided the best performance up to this point. Then, we trained the SGD based model several times by adding a set of 5 new features at a time starting from the most important to the least ones. These models were trained for 25 epoch in order to select the set of features that outperforms the others while keeping constant all the parameters included into the neural network structure (See Figure 72). Refer to Figure 62 to see the order of the features included during the process.

This simple, heuristic, time consuming, and probably not highly accurate methodology, leaded us opt for a group of 25 features considering that at the end of the 25th epoch the training and validation accuracies kept growing tendencies and the training and validation losses reached the lowest points and decreased similarly.

4.4.3 Bayesian Optimization

Normally, best parameters selection in any Machine and Deep Learning model is a time consuming and sometimes tedious and sometimes an impossible task. Even though there are some methodologies that might be useful such as Grid Parameter Search, it may be only be consider applicable while optimizing very few parameters, but in cases where the number of hyper-parameter is extended this procedures become costly in terms of computational power and running time.

For instance, imagine we want to optimize 4 hyper-parameters with 10 possible values in each, this means we will have to run 10 to the power of 4 neural network model, which is a massive job and in consequence these type of approaches become less suitable when handling big datasets as in the current case. In the other hand, another common approach is a Random Parameter Search that is normally used to narrow down the possible ranges for the hyper-parameters being optimized; however, if the number of parameters becomes larger, the probability of getting the right combination of them gets very unlikely to get.

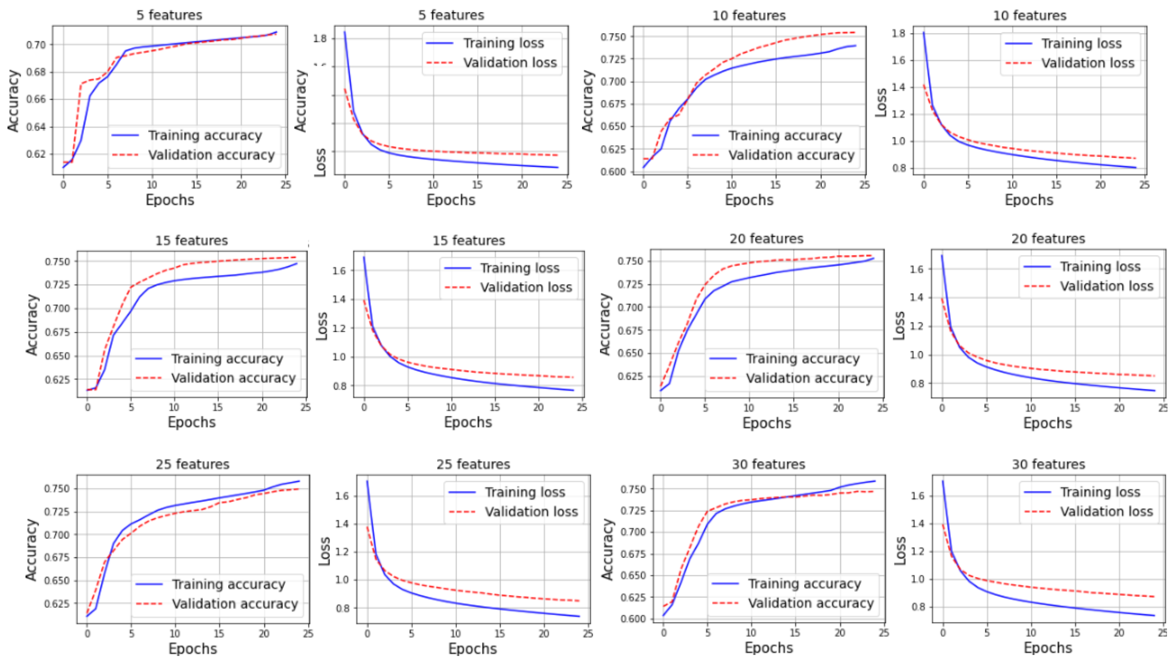


Figure 72 Neural Network: Feature selection.

Consequently, in the present study we propose a hyper-parameters optimization by using an open source library called Scikit-Optimize, which provides an implementation of a Bayesian optimization, where a surrogate model is used to model the search space in order to get an optimal set of hyper-parameters.

The current section attempts to optimize the following hyper-parameters:

1. Learning Rate
2. Number of hidden layers
3. Number of neuron per hidden layer
4. Activation function

Additionally, prior to commence the optimization function that will minimized the complex cost function based on weights and biases involve in the current lithology classification task, each parameter investigation range had to be defined. Table 31 summarizes each parameter's range.

Table 31 Neural Network: Hyper-parameter search space used during the Bayesian optimization

Hyper-parameter	Low boundary	High boundary
Learning Rate	1e-4	1e-1
Number of Hidden Layers	1	5
Number of Neurons	64	512
Activation Function	'relu' or 'sigmoid'	

In order to give a general understanding about what the Bayesian optimization attempts to do, Figure 73 represent the objective function and how skopt intends to find the optimal minimum. The red dotted line represents the true objective function that is surrounded by noise represented by the red shade; every red point represents a sample set of hyper-parameters from the search space and then through a Gaussian process the space between samples is estimated, represented as the green line. In addition, the green shade represents the uncertainty on the approximation given by skopt that normally is caused to the lack of sufficient number of investigated point within that particular range.

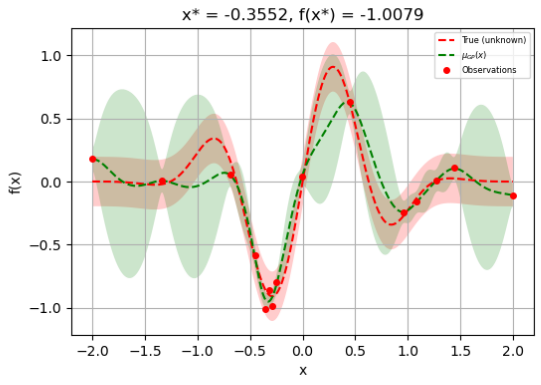


Figure 73 Neural Network: General optimization scheme.

Then, in order to optimized the hyper-parameters through skopt we constructed hyper-parameter optimization wrapper. It is important to note that the neural network in which the optimization will be based is the one based on the SGD optimizer, which previously seemed to be much more stable than the one obtained while implementing an Adam optimizer. In addition, we introduced a momentum into the optimizer in order to take advantage of the knowledge accumulated in previous steps to facilitate the neural network converge faster and much more easily.

Lastly, the Bayesian optimization was executed which aimed to create a loop to evaluate each set of hyper-parameters until the 4th epoch while evaluating the training process with the open test set to provide a much more generalized trained neural network. The Bayesian optimization loop was performed for 75 epochs or calls using different set of hyper-parameters while updating continuously the best performing model accuracy in order compare it with subsequent trained neural networks on different set of hyper-parameters. Figure 74 documented the convergence process of the neural network after each iteration, which for the current stage was stated to be a negative accuracy in order to let skop handle the problem as a minimization exercise. Considering that the optimization process was executed for only 75 epoch due to the Google GPUs usage limitation, it is important to mention that the optimal value reached by the optimization is not necessarily the ultimate optimal value since there may be a better set of hyper-parameters capable to outperform the set selected by the optimizer as the number of epochs raises or the evaluated parameter ranges increase.

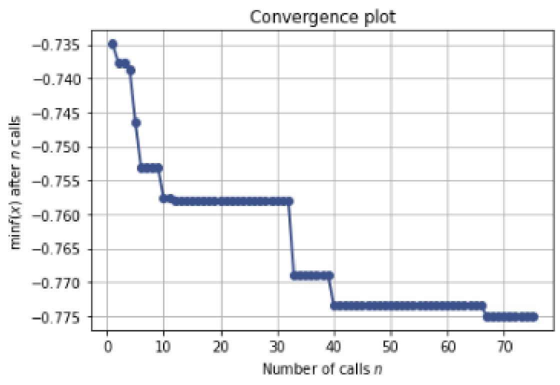


Figure 74 Neural Network: Bayesian optimization neural network convergence.

Figure 74 describes the convergence process after each iteration, and as visible, the validation accuracy could reach values beyond 78%. In addition, as we mention before, the Bayesian optimization uses a surrogate model to model the expensive to evaluate the objective fiction. In other words, the surrogate model aims to provide interpretability to a complex model as the case of neural networks, and it is the surrogate model that is used to determine at which points the objective function will be evaluated at each iteration.

Additionally, Figure 75 show in the diagonal a histogram for each of the evaluated hyper-parameters, while the non-diagonal scatter plots show the spatial location of every evaluated point, where the darker points correspond to the initial evaluated points and the lighter ones reflect subsequent evaluations that tend to cluster around the optimal parameter marked as red. Hence, the histograms' major frequencies are allocated around the optimal hyper-parameter, which implies the optimization performed correctly while looking for the minima.

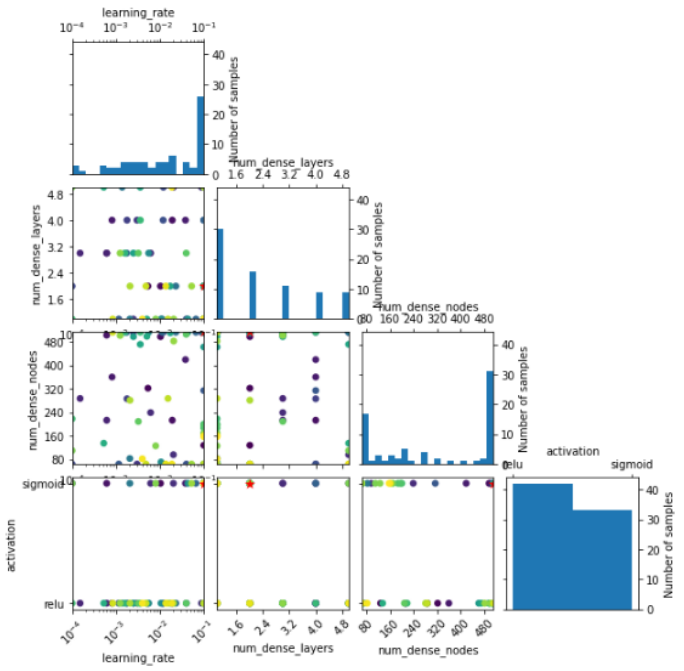


Figure 75 Neural Network: Hyper-parameter evaluation histograms.

Furthermore, Figure 76 shows the partial dependences of the surrogate model for each evaluated hyper-parameter during the Bayesian optimization; in general, partial dependences describes the marginal impact of a particular couple hyper-parameter while holding the other parameters constant. Initially partial dependence plots is a method originally proposed to measure feature importance in gradient boosting based learning machines and were later introduced as a method to measure parameter importance while implementing neural networks.

Moreover, form Figure 76, it is also noticeable that the optimal number of hidden layers oscillates between 1 to 3, smaller learning rates provide higher accuracies while using a relu activation function and larger number of hidden layers when using a sigmoid activation function, the model optimized better while a high number of hidden neurons was used. Further, it has to be noted that the partial dependence is merely based on the surrogate model which just provides an approximation of the objective function, and hence it might not be a good representation of the objective in places where less number of samples were evaluated an far from the location were the minima was found.

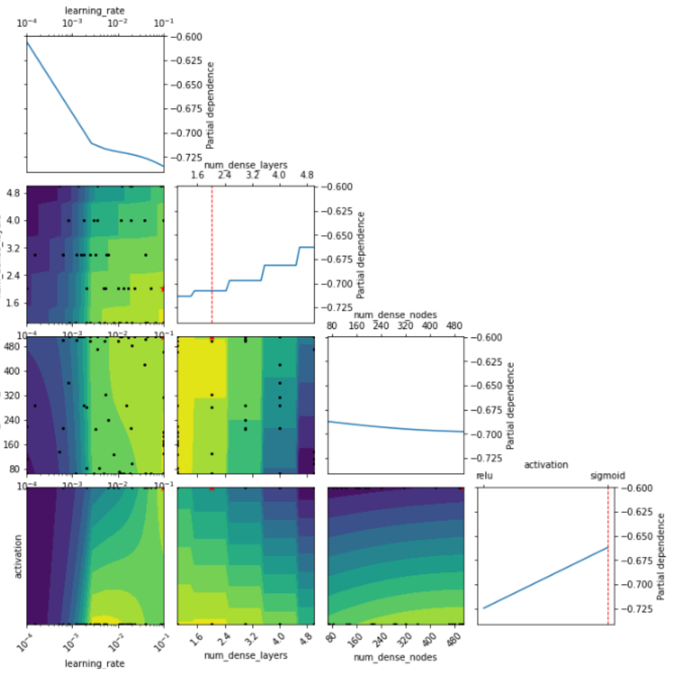


Figure 76 Neural Network: Hyper-parameter two-dimensional partial dependence.

Finally, a new neural network was trained based on the hyper-parameters found by the optimizer after 75 iterations. Table 32 summarizes the optimal hyper-parameters.

Table 32 Neural Network: Optimal hyper-parameters after running the optimization for 75 epochs.

Hyper-parameter	**Best Value** (After 75 epochs or calls)
Learning Rate	0.1
Number of Hidden Layers	2
Number of Neurons	512
Activation Function	sigmoid

Further, since the model was overfitting immediately after the 7th epoch, we introduced two dropout regularization layers before each hidden layer, this helped to train the network longer and reduce the loss function. The optimized model training accuracy and loss evolution is documented on Figure 77 where the training accuracy increases beyond 80% while the validation accuracy plateaus slightly above 77%.

In the other hand, the loss function decreased smoothly for the training and validation until the 30th epoch, then the validation loss started to increase again. This implies that the optimized neural network is unable to provide test accuracies beyond 77% and from a certaing point it starts learning patterns only present and applicable to the training data. For more details about the Bayesian, optimization refer to Appendix C where the complete optimization algorithm is described.

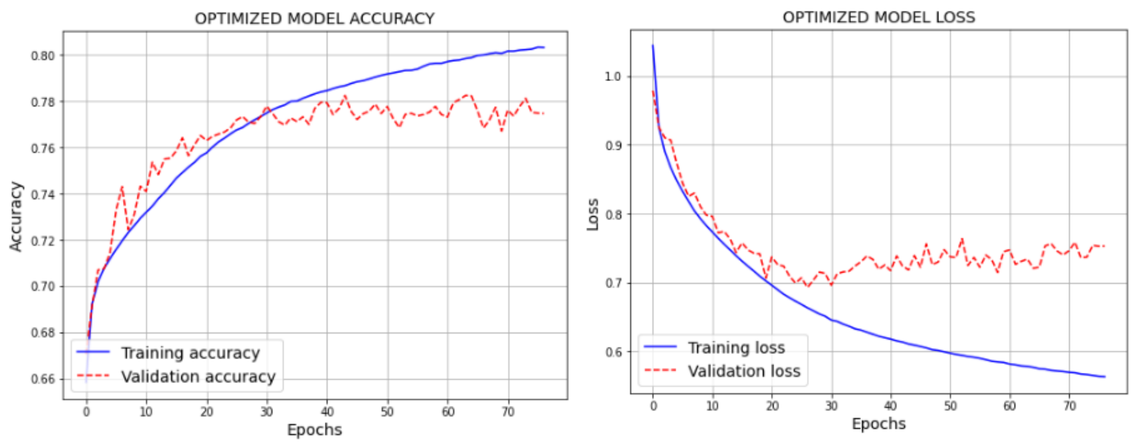


Figure 77 Neural Network: Optimized model accuracy (left) and loss function (right) training history.

A detailed classification report for each dataset is presented on Table 33 where the accuracy reached by the optimized neural network showed performances of 83, 77, and 77% on the training, open test, and hidden test datasets, respectively.

Table 33 Neural Network: Classification reports for the training, open test, and hidden test datasets

NEURAL NETWORK CLASSIFICATION REPORT									
Class	**Training Set**			**Open Set**			**Hidden Set**		
	Prec.	*Rec.*	*F1.*	*Prec.*	*Rec.*	*F1.*	*Prec.*	*Rec.*	*F1.*
Sandstone (0)	0.82	0.68	0.74	0.85	0.75	0.80	0.82	0.66	0.73
Sandstone/Shale (1)	0.67	0.39	0.40	0.47	0.23	0.31	0.44	0.19	0.26
Shale (2)	0.82	0.97	0.89	0.80	0.95	0.87	0.82	0.95	0.88
Marl (3)	0.68	0.49	0.57	0.66	0.06	0.11	0.30	0.22	0.26
Dolomite (4)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Limestone (5)	0.76	0.60	0.67	0.44	0.48	0.46	0.55	0.60	0.57
Chalk (6)	0.87	0.59	0.70	0.00	0.00	0.00	0.60	0.50	0.55
Halite (7)	0.96	0.99	0.98	-	-	-	0.98	1.00	0.99
Anhydrite (8)	0.86	0.80	0.83	1.00	0.07	0.13	0.97	0.38	0.54
Tuff (9)	0.73	0.55	0.62	0.71	0.61	0.66	0.62	0.50	0.55
Coal (10)	0.85	0.38	0.53	0.88	0.37	0.52	0.85	0.54	0.66
Basement (11)	0.00	0.00	0.00	-	-	-	-	-	-
Weighted Metric	0.79	0.81	0.79	0.75	0.77	0.74	0.75	0.77	0.75
Accuracy Score	0.81			0.77			0.77		
Matrix Score	-0.511			-0.594			-0.563		

Finally, for a better understanding of the neural network classification, the confusion matrices normalized by the total number of predictions per class are presented on Figure 78. The main observation from it is that the neural network achieved good accuracies on both test sets; however, its accuracy is dramatically affected while classifying carbonates, same that were greatly misclassified as the case of chalk, while dolomites were not even predicted at all.

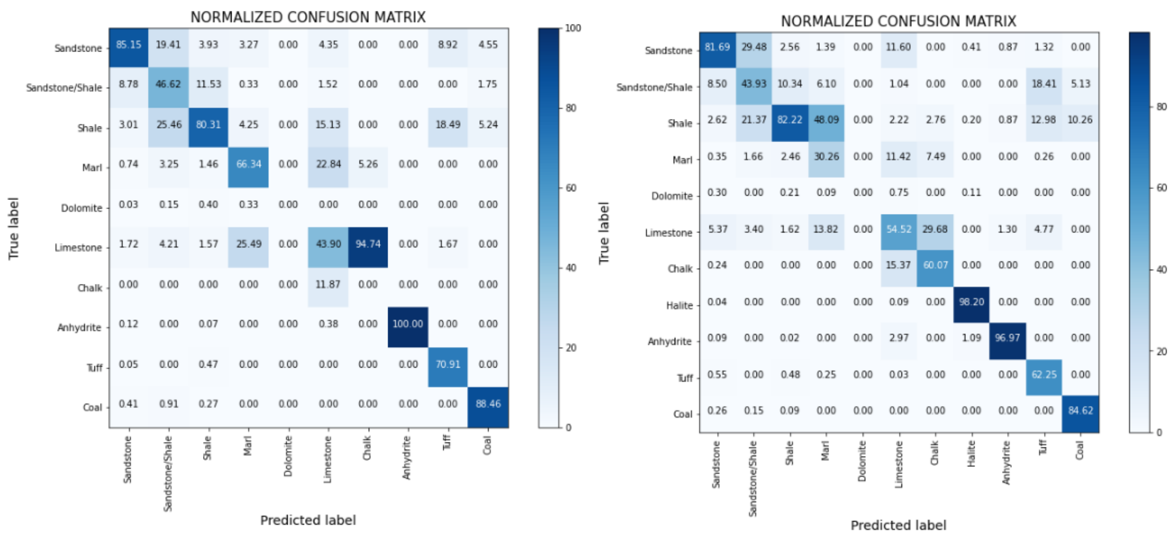


Figure 78 Neural Network classifier: Classification confusion matrices normalized by the number of predictions by class (a) Open test set, (b) Hidden test set.

Chapter 5

5. PERFORMANCE COMPARISION

Once the machine-learning models construction, hyper-parameter optimization, training, validation, and testing stages have been finished, we are ready to present and compare the machine-learning modes' global performances while solving the lithofacies classification problem. Besides, it is important to consider that each model went into different feature selection and hyper-parameter optimization techniques; thus, not every model used the same number or set of features to provide their optimal results.

First, Table 34 summarizes the classification scores all the optimized algorithms obtained on the hidden test set. From this table we can observe that the tree-based gradient boosting (GBDT) achieved greater results over neural networks, decision trees-based, and traditional stand-alone machine learning algorithms. In addition, GBDT do not only offer higher accuracy, precision, recall, f1-score classification scores, but also lower FORCE penalization scores. This infers that GBDT algorithms perform more consistently even from a petrophysicist perspective, which was the purpose the FORCE scoring matrix was built for.

Table 34 Machine-learning models performance comparison: Hidden test set.

Algorithm	Acc	Prec	Rec	F1 Score	M. Score
Extreme Boosting	82.52	81.54	82.52	81.74	-0.43
Categorical Boosting	81.38	80.16	81.38	80.36	-0.45
Light Boosting	80.39	79.01	80.39	79.00	-0.48
Random Forest	79.82	77.29	79.82	77.56	-0.50
Support Vector Machines	79.08	76.86	79.08	77.16	-0.54
K-Nearest Neighbors	78.22	76.31	78.22	76.41	-0.56
Neural Networks	77.41	74.61	77.41	74.99	-0.56
Logistic Regression	75.06	71.44	75.06	72.42	-0.64
Decision Tree	74.59	70.40	74.59	68.54	-0.67

Moreover, whether we analyze in detail the total number of predictions each model produced per each lithology class, we can easily observe how for the most frequent classes in the training and hidden test sets such as shale, sandstone, shaly-sandstone, and limestone, every model achieved a quite balanced number of predictions with exemption of the decision tree DT, which was produced by the pruning process DT went through. However, if we see closer into the number of wrong predictions every model provided, we could better see how tree-based gradient boosting algorithms misclassify less instances as any of the most frequent lithologies.

In addition, even though GBDT models perform better and present less bias towards the most frequent classes than the other models, there is still a visible tendency to misclassify other lithologies as shale, same that is particularly caused by the massive number of shale instances present in the training dataset, 61.6%. Besides, apart from sandstone, shale, and limestone, for which several models presented high classification accuracies, it is when it comes about shaly-sandstone classification where GBDT models distance themselves from the other models followed closely by K-Nearest Neighbors and Neural Networks (See Figure 79).

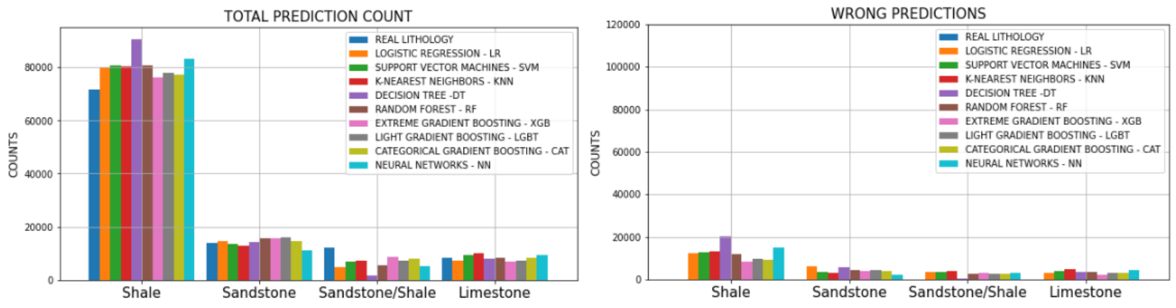


Figure 79 Hidden test set prediction histograms: Total predictions count (left) and wrong predictions counts (right) for shale, sandstone, shaly-sandstone, and limestone.

Furthermore, while classifying the medium-frequency classes such as halite, marl, chalk, and tuff, which together represent only 5.7% of the complete training dataset, every model appeared to perform at high level while classifying halite although only very few halite instances are represent in the training dataset, 0.7%. Besides, chalk and tuff appeared to be under or over misclassified moderately by most of the models; however, GBDT models and Neural Network appeared to be able to capture and classify these lithologies in much more accurate manner although chalk and tuff may have been underrepresented on the training dataset, 0.9% and 1.3%, respectively. In the other hand, marl appeared to be highly misclassified as shale or limestone by all the models, which in not surprising considering that marl is a sedimentary rock composed mainly of clay and lime, which makes it hard to properly define a proper boundary between these three classes (See Figure 80).

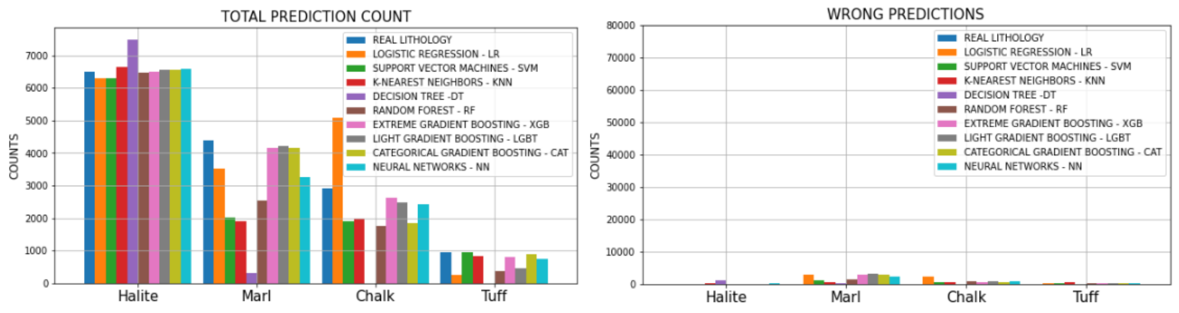


Figure 80 Hidden test set prediction histograms: Total predictions count (left) and wrong predictions counts (right) for halite, marl, chalk, and tuff.

Likewise, the models provided great discrepancies while classifying the least frequent classes, which together represent only 0.5% of the training set. As visible, anhydrite is moderately well classified by RF, CAT, and XGB models, while the other models were not able to classify it correctly in more than 50% of the cases. Although coal represents only 0.3% of the training set, this number of instances was enough to provide GBDT algorithms with the information needed to classify it correctly in about 80% of the cases. In contrast, dolomite was the lithology class every model struggle with the most, which from our perspective is directly linked and caused by the number of instances used for training and the similarity in wireline response that dolomite has when compared to other classes such as limestone, chalk and marl, which hinders its proper classification (See Figure 81).

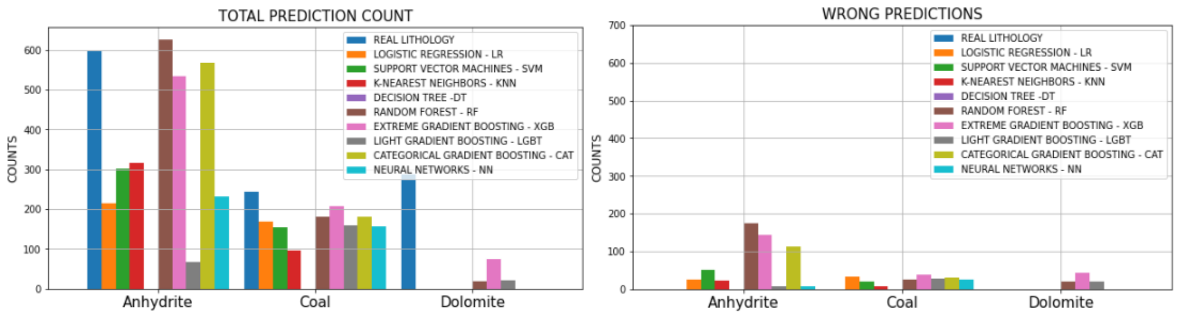


Figure 81 Hidden test set prediction histograms: Total predictions count (left) and wrong predictions counts (right) for anhydrite, coal, and dolomite.

Complementary, Table 35 summarizes the classification scores obtained by the optimized model on the open test set, same that clearly shows similar results to the ones achieved on the hidden test set (See Table 34). However, it is important to note that there are slight differences between the performances each model obtained on the open and hidden test sets caused mainly by the variability on the lithology distributions each test set holds. For instance, the hidden set has many more halite data point than the open test set, and considering that most of the models perform similarly at high level while classifying this particular lithology, the prediction on the hidden set obtains much more improvement from this particular class on the global accuracy score. Something similar happens if we refer to the higher number of shaly-sandstone instances the open test set holds in comparison to the hidden test set, which considering the difficulties every model faces while predict this mixed-based lithology, the global open test set accuracy gets much more affected by this distribution dissimilarity.

Table 35 Machine-learning models performance comparison: Open test set.

Algorithm	Acc	Prec	Rec	F1 Score	M. Score
Categorical Boosting	80.02	78.19	79.91	77.22	-0.52
Extreme Boosting	80.00	77.31	79.61	77.06	-0.53
Light Boosting	79.36	76.81	79.36	76.98	-0.53
Random Forest	77.71	74.84	77.71	75.05	-0.58
K-Nearest Neighbors	77.53	74.19	77.53	74.44	-0.59
Neural Networks	77.34	74.58	77.34	74.32	-0.59
Support Vector Machines	76.10	73.41	76.10	73.79	-0.62
Decision Tree	74.55	69.67	74.55	67.14	-0.69
Logistic Regression	71.54	67.77	71.54	69.11	-0.75

Finally, following the same logic we used to analyze the results obtained on the hidden test set, the open test set results show similar nature in regard of the classification bias towards the majority classes in particularly to shale; besides, the classification becomes problematic when it comes to shaly-sandstone and marl, and deficient for dolomite. Refer to Appendix G to see the open test set classification histograms.

Secondly, in regards of the implemented machine learning imputation procedure we included as an attempt to improve the classification performance, Table 36 summarized the impact the imputation had on the XGB model's classification performance when compared to the results achieved without implementing such methodology. Additionally, it allows us to observe that the improvement we initially achieved on a logistic regression model (See Table 12) by implementing the proposed imputation technique remains similarly while implementing much more robust algorithms like GBDT models as in the case of XGB. However, it is necessary to mention that although the presented imputation technique provided clear classification enhancements, it did not provide much larger improvements while using complex algorithms as we expected initially.

Table 36 Feature augmentation and engineering impact on the best performing model - XGB.

XGB MODEL	Features	Accuracy		
		Training %	Test %	Hidden %
Base Model	Median Imputed (27)	83.0	78.0	76.0
Tuned Model 2	Median Imputed + Additional Features (30)	89.7	79.4	80.1
Tuned Model 4	Augmented Features + Additional Features (28)	87.0	80.0	82.5

Moreover, different random imputation orders were initially tested to overcome the data sparsity effect on the lithology classification, all of which provided higher error measurements on the test sets when the four investigated wireline logs, DTC, DTS, NPHI, and RHOB were predicted and evaluated. In consequence, the ascending priority ranking approach we designed and proposed, based on our petrophysical experience about which specific wireline logs play the most important role for lithology interpretation purposes and the dataset completeness available for training, provided lower root mean squared errors and thus better results than any other random imputation order we tested. However, considering that, each model supports its performance on different sets of features, a much more consistent and robust approach could be to impute the wireline logs in a consistent order according to the treated model, data completeness, and prioritizing based on the feature importance provided by the model itself and petrophysical experience.

Third, along with the promising and relative high results some machine-learning algorithms offer to solve the lithofacies classification problem, most of the models exhibited great difficulties to properly classify carbonates in particular marl, dolomite, and limestone as well as shaly-sandstone. However, the question that arises is until what extent these misclassifications could be consider acceptable from a geological perspective or if these misclassifications are actually mistaken. In consequence, now we present a closer view into some particular examples where this questioning provides interesting observations and answers. In consequence, in order to try to give an answer to that question it is necessary to put ourselves in perspective about which were the wells that the models struggle the most to predict accurately. Table 37 presents a performance report per each well present on both the open and hidden test sets, from which we will initially concentrate most of our discussion on the particular wells that presented most of the difficulties to be accurately classified by the best performing model XGB.

Table 37 Extreme gradient boosting model's performance on each well present on the open test and hidden test sets – low performance wells highlighted.

OPEN TEST SET							
Well	**Interpreter**	**Cluster**	**Acc**	**Rec**	**Prec**	**F1 Score**	**M. Score**
34/3-3 A	EXP3	2	0.94	0.94	0.92	0.93	-0.16
25/5-3	EXP1	0	0.88	0.88	0.86	0.86	-0.32
29/3-1	EXP1	2	0.85	0.85	0.84	0.84	-0.40
34/10-16 R	EXP1	2	0.85	0.85	0.83	0.83	-0.38
25/10-10	EXP1	0	0.83	0.83	0.82	0.81	-0.54
35/6-2 S	EXP2	1	0.74	0.74	0.69	0.70	-0.69
34/6-1 S	EXP3	2	0.72	0.72	0.74	0.72	-0.67
25/11-24	EXP1	0	0.70	0.70	0.70	0.60	-0.99
35/9-8	EXP2	1	0.66	0.66	0.63	0.63	-0.85
15/9-14	EXP1	0	0.57	0.57	0.55	0.46	-1.02
HIDDEN TEST SET							
Well	**Interpreter**	**Cluster**	**Acc**	**Rec**	**Prec**	**F1 Score**	**M. Score**
34/3-2 S	IG2	2	0.90	0.90	0.85	0.87	-0.27
31/2-10	EXP1	1	0.89	0.89	0.90	0.89	-0.31
35/11-5	EXP2	1	0.86	0.86	0.84	0.84	-0.34
31/2-21 S	IG2	1	0.83	0.83	0.82	0.83	-0.41
16/7-6	EXP3	0	0.83	0.83	0.76	0.79	-0.40
35/9-7	EXP2	1	0.82	0.82	0.77	0.79	-0.48
15/9-23	EXP3	0	0.82	0.82	0.81	0.77	-0.42
16/2-7	EXP1	0	0.81	0.81	0.80	0.80	-0.50
25/10-9	EXP1	0	0.79	0.79	0.75	0.76	-0.56
17/4-1	EXP1	0	0.61	0.61	0.71	0.63	-0.90

Based on the report presented above, we can easily observe that XGB struggled more to properly predict lithologies on the open set wells, reason why we achieved a slightly higher performance on the hidden set. Further, along with some mistakes most of the models have to classify carbonates and shaly-sandstones, there are also some ambiguities involved on the provided interpretation that must be noted in order to have a fair comparison between the performances achieved by machine learning and a human interpreter. For instance, the predictions that the top five best performing machine-learning models obtained on well 16/2-7 that belongs to the open test set showed a general agreement on the main predicted lithology trends (Figure 82); however, there are some particular intervals where there is a conflict between the predicted lithologies and the interpretation given by the data provider. One of these intervals goes from 1500 to 1950 meters (interval 1), where most of the models seem to have misclassified marl as limestone or shale. However, if we consider the inherent nature of marl, which is a mixed rock composed of clay and lime, we could say that making these misclassifications is totally permissible not only for a machine learning model but even for an experienced petrophysicist.

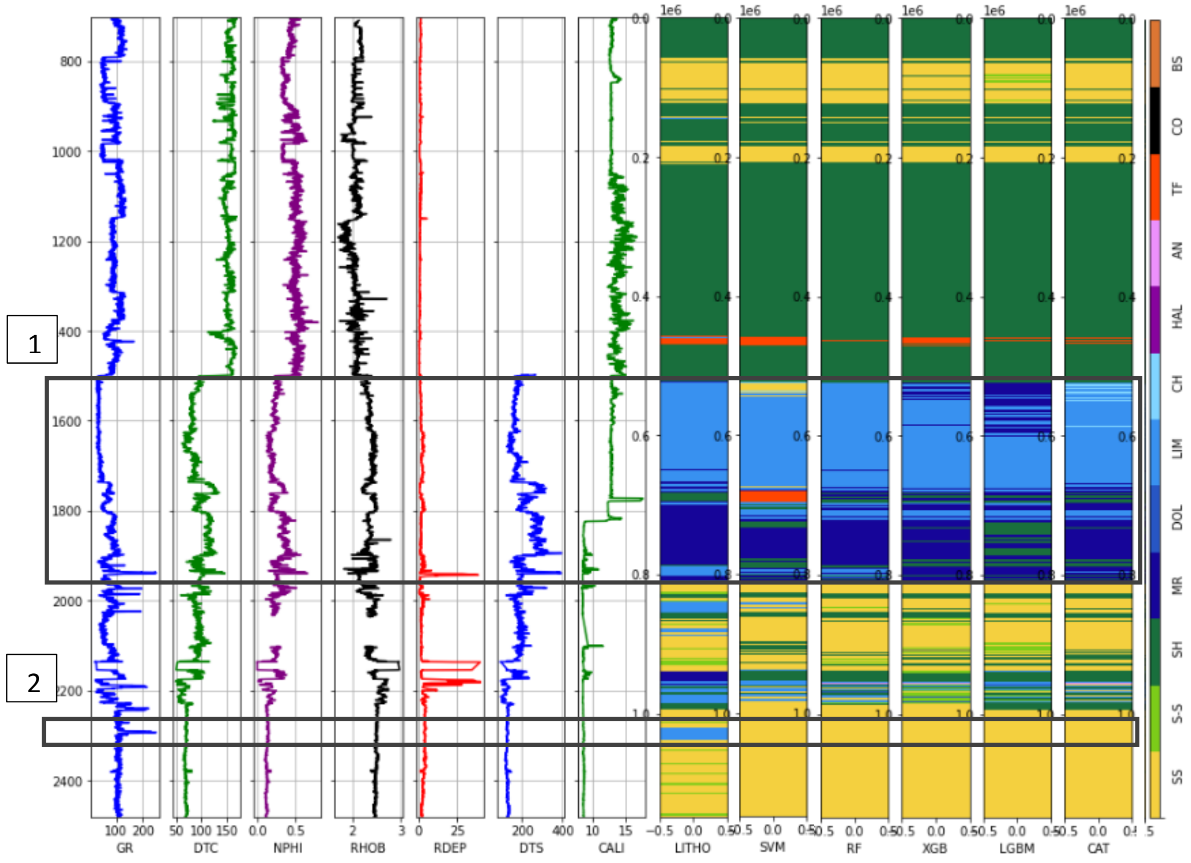


Figure 82 Prediction analysis well 16/2-7

Additionally, there is another conflictive, less extensive, but much more interesting interval in the same well, 16/2-7, which goes from 2285 to 2315 meters (interval 2), showed that even though the interpreter characterized it as a limestone interval, none of the top performing models was able to classify that interval as limestone but as sandstone. However, when core images were studied, they revealed that the section actually consists of conglomerate and breccia (See Figure 83), same that although are strongly correlated to sandstone, they are technically different from each other due to the grains size they compromise (NPD, 2021). Thus, it implies that the machine learning models classified this interval more accurately, which is something that has to be considered as advantage that machine learning provides against an standard human interpretation.

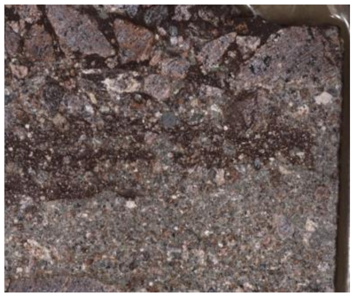


Figure 83 Well 16/2-7, core taken within interval from 2285 to 2315 meters.

In addition, there is a second interesting observation that comes from well 15/9-14 belonging to the open test set, in which most of the models provided high quality prediction with regards of sandstone and limestone lithotypes; however, there is a visible bias to over classify shale as we discussed beforehand. Initially, it appeared that the shaly-sandstone lithology identification was highly affected by the lack of the shear sonic log, which we attempted to overcome by implementing machine learning for missing values imputation. However, once we studied the feature importance that the GBDT models provided for that specific lithotype, we observed that DTS just appeared as the 11th position of the features that contribute the most to its proper identification (Figure 67b). In consequence, the poor ability every model has to accurately map shaly-sandstone seemed to be linked to the way how the data was normalized before training the classifiers, specially the GR log.

In other words, if we have a look to the way the gamma ray log responds according to the well location (See Figure 85), we can presume that when we standardize the data as a unique dataset, we are likely to lose sensitivity to distinguish between shaly-sandstones, and shale since during interpretation the base line for the last is normally defined according to the subjectivity of the interpreter, which in turns depends on the well's geological location. Therefore, based on Table 37, where most of the problematic wells belong to location cluster zero which in turns are linked mostly to wrong shaly-sandstone predictions, we could say that there is a great effect on the models' ability to map such lithology due to the interpreter's subjectivity, which is generally introduced while defining the shale baseline during well log interpretation.

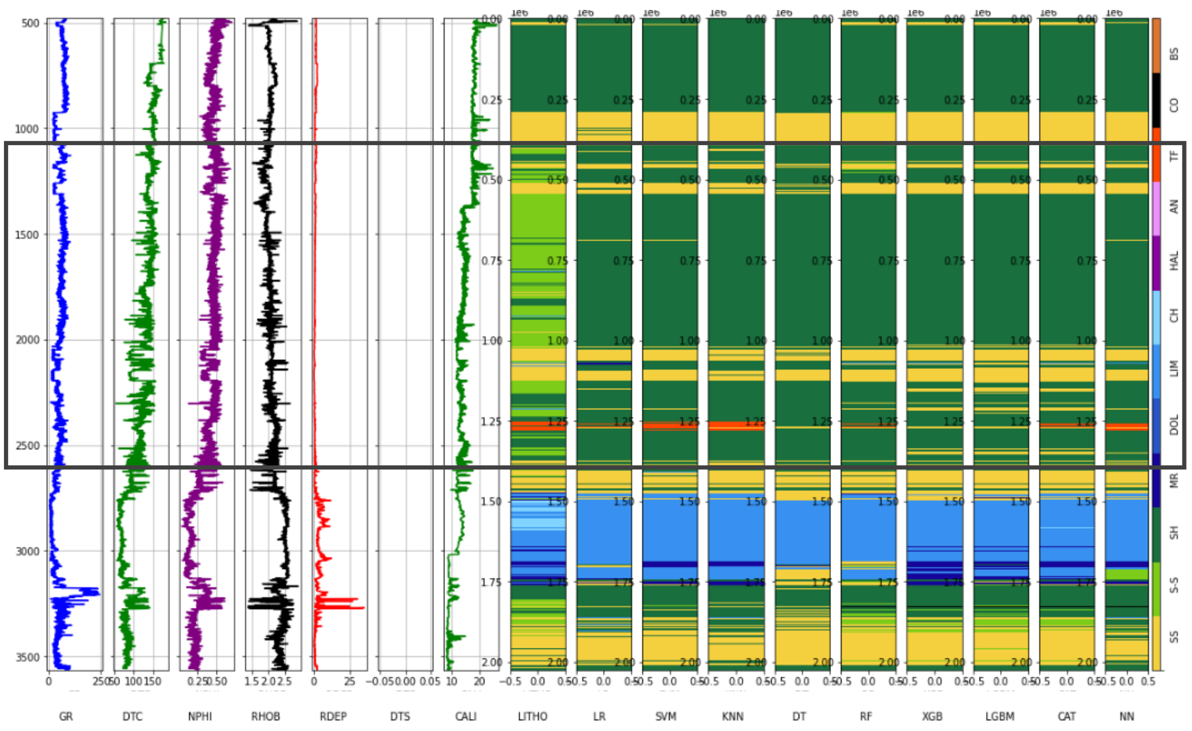


Figure 84 Prediction analysis well 15/9-14

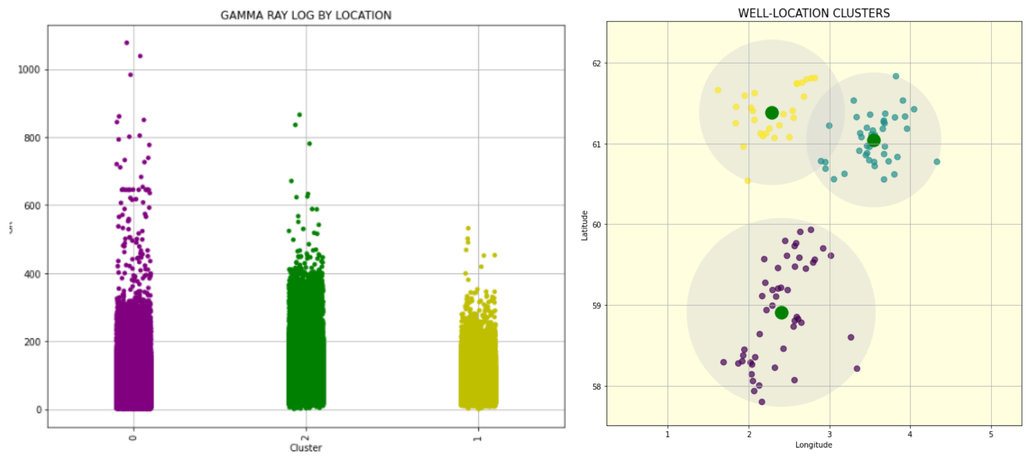


Figure 85 Gamma ray log response according to well location

Moreover, some other observations are visible on Figure 89 from which we could note that most of the models tend to misclassify chalk as either limestone or marl, which as stated above could be considered permissible mistakes. However, although GBDT models tend to confuse limestone with marl and dolomite similarly as the other models, they offer a much more robust ability to classify carbonates when they are surrounded by different types of lithologies. Refer to Appendix F to visualize the classification results on the open and hidden datasets.

In fact, GBDT algorithms are able to provide a detail-oriented performance due to their capability to map sandstone, tuff, anhydrite, coal, and most importantly carbonate thin beds, last of which may be of particular importance in unconventional reservoirs considering that those laminations play a crucial role on hydro-fracturing acting as limitations for fracture propagation and consequently reservoir productivity.

On the other hand, tempted by the idea that interpreters' subjectivity could also affect in great degree the performance of learning machines, a sensitivity analysis was executed by implementing the best performing and fastest model XGB when different sets of data coming from different interpreters were used for training and testing purposes.

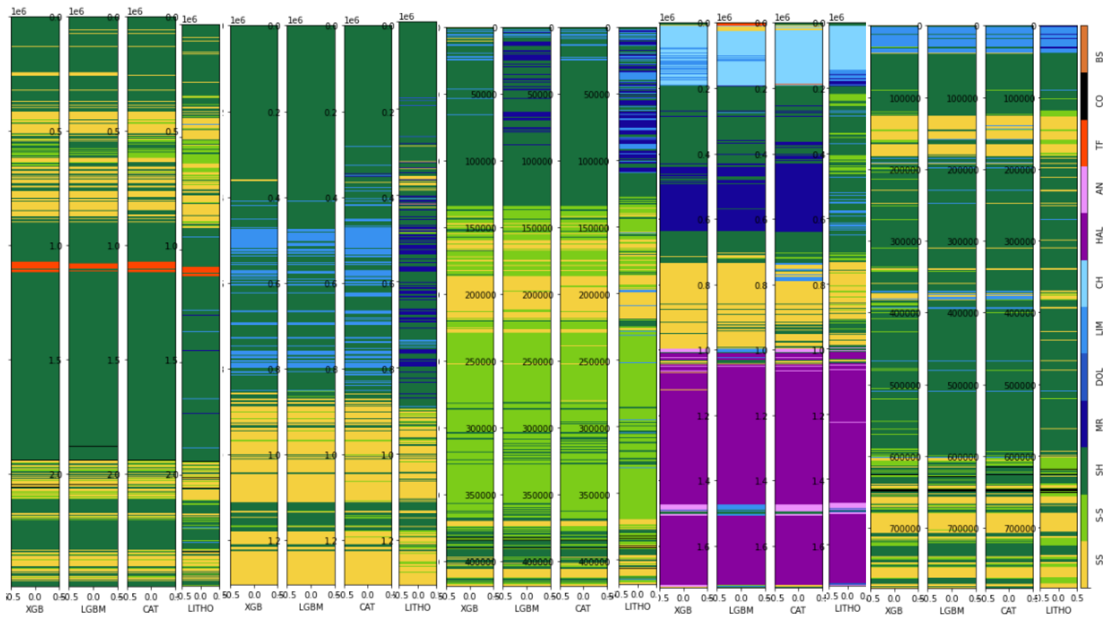


Figure 86 Prediction analysis wells 34/10-16R (a), 35/6-2S (b), 35/9-8 (c), 17/4-1 (d), and 31/2-21S (e).

The information regarding interpreters was provided by Peter Bormann, the FORCE competition organizer, and Table 38 records how XGB performs whether we vary the datasets we used. Additionally, it is necessary to bring into the discussion the fact that the FORCE datasets were provided by two different sources, 83 wells from Explocowd and 15 from IG2; besides, Explocrowd's data was interpreted by three different groups of interpreters which for practical purposes we will call EXP1, EXP2, and EXP3 from now on.

Table 38 Interpreter subjectivity analysis. An XGB classifier was trained several times by keeping a particular set of wells from a specific interpreter and then tested on the wells provided by other interpreters on the open and hidden test datasets.

Accuracy % obtained by XGB on wells provided by different interpreters								
Training dataset	Open test set				Hidden test set			
	EXP1 (6 wells)	**EXP2** (2 wells)	**EXP3** (2 wells)	**IG2** (0 wells)	**EXP1** (4 wells)	**EXP2** (2 wells)	**EXP3** (2 wells)	**IG2** (2 wells)
EXP1 (49 wells) 88.0	78.0	29.0	89.0	-	78.0	68.0	75.0	88.0
EXP2 (23 wells) 91.0	60.0	70	85.0	-	51.0	77.0	69.0	78.0
EXP3 (11 wells) 89.0	54.0	37.0	73.0	-	49.0	58.0	74.0	69.0
IG2 (11 wells) 97.0	60.0	32.0	82.0	-	42.0	65.0	70.0	81.0

High (accuracy >80)	Medium (60<accuracy<80)	Low (accuracy<60)
🟩	🟦	🟧

The first group of interpreters from Explocrowd provided 49 wells, the second one 23 wells, and the third one 11 wells. The idea of the sensitivity analysis consisted on training XGB on a set of wells belonging to a particular interpreter and then testing the classifier's performance on the other interpreter's wells from the open and hidden test datasets, so we could quantify the interpreter's subjectivity influence on the performance and the possible dissimilarities between interpretations.

The extreme gradient boosting model was trained for 100 epochs without including any regularization technique, meaning that most likely it overfitted the training data in every case, however, by comparing how much the training and test accuracies differ from each other is in general the only way how we could understand any possible inconsistency between interpretations given massive size of the datasets.

Table 38 summarizes the results we obtained, from which we can visualize that when the model is trained based on the wells interpreted by either EXP1 or EXP2, the model was able to provide medium-high accuracies on the wells provided by other interpreters, meaning that there is a good consistency between the them and the others' interpretations. However, XGB's performance presents an important and visible drop on the accuracies when only the wells interpreted by EXP3 are used during training, providing only medium to high results when they are tested on the test wells provided by the same interpreter. This suggests that excluding the wells interpreted by EXP3 from the training set may improve the global classification performance.

In addition, if we look Table 38 in the vertical direction we can also observe how the classifier in some cases was unable to perform at high level when it was tested on the wells interpreted by EXP2 and EXP1 regardless of the data used for training the model. However, it does not mean that all the wells provided by EXP2 or EXP1 went into difficulties to be precisely classified, but it does mean that when the wells contain an important amount of mixed sediments, especially shaly-sandstone, the model finds great difficulties to do a proper work as in the case for wells 34/6-1 S, 25/11-24, and 15/9-14. Therefore, this analysis reinforces our first conjecture regarding the role the interpreters' subjectivity plays into the classifier performance in special when it comes to properly classify shaly-sandstones.

Chapter 6

6. CONCLUSIONS, AND FUTURE ENHACEMENTS

6.1 Conclusions

- In the current study, the performances of stand-alone standard classifiers, random forest (RF), generalized boosting machines (GBM), and neural networks (NN) were compared for the lithofacies classification problem by using the FORCE competition dataset. Generally, the highest performances were given by decision trees-based generalized boosting machines, which accomplished to outperform standalone classifiers, standard ensemble models, and even much more complex structures such as neural networks. GMB produced better performances mainly while classifying the minority and mineral-mixed lithofacies, meaning that they are able to provide a much more detail-driven lithology classification.

- Generalized boosting machines (GBM) proved to be highly robust, powerful, efficient, and overall scalable machine learning algorithms perfectly suitable to deal with large, imbalance, and sparse datasets. In addition, their compatibility with either CPUs or GPUs as opposed to the other studied algorithms makes it possible optimizing the model hyper-parameters manually in a matter of minutes. Hence, GBM are almost a perfect blend of software and hardware capabilities designed to enhance the pre-existing boosting techniques in terms of training time and efficacy.

- By comparing the performances achieved by the base line models and the optimized ones, we could categorically conclude that the efficiency of any leaning machine is able to provide depends importantly upon a proper and efficient feature and hyper-parameter selection along with other important processing steps such outlier identification, data standardization, feature augmentation, and feature engineering. In addition, including an extensive cross validation technique while training the learning machines provided the best results as the model avoids overfitting the training data and thus improves generalization.

- The implemented machine learning-based feature augmentation on the DTS, NPHI, RHOB, and DTC logs along with the addition of new features proved to provide a small but still important enhancement on the classification, most remarkably on the hidden test dataset rather than in the open test dataset, difference that is originated mainly due to the dissimilarity on lithologies distributions each test dataset holds. In addition, in regards of the feature augmentation process, there is a genuine need to study the proposed approach in a much more detailed manner in order to measure the uncertainty that might be introduced into the datasets by implementing machine-learning-based imputation techniques in highly sparse datasets, especially when dealing with big and continuous missing value gaps.

- After testing several approaches to properly clean and process the datasets, improve the quality of the data by machine learning implementation, define, optimize, train, and test several an diverse machine leaning algorithms, and post-process the predictions by using the predicted class probabilities without having further improvements beyond the boundary of 82.5% of accuracy, we could conclude that the missed accuracy in about 17.5% derives from the uncertain nature of the datasets themselves. This uncertainty seems principally to come from the subjectivity that petrophysicists include when interpreting wireline logs, which in turns depends upon the geological location that is being studied and the expertise of the interpreter. Therefore, having a large but more importantly consistent dataset are the two most relevant factors that could guarantee to obtain the best possible outcome while implementing machine learning to classify lithofacies.

- In general, all the models faced more difficulties to accurately classify shaly-sandstone, marl, and dolomite. The first two seemed to be linked to the interpretation subjectivity as they are normally misclassifies as shale, which is not surprising given their mineralogical composition, while the third one seems to be linked to the low number of data instances available for training. In fact, even though the top performing generalized boosting machine algorithm, XGB, provided the highest accuracy on unseen objects, individually speaking there were wells in which XGB performed at higher level of precision when compared to the global accuracy of 82.5%, reaching values up to 94%. However, there were also wells that seemed to be complicated for XGB to be properly classified reaching individual accuracies up to 57%, which in turns worsened the global accuracy that could have been achieved. Consequently, considering the poor accuracy in some particular wells seems to be linked mainly to shale and shaly-sandstone differentiation, further analysis is required in order to better understand and overcome such challenge.

6.2 Future enhancements

As extensively discussed in the current study, there is a great need to find a better way to separate shaly-sandstone and carbonates adjacent lithofacies. One initial way to overcome the current challenge could be by normalizing the datasets based on their geological location, especially the GR log, so that we preserve every interpreter's subjectivity without being affected by the others' interpretations during data normalization. Besides, the same logic could be followed once some other additional features are created such as volume of clay or shale index logs. Second, a stacking or voting machine learning model could be constructed base the other model's predictions in order to have an agreement between each other and thus incorporate the predictions at which the other model may be better at. Third, incorporate inherent geological spatial continuity by developing either variograms, correlograms, or coefficient of variations of the most relevant wireline logs so that we can quantify heterogeneity and connect the prediction along the y-axis, aiming in this way to correct wrong isolated interpretations. Fourth, quality check the petrophysical interpretations hold by the datasets especially in wells with the lowest accuracies so we can base future analysis in a much more consistent set of data. Finally, trying novel techniques in machine-learning specially designed to identify anomalies within the data such as wavelet transformation, which normally intends to capture data variations at different scales by extracting both spectral and temporal information from wireline logs may help capture the major lithology trend in the subsurface but also the minor details within it.

7. REFERENCES

Akatsuka, K., 2000, 3D Geological Modeling of a Carbonate Reservoir, Utilizing Open-Hole Log Response - Porosity & Permeability - Lithofacies Relationship: OnePetro, doi:10.2118/87239-MS.

Al-Anazi, A., and I. D. Gates, 2010, A support vector machine algorithm to classify lithofacies and model permeability in heterogeneous reservoirs: Engineering Geology, v. 114, no. 3, p. 267–277, doi:10.1016/j.enggeo.2010.05.005.

Anifowose, F. A., J. Labadin, and A. Abdulraheem, 2017, Ensemble machine learning: An untapped modeling paradigm for petroleum reservoir characterization: Journal of Petroleum Science and Engineering, v. 151, p. 480–487, doi:10.1016/j.petrol.2017.01.024.

Arabameri, A., W. Chen, M. Loche, X. Zhao, Y. Li, L. Lombardo, A. Cerda, B. Pradhan, and D. T. Bui, 2020, Comparison of machine learning models for gully erosion susceptibility mapping: Geoscience Frontiers, v. 11, no. 5, p. 1609–1620, doi:10.1016/j.gsf.2019.11.009.

Awad, M., and R. Khanna, 2015, Efficient Learning Machines: Theories, Concepts, and Applications for Engineers and System Designers: Apress, Berkeley, CA, XIX, 268 p.

Bonaccorso, G., 2020, Mastering Machine Learning Algorithms: Expert Techniques for Implementing Popular Machine Learning Algorithms, Fine-Tuning Your Models, and Understanding How They Work.: Packt Publishing Ltd, 576 p.

Breiman, L., J. Friedman, C. J. Stone, and R. A. Olshen, 1984, Classification and Regression Trees: Taylor & Francis, 372 p.

Chen, T., and C. Guestrin, 2016, XGBoost: A Scalable Tree Boosting System, *in* Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining: MIT Press, p. 785–794, doi:10.1145/2939672.2939785.

Criminisi, A., J. Shotton, and E. Konukoglu, 2011, Decision forests for classification, regression, density estimation, manifold learning and semi-supervised learning: Microsoft Research Cambridge, Tech. Rep. MSRTR-2011-114, v. 5, no. 6, p. 12, doi:10.1561/0600000035.

Defazio, A., F. Bach, and S. Lacoste-Julien, 2014, SAGA: A Fast Incremental Gradient Method with Support for Non-Strongly Convex Composite Objectives: Proceedings of the 27th International Conference on Neural Information Processing Systems, v. 1, p. 1646–1654.

Dubois, M. K., G. C. Bohling, and S. Chakrabarti, 2007, Comparison of four approaches to a rock facies classification problem: Computers & Geosciences, v. 33, no. 5, p. 599–617, doi:10.1016/j.cageo.2006.08.011.

Ghori, K. M., R. A. Abbasi, M. Awais, M. Imran, A. Ullah, and L. Szathmary, 2019, Performance Analysis of Different Types of Machine Learning Classifiers for Non-Technical Loss Detection: IEEE Access, v. 8, p. 16033–16048, doi:10.1109/ACCESS.2019.2962510.

Gong, Z., Z. Wang, M. J. F. Stive, C. Zhang, and A. Chu, 2012, Process-Based Morphodynamic Modeling of a Schematized Mudflat Dominated by a Long-Shore Tidal Current at the Central Jiangsu Coast, China: Journal of Coastal Research, v. 28, no. 6, p. 1381–1392, doi:10.2112/JCOASTRES-D-12-00001.1.

Hall, B., 2016, Facies classification using machine learning: The Leading Edge, v. 35, no. 10, p. 906–909, doi:10.1190/tle35100906.1.

Huang, L., X. Dong, and T. E. Clee, 2017, A scalable deep learning platform for identifying geologic features from seismic attributes: The Leading Edge, v. 36, no. 3, p. 249–256, doi:10.1190/tle36030249.1.

Ke, G., Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, 2017, LightGBM: A Highly Efficient Gradient Boosting Decision Tree: Proceedings of the 31st International Conference on Neural Information Processing Systems, v. 30, p. 3146–3154.

Li, Y., and R. Anderson-Sprecher, 2006, Facies identification from well logs: A comparison of discriminant analysis and naïve Bayes classifier: Journal of Petroleum Science and Engineering, v. 53, no. 3, p. 149–157, doi:10.1016/j.petrol.2006.06.001.

Lundberg, S., and S.-I. Lee, 2017, A Unified Approach to Interpreting Model Predictions: Proceedings of the 31st International Conference on Neural Information Processing Systems, p. 4768–4777.

Mahmoud, A. A., S. Elkatatny, and A. Al-AbdulJabbar, 2021, Application of machine learning models for real-time prediction of the formation lithology and tops from the drilling parameters: Journal of Petroleum Science and Engineering, v. 203, p. 108574, doi:10.1016/j.petrol.2021.108574.

Nielsen, M. A., 2015, Neural networks and deep learning: Determination press San Francisco, CA.

NPD, 2015, CO2 atlas for the Norwegian Continental Shelf: </en/facts/publications/co2-atlases/co2-atlas-for-the-norwegian-continental-shelf/> (accessed June 10, 2021).

NPD, 2021, FORCE 2020 Lithology Machine Learning Competition Results: <https://www.npd.no/en/force/Previous-events/results-of-the-FORCE-2020-lithology-competition/> (accessed June 27, 2021).

Pedregosa, F. et al., 2011, Scikit-learn: Machine Learning in Python: Journal of Machine Learning Research, v. 12, no. 85, p. 2825–2830.

Prokhorenkova, L., G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin, 2019, CatBoost: unbiased boosting with categorical features: arXiv preprint arXiv:1706.09516.

Rokach, L., and O. Z. Maimod, 2014, Data Mining With Decision Trees: Theory and Applications: World Scientific Publishing Co., 328 p.

Sebtosheikh, M. A., R. Motafakkerfard, M. A. Riahi, S. Moradi, and N. Sabety, 2015, Support vector machine method, a new technique for lithology prediction in an Iranian heterogeneous carbonate reservoir using petrophysical well logs: Carbonates and Evaporites, v. 30, no. 1, p. 59–68, doi:10.1007/s13146-014-0199-0.

Sharma, S., 2019, How to Classify Non-linear Data to Linear Data? <https://medium.com/analytics-vidhya/how-to-classify-non-linear-data-to-linear-data-bb2df1a6b781> (accessed June 27, 2021).

Wang, G., and T. R. Carr, 2012a, Marcellus Shale Lithofacies Prediction by Multiclass Neural Network Classification in the Appalachian Basin: Mathematical geosciences, v. 44, no. 8, p. 975–1004, doi:10.1007/s11004-012-9421-6.

Wang, G., and T. R. Carr, 2012b, Methodology of Organic-Rich Shale Lithofacies Identification and Prediction: A Case Study from Marcellus Shale in the Appalachian Basin: Computers & Geosciences, v. 49, p. 151–163, doi:10.1016/j.cageo.2012.07.011.

Zhang, Y., H. A. Salisch, and J. G. McPherson, 1999, Application of neural networks to identify lithofacies from well logs: Exploration Geophysics, v. 30, no. 2, p. 45–49, doi:10.1071/eg999045.

Zuo, R., 2017, Machine Learning of Mineralization-Related Geochemical Anomalies: A Review of Potential Methods: Natural resources research (New York, N.Y.), v. 26, no. 4, p. 457–464, doi:10.1007/s11053-017-9345-4.

8. APPENDIXES

Every python appendix included or mentioned in the current section could also be found open sourced in digital format on the following GitHub repository:

https://github.com/JohnMasapantaPozo/Machine-and-Deep-Learning-Applied-to-Geosciences

8.1 Appendix A – Additional utility functions Python Code

8.1.1 Plotting Functionalities (plotting.py)

```python

"""Log plotting functions

This script holds different functions such as raw_plot, augmented_logs, and litho_predictions,
which plot wireline raw logs, augmented logs, and lithology predictions, respecively. |
They can be imported as a modules when needed.

They require some functionalities from libraries such as  pandas, numpy, matplotlib,
and mpl_toolkits.
"""

#PLOTTING RAW LOGS
def raw_logs(logs, well_num):

  """Plots the raw logs contained in the original datasets after they have been formated.

  Parameters
  ----------
  logs: dataframe
    The raw logs once the headers and necessary columns have been formated and fixed.
  well_num: int
    The number of the well to be plotted. raw_logs internally defines a list of weells
    contained by the dataset, each of them could be called by its list index.

  Returns
  ----------
  plot:
    Different tracks having one well log each and a final track containing the
    lithofacies interpretation.
  """


    import numpy as np
    import matplotlib.pyplot as plt
    import matplotlib.colors as colors
    from mpl_toolkits.axes_grid1 import make_axes_locatable

    facies_colors = ['#F4D03F','#7ccc19','#196F3D','#160599','#2756c4','#3891f0','#80d4ff','#87039e','#ec90fc','
    facies_labels = ['SS', 'S-S', 'SH', 'MR', 'DOL','LIM', 'CH','HAL', 'AN', 'TF', 'CO', 'BS']

    facies_color_map = {} # creating facies color map
    for ind, label in enumerate(facies_labels):
        facies_color_map[label] = facies_colors[ind]

    wells = logs['WELL'].unique() # creating a wells list
    logs = logs[logs['WELL'] == wells[well_num]] # selecting well by index number
    logs = logs.sort_values(by='DEPTH_MD') # sorting well log by depth
    cmap_facies = colors.ListedColormap(facies_colors[0:len(facies_colors)], 'indexed')

    top = logs.DEPTH_MD.min()
    bot = logs.DEPTH_MD.max()

    real_label = np.repeat(np.expand_dims(logs['LITHO'].values, 1), 100, 1)

    f, ax = plt.subplots(nrows=1, ncols=17, figsize=(20, 12))
    log_colors = ['black', 'red', 'blue', 'green', 'purple','black', 'red', 'blue', 'green', 'purple', 'black',
                  'red', 'blue', 'green', 'purple', 'black', 'black', 'red', 'blue', 'green', 'purple', 'black',
                  'red', 'blue', 'green', 'purple', 'black']

    for i in range(7,23):
      ax[i-7].plot(logs.iloc[:,i], logs.DEPTH_MD, color=log_colors[i]) # plotting each well log on each track

      ax[i-7].set_ylim(top, bot)
      ax[i-7].set_xlabel(str(logs.columns[i]))
      ax[i-7].invert_yaxis()
      ax[i-7].grid()

    im = ax[-1].imshow(real_label, interpolation='none', aspect='auto', cmap=cmap_facies, vmin=0, vmax=12)
    ax[-1].set_xlabel('LITHO') # creating a facies log on the final track

    divider = make_axes_locatable(ax[-1]) # appending legend besides the facies log
    cax = divider.append_axes("right", size="20%", pad=0.05)
    cbar=plt.colorbar(im, cax=cax)
    cbar.set_label((12*' ').join(['SS', 'S-S', 'SH', 'MR', 'DOL','LIM', 'CH','HAL', 'AN', 'TF', 'CO', 'BS']))

    cbar.set_ticks(range(0,1)); cbar.set_ticklabels('')

    f.suptitle('WELL LOGS '+str(wells[well_num]), fontsize=16,y=0.9)


#PLOTTING LOGS AUGMENTED BY ML
def augmented_logs(logs, well_num):

    """Plots the raw, predicted, and augmented wireline logs after applying data augmentation.

    Parameters
    ----------
    logs: dataframe
      The raw, predicted, and augmented logs.
    well_num: int
      The number of the well to be plotted. augmented_logs internally defines a list of
      weells contained by the logs dataframe. each of which could be called by its list index.

    Returns
    ----------
    plot:
      Different tracks containing the raw, predicted, and augmented logs.
      Augmented logs mean that the missing values hbeen filled up by machine-learning
      predicted readings.
    """

    #auxiliar libraries
    import numpy as np
    import matplotlib.pyplot as plt
    import matplotlib.colors as colors
    from mpl_toolkits.axes_grid1 import make_axes_locatable

```

```
     facies_colors = ['#F4D03F','#7ccc19','#196F3D','#160599','#2756c4','#3891f0','#80d4ff','#87039e','#ec90fc','#F
     facies_labels = ['SS', 'S-S', 'SH', 'MR', 'DOL','LIM', 'CH','HAL', 'AN', 'TF', 'CO', 'BS']

     facies_color_map = {}  # creating facies color map
     for ind, label in enumerate(facies_labels):
         facies_color_map[label] = facies_colors[ind]

     wells = logs['WELL'].unique()
     logs = logs[logs['WELL'] == wells[well_num]] # selecting well by index number
     logs = logs.sort_values(by='DEPTH_MD') # sorting well log by depth
     cmap_facies = colors.ListedColormap(facies_colors[0:len(facies_colors)], 'indexed')

     top = logs.DEPTH_MD.min()
     bot = logs.DEPTH_MD.max()

    real_label = np.repeat(np.expand_dims(logs['LITHO'].values, 1), 100, 1)

    f, ax = plt.subplots(nrows=1, ncols=13, figsize=(20, 12))
    log_colors = ['black', 'red', 'blue', 'green', 'purple','black', 'red', 'blue', 'green', 'purple', 'black',
                  'red', 'blue', 'green', 'purple', 'black', 'black', 'red', 'blue', 'green', 'purple', 'black',
                  'red', 'blue', 'green', 'purple', 'black']

    for i in range(3,15):
      ax[i-3].plot(logs.iloc[:,i], logs.DEPTH_MD, color=log_colors[i]) # plotting raw, predicted, and augmented l
      ax[i-3].set_ylim(top, bot)

      ax[i-3].set_xlabel(str(logs.columns[i]))
      ax[i-3].invert_yaxis()
      ax[i-3].grid()

    im = ax[-1].imshow(real_label, interpolation='none', aspect='auto', cmap=cmap_facies, vmin=0, vmax=12)
    ax[-1].set_xlabel('LITHO') # creating a facies log on the final track

    divider = make_axes_locatable(ax[-1]) # appending legend besides the facies log
    cax = divider.append_axes("right", size="20%", pad=0.05)
    cbar=plt.colorbar(im, cax=cax)
    cbar.set_label((12*' ').join(['SS', 'S-S', 'SH', 'MR', 'DOL','LIM', 'CH','HAL', 'AN', 'TF', 'CO', 'BS']))

    cbar.set_ticks(range(0,1)); cbar.set_ticklabels('')

    f.suptitle('WELL LOGS '+str(wells[well_num]), fontsize=16,y=0.9)


#PLOTTING LITHOFACIES PREDICTION
def litho_prediction(logs, well_num, n_pred):

    """Plots the raw logs, the lihtology interpretation, and the n_pred number of predcted
    lithologies by machine learning.

    Parameters
    ----------
    logs: dataframe
      Dataframe holding the raw wireline logs, true lithology, and n_pred columns
      containing different ML model predictions each.
    well_num: int
      The number of the well to be plotted. litho_prediction internally defines a list of
      weells contained by the logs dataframe, each of which could be called by its list index.

    Returns
    ----------
    plot:
      Different track plots representing each wireline log, the true lihtology and the
      predicted lithologies by dfferent mane-learning models.
    """

    import numpy as np
    import matplotlib.pyplot as plt
    import matplotlib.colors as colors
    from mpl_toolkits.axes_grid1 import make_axes_locatable

    facies_colors = ['#F4D03F','#7ccc19','#196F3D','#160599','#2756c4','#3891f0','#80d4ff','#87039e','#ec90fc','#
    facies_labels = ['SS', 'S-S', 'SH', 'MR', 'DOL','LIM', 'CH','HAL', 'AN', 'TF', 'CO', 'BS']

    facies_color_map = {} #creating facies coorap
    for ind, label in enumerate(facies_labels):
        facies_color_map[label] = facies_colors[ind]

    wells = logs['WELL'].unique() # well names list
    logs = logs[logs['WELL'] == wells[well_num]]
    logs = logs.sort_values(by='DEPTH_MD') # sorting the plotted well logs by depth
    cmap_facies = colors.ListedColormap(facies_colors[0:len(facies_colors)], 'indexed')

    top = logs.DEPTH_MD.min()
    bot = logs.DEPTH_MD.max()

    f, ax = plt.subplots(nrows=1, ncols=(12+n_pred), figsize=(20, 12))
    log_colors = ['black', 'red', 'blue', 'green', 'purple','black', 'red', 'blue', 'green', 'purple', 'black',
                  'red', 'blue', 'green', 'purple', 'black', 'black', 'red', 'blue', 'green', 'purple', 'black',
                  'red', 'blue', 'green', 'purple', 'black']

    for i in range(7,18):
      ax[i-7].plot(logs.iloc[:,i], logs.DEPTH_MD, color=log_colors[i]) # plotting continuous wireline logs
      ax[i-7].set_ylim(top, bot)
      ax[i-7].set_xlabel(str(logs.columns[i]))
      ax[i-7].invert_yaxis()
      ax[i-7].grid()

    for j in range((-1-n_pred), 0): # ploting the lithology predictions obtainedby ML
      label = np.repeat(np.expand_dims(logs.iloc[:,j].values, 1), 100, 0)
      im = ax[j].imshow(label, interpolation='none', aspect='auto', cmap=cmap_facies, vmin=0, vmax=12)
      ax[j].set_xlabel(str(logs.columns[j]))

    divider = make_axes_locatable(ax[-1]) # appending lithology legend
    cax = divider.append_axes("right", size="20%", pad=0.05)
    cbar=plt.colorbar(im, cax=cax)
    cbar.set_label((12*' ').join(['SS', 'S-S', 'SH', 'MR', 'DOL','LIM', 'CH','HAL', 'AN', 'TF', 'CO', 'BS']))
    cbar.set_ticks(range(0,1)); cbar.set_ticklabels('')

    f.suptitle('WELL LOGS '+str(wells[well_num]), fontsize=14,y=0.94)
```

8.1.2 Confusion Matrix and Penalty Matrix Score (additional_functions.py)

```

"""Additional functions

This script holds the matrix_score and confusion_matrix functions, which serves as
evaluation for the classification performance each machine-learning has.

They require some functionalities from libraries such as  pandas, numpy, matplotlib,
and scikit-lean.
"""

def matrix_score(y_true, y_pred):

    """Returns the penalty matrix score obined by the predicted lithofacies a
    particular machine-learning model is able to provide. The matrix score was a metric
    measure proposed by the FORCE commitee in order to provide the prediction performance
    measure from a petrophyicist perpective.

    Parameters
    ----------
    y_true: list
      The actual lithologies given by the datasets provider.
    y_pred: list
      The predicted lithofacies obtained by a particular machine learning model.

    Returns
    ----------
    matrix penaty score:
      Penalty matrix score obined by a particular machine-learning model.
    """

    import numpy as np
    matrix_path = '/content/drive/MyDrive/Thesis_data/penalty_matrix.npy'
    A = np.load(matrix_path)
    S = 0.0
    y_true = y_true.astype(int)
    y_pred = y_pred.astype(int)
    for i in range(0, y_true.shape[0]):
        S -= A[y_true[i], y_pred[i]]
    return S/y_true.shape[0]

# Confusion Matrix Function
def confusion_matrix(y_true, y_pred):

    """Plots a confusion matrix normalized by the number of predictions a particular
    machine learning algorithm has. By ormalize we look at the number of predictions
    the model gets right.

    Parameters
    ----------
    y_true: list
      The actual lithologies given by the datasets provider.
    y_pred: list
      The predicted lithofacies obtained by a particular machine learning model.

    Returns
    ----------
    confusion matrix:
      A normalized confusion matrix by the number of predictions.
    """

    """
    from sklearn.metrics import confusion_matrix
    from sklearn.metrics._plot.confusion_matrix import ConfusionMatrixDisplay
    from itertools import product

    def litho_confusion_matrix(y_true, y_pred):
      facies_dict = {0:'Sandstone', 1:'Sandstone/Shale', 2:'Shale', 3:'Marl',
                     4:'Dolomite', 5:'Limestone', 6:'Chalk', 7:'Halite',
                     8:'Anhydrite', 9:'Tuff', 10:'Coal', 11:'Basement'}

      # creating a lithofacies names
      labels = list(set(list(y_pred.unique()) + list(y_true.unique())))
      label_names = [facies_dict[k] for k in labels]

      # normalizing confusion matrix by the number of predictions
      cm = pd.DataFrame(confusion_matrix(y_true.values, y_pred.values))
      summ = cm.sum(axis=0)
      cm_norm = pd.DataFrame(np.zeros(cm.shape))
      for i in range(cm.shape[1]):
        for j in range(cm.shape[0]):
          cm_norm[i][j] = cm[i][j]*100/summ[i]
      cm_final = cm_norm.fillna(0).to_numpy()

      fig, ax = plt.subplots(figsize=(12,8))
      plt.imshow(cm_final, interpolation='nearest', cmap=plt.cm.Blues)
      plt.title('NORMALIZED CONFUSION MATRIX', size=15)
      tick_marks = np.arange(len(label_names))
      plt.xticks(tick_marks, label_names, rotation=90)
      plt.yticks(tick_marks, label_names)
      plt.colorbar()

      # creating a scores format (black and white)
      fmt = '.2f'
      thresh = cm_final.max() / 2.
      for i, j in product(range(cm_final.shape[0]),   range(cm_final.shape[1])):
        plt.text(j, i, format(cm_final[i, j], fmt),
                   horizontalalignment="center",
                   color="white" if cm_final[i, j] > thresh else "black")

      plt.ylabel('True label', size=14)
      plt.xlabel('Predicted label', size=14)
```

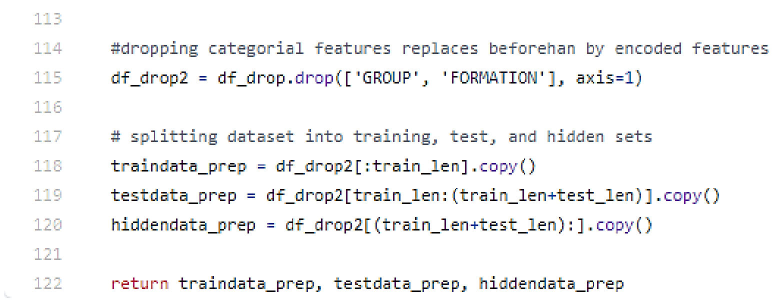
8.1.3 Data formatting (data_formating.py)

```python

"""Data Formating

This script simply rename the column names in a consitent manner for the training,
open test, and hidden test sets. It also maps the lithofacies labesl with numbers
from 0 to 11 and drops the intepretation confidence column.
"""

def formating(training_raw, test_raw, hidden_raw):

  """Returns the training, open test, and hidden test dataframes with consistent formats.

  Parameters
  ----------
  training_raw: Dataframe
    Raw training dataframe.
  test_raw: Dataframe
    Raw open test dataframe.
  hidden_raw: Dataframe
    Raw hidden test dataframe.

  Returns
  ----------
  training_formated: Dataframe
    Formated training dataframe.
  test_formated: Dataframe
    Formated open test dataframe.
  hidden_formated: Dataframe
    Formated hidden test dataframe.
  """

  lithology_numbers = {30000: 0, 65030: 1, 65000: 2, 80000: 3, 74000: 4, 70000: 5,
                       70032: 6, 88000: 7, 86000: 8, 99000: 9, 90000: 10,
                       93000: 11
                       }

  # formating raw training set
  training_formated = training_raw.rename(columns={'FORCE_2020_LITHOFACIES_LITHOLOGY':'LITHO'})
  training_formated = training_formated.drop(['FORCE_2020_LITHOFACIES_CONFIDENCE'], axis=1)
  training_formated['LITHO'] = training_formated["LITHO"].map(lithology_numbers)

  #formating raw test set
  test_formated = test_raw.rename(columns={'FORCE_2020_LITHOFACIES_LITHOLOGY':'LITHO'})
  test_formated['LITHO'] = test_formated["LITHO"].map(lithology_numbers)

  # formating raw hidden set
  hidden_formated = hidden_raw.rename(columns={'FORCE_2020_LITHOFACIES_LITHOLOGY':'LITHO'})
  hidden_formated = hidden_formated.drop(['FORCE_2020_LITHOFACIES_CONFIDENCE'], axis=1)
  hidden_formated['LITHO'] = hidden_formated['LITHO'].map(lithology_numbers)

  return(training_formated, test_formated, hidden_formated)
```

8.1.4 Data Pre-processing (preprocessing.py)

```

"""Data preprocessing

This script preprocess the training, open test, and hidden test sets.
The preprocess involves droping unnecessary columns bases un presence percentage,
clustering well logs by location as features, and splitting again the data into
3 subsets. Clustering performed by unsipervides K-Means algorithm.
"""

def base_well_name(row):
    well_name = row['WELL']
    return well_name.split()[0]

def preprocess_data(training_formated, test_formated, hidden_formated):

  """Returns the training, open test, and hidden test dataframes with without
  'SGR', 'ROPA', 'RXO', 'MUDWEIGHT', and including 'Cluster' as a feature.

  Parameters
  ----------
  training_formated: Dataframe
    Formated training dataframe.
  test_formated: Dataframe
    Formated open test dataframe.
  hidden_formated: Dataframe
    Formated hidden test dataframe.

  Returns
  ----------
  traindata_prep: Dataframe
    Pre-processed training dataframe.
  testdata_prep: Dataframe
    Pre-processed open test dataframe.
  hiddendata_prep: Dataframe
    Pre-processed hidden test dataframe.
  """

  import pandas as pd
  import numpy as np
  from sklearn.cluster import KMeans

  train_len = training_formated.shape[0] # storing datasets lenghts
  test_len = test_formated.shape[0]
  hidden_len = hidden_formated.shape[0]

  # concatenating datasets and dropping indexes
  df_concat = pd.concat((training_formated,
                         test_formated,
                         hidden_formated)).reset_index(drop=True)

  drop_cols = ['SGR', 'ROPA', 'RXO', 'MUDWEIGHT']
  df_drop = df_concat.drop(drop_cols, axis=1) # dropping unnecesary columns

  # encoding GROUP, FORMATION, and WELL
  df_drop['GROUP_encoded'] = df_drop['GROUP'].astype('category')
```

```
df_drop['GROUP_encoded'] = df_drop['GROUP_encoded'].cat.codes

df_drop['FORMATION_encoded'] = df_drop['FORMATION'].astype('category')
df_drop['FORMATION_encoded'] = df_drop['FORMATION_encoded'].cat.codes

df_drop['WELL_encoded'] = df_drop['WELL'].astype('category')
df_drop['WELL_encoded'] = df_drop['WELL_encoded'].cat.codes

# creating a well names dataframe
training_wells = training_formated['WELL'].unique()
test_wells = test_formated['WELL'].unique()
hidden_wells= hidden_formated['WELL'].unique()

well_names = np.concatenate((training_wells, test_wells, hidden_wells))
well_names_df = pd.DataFrame({'WELL':well_names})

# importing wells metadata
well_meta_df = pd.read_csv('/content/drive/MyDrive/Thesis_data/wellbore_exploration_all.csv')
well_meta_df.rename(columns={'wlbWellboreName': 'WELL',
                             'wlbWell': 'WELL_HEAD',
                             'wlbNsDecDeg': 'lat',
                             'wlbEwDesDeg': 'lon',
                             'wlbDrillingOperator': 'Drilling_Operator',
                             'wlbPurposePlanned': 'Purpose',
                             'wlbCompletionYear': 'Completion_Year',
                             'wlbFormationAtTd': 'Formation'
                             }, inplace=True)

well_locations_df = well_meta_df[['WELL_HEAD', 'lat', 'lon']].drop_duplicates(subset=['WELL_HEAD'])
well_meta_df = well_meta_df[['WELL', 'Drilling_Operator', 'Purpose', 'Completion_Year', 'Formation']]

well_names_df['WELL_HEAD'] = well_names_df.apply(lambda row: base_well_name(row), axis=1)

locations_df = well_names_df.merge(well_locations_df, how='inner', on='WELL_HEAD')
locations_df = locations_df.merge(well_meta_df, how='left', on='WELL')

# labeling train and test wells in a new column
locations_df.loc[locations_df['WELL'].isin(training_wells), 'Dataset'] = 'Train'
locations_df.loc[locations_df['WELL'].isin(test_wells), 'Dataset'] = 'Test'
locations_df.loc[locations_df['WELL'].isin(hidden_wells), 'Dataset'] = 'Hidden'

LonLat_df =  locations_df.drop(['WELL',
                                'WELL_HEAD',
                                'Drilling_Operator',
                                'Purpose',
                                'Completion_Year',
                                'Formation',
                                'Dataset'],
                                axis=1)
# applying K-Means
location = LonLat_df[['lon', 'lat']].values
kmeans = KMeans(n_clusters=3, init='k-means++', random_state=1)
labels = kmeans.fit_predict(location)

# including Cluster as a feature
df_drop = df_drop.rename(columns={'WELL':'Cluster'})
clust_map = dict(zip(locations_df.WELL.values, labels))
df_drop['Cluster'] = df_drop['Cluster'].map(clust_map)
```



```
    
    #dropping categorial features replaces beforehan by encoded features
    df_drop2 = df_drop.drop(['GROUP', 'FORMATION'], axis=1)
    
    # splitting dataset into training, test, and hidden sets
    traindata_prep = df_drop2[:train_len].copy()
    testdata_prep = df_drop2[train_len:(train_len+test_len)].copy()
    hiddendata_prep = df_drop2[(train_len+test_len):].copy()
    
    return traindata_prep, testdata_prep, hiddendata_prep
```

8.1.5 Data machine-learning augmentation (augmentation.py)

```python

"""Data augmentaion

This script test three diffrent machine learning regressors in order
to predict the four most relevant wireline logs,DTC, NPHI, DTS, and RHOB.
Afterwards, the models' results are compared between each other in each
prediction stage in order to select the best performing model. Later,
at each stage the missing instances encountered in the training, open test,
and hidden test datasets are imputed by the predictions otained by the best
ML model at each stage. Finally, 6 additional features are included in each
data subset.

It requires xgboost, lightgbm, catboost to be installed before running, as
well as pandas and numpy functionalities.
"""

def data_augmentation(traindata, testdata, hiddendata):

  """Receives the pre-processed data and returns the data with additional
  columns named as cleaned data. These additional columns include the
  predicted, augmented, and 6 additional features icluding impedances
  (S_I, P_I), bulk and shear modulus (K, G), slowness ratio (DT_R), and
  true and measure depths ratio (MD_TVD).

  Parameters
  ----------
  training: Dataframe
    Pre-processed training dataframe.
  testdata: Dataframe
    Pre-processed open test dataframe.
  hiddendata: Dataframe
    Pre-processed hidden test dataframe.

  Returns
  ----------
  cleaned_traindata: Dataframe
    Cleaned trainig dataframe.
  cleaned_testdata: Dataframe
    Cleaned test dataframe.
  cleaned_hiddendata: Dataframe
    Cleaned hidden dataframe.
  """
  import numpy as np
  import pandas as pd
  from xgboost import XGBRegressor
  from lightgbm import LGBMRegressor
  from catboost import CatBoostRegressor
  from sklearn.model_selection import train_test_split
  from sklearn.metrics import max_error, mean_absolute_error, mean_squared_error
  from sklearn.metrics import explained_variance_score, r2_score

  # list of regressors to be tests set to be supported by GPUs
  estimators = [XGBRegressor(n_estimators=250,
                             tree_method='gpu_hist',
                             learning_rate=0.05),
               CatBoostRegressor(task_type='GPU'),

               LGBMRegressor(device='gpu',
                             gpu_platform_id=1,
                             gpu_device_id=0),
               ]

estimator_names = ['EXTREME BOOST REGRESSOR',
                   'CATBOOST REGRESSOR',
                   'LIGHTBOOST REGRESSOR'] # regressors names

# predicting DTS
print('-----------------------------------------PREDINCTING DTS-----------------------------------------')

"""Getting data containing DTS readings and
spliting into a new training and validation subsets.
"""
traindata_dts = traindata[traindata.DTS.notna()]
X_dts = traindata_dts.drop(['LITHO', 'DTS'], axis=1)
Y_dts = traindata_dts['DTS']

"""Inputing remanent features by their median.
"""
X_dts_inp = X_dts.apply(lambda x: x.fillna(x.median()), axis=0) #Imputation
X_dts_train, X_dts_val, Y_dts_train, Y_dts_val = train_test_split(X_dts_inp,
                                                                  Y_dts,
                                                                  test_size=0.3,
                                                                  random_state=42) #train-validation split
"""Getting DTS testing set from open test set
"""
testdata_dts = testdata[testdata.DTS.notna()]
X_dts = testdata_dts.drop(['LITHO', 'DTS'], axis=1)
X_dts_test = X_dts.apply(lambda x: x.fillna(x.median()), axis=0)
Y_dts_test = testdata_dts['DTS']

"""Testing regressors
"""
i = 0
rme_errors = []
for estimator in estimators:

  print('----------------------{} MODEL----------------------'.format(estimator_names[i]))

  estimator.fit(X_dts_train,
                Y_dts_train.values.ravel(),
                early_stopping_rounds=100,
                eval_set=[(X_dts_test, Y_dts_test)],
                verbose=100
                ) # fitting while validationg on open test set

  """Predicting DTS on training, validation, and test sets and computing metrics
  """
  train_pred = estimator.predict(X_dts_train)
  val_pred = estimator.predict(X_dts_val)
  test_pred = estimator.predict(X_dts_test)

  varinace_train = explained_variance_score(Y_dts_train, train_pred)
  max_eror_train = max_error(Y_dts_train, train_pred)
```

```
        ms_error_train = np.sqrt(mean_squared_error(Y_dts_train, train_pred))
        mabs_error_train = mean_absolute_error(Y_dts_train, train_pred)
        r2_train = r2_score(Y_dts_train, train_pred)

        varinace_val = explained_variance_score(Y_dts_val, val_pred)
        max_eror_val = max_error(Y_dts_val, val_pred)
        ms_error_val = np.sqrt(mean_squared_error(Y_dts_val, val_pred))
        mabs_error_val = mean_absolute_error(Y_dts_val, val_pred)
        r2_val = r2_score(Y_dts_val, val_pred)

        varinace_test = explained_variance_score(Y_dts_test, test_pred)
        max_eror_test = max_error(Y_dts_test, test_pred)
        ms_error_test = np.sqrt(mean_squared_error(Y_dts_test, test_pred))
        mabs_error_test = mean_absolute_error(Y_dts_test, test_pred)
        r2_test = r2_score(Y_dts_test, test_pred)

        rme_errors.append(ms_error_test) #storing root mean squared error for model comparition

        print('''--------TRAINING SET METRICS--------
        explianed varianve {},
        maximum error {},
        root mean squared error {},
        maximum absolute error {},
        R2 {}'''.format(varinace_train, max_eror_train, ms_error_train, mabs_error_train, r2_train))

        print('''-------VALIDATION SET METRICS-------
        explianed varianve {},
        maximum error {},
        root mean squared error{},
        maximum absolute error {},
        R2 {}'''.format(varinace_val, max_eror_val, ms_error_val, mabs_error_val, r2_val))

        print('''-----------TEST SET METRICS----------
        explianed varianve {},
        maximum error {},
        root mean squared error {},
        maximum absolute error {},
        R2 {}'''.format(varinace_test, max_eror_test, ms_error_test, mabs_error_test, r2_test))

        i += 1

    # selecting bet model to perform imputation
    selected_model_index = rme_errors.index(min(rme_errors))
    print('TRAINING BEST MODEL TO PERFORM DTS IMPUTATION {}'.format(estimator_names[selected_model_index]))
    model1000 = estimators[selected_model_index]
    model1000.fit(X_dts_train,
                  Y_dts_train.values.ravel(),
                  early_stopping_rounds=100,
                  eval_set=[(X_dts_test, Y_dts_test)],
                  verbose=100) # fitting best ML regresor

    # filling nan values before predicting DTS
    X_train_DTS = traindata.drop(['LITHO', 'DTS'], axis=1)
    X_train_DTS2 = X_train_DTS.apply(lambda x: x.fillna(x.median()), axis=0)

    X_test_DTS = testdata.drop(['LITHO', 'DTS'], axis=1)
    X_test_DTS2 = X_test_DTS.apply(lambda x: x.fillna(x.median()), axis=0)
```

```
X_hidden_DTS = hiddendata.drop(['LITHO', 'DTS'], axis=1)
X_hidden_DTS2 = X_hidden_DTS.apply(lambda x: x.fillna(x.median()), axis=0)

# predicting DTS on complete datasets
traindata['DTS_pred'] = model1000.predict(X_train_DTS2)
testdata['DTS_pred'] = model1000.predict(X_test_DTS2)
hiddendata['DTS_pred'] = model1000.predict(X_hidden_DTS2)

# imputing nan values in DTS by ML predictions
traindata['DTS_COMB'] = traindata['DTS']
traindata['DTS_COMB'].fillna(traindata['DTS_pred'], inplace=True)

testdata['DTS_COMB'] = testdata['DTS']
testdata['DTS_COMB'].fillna(testdata['DTS_pred'], inplace=True)

hiddendata['DTS_COMB'] = hiddendata['DTS']
hiddendata['DTS_COMB'].fillna(hiddendata['DTS_pred'], inplace=True)

# predicting NPHI
print('----------------------------------------PREDINCTING NPHI-----------------------------------------')
"""Getting data containing DTS readings and
spliting into a new training and validation subsets.
"""

traindata_nphi = traindata[traindata.NPHI.notna()]
X_nphi = traindata_nphi.drop(['LITHO', 'DTS', 'DTS_pred', 'NPHI'], axis=1)
Y_nphi = traindata_nphi['NPHI']

"""Inputing remanent features by their median.
"""


X_nphi_inp = X_nphi.apply(lambda x: x.fillna(x.median()), axis=0) # imputation
X_nphi_train, X_nphi_val, Y_nphi_train, Y_nphi_val = train_test_split(X_nphi_inp,
                                                                      Y_nphi,
                                                                      test_size=0.3,
                                                                      random_state=42) # train-validation split
"""Getting NPHI testing set from open test set
"""
testdata_nphi = testdata[testdata.NPHI.notna()]
X_nphi = testdata_nphi.drop(['LITHO', 'DTS', 'DTS_pred', 'NPHI'], axis=1)
X_nphi_test = X_nphi.apply(lambda x: x.fillna(x.median()), axis=0)
Y_nphi_test = testdata_nphi['NPHI']

print('Training set sahpe {} and validation set shape {} and test set shape'.format(X_nphi_train.shape,
                                                                                     X_nphi_val.shape,
                                                                                     X_nphi_test.shape))
"""Teting regressors
"""
i = 0
rme_errors = []
for estimator in estimators:

  print('----------------------{} MODEL----------------------'.format(estimator_names[i]))

  estimator.fit(X_nphi_train,
                Y_nphi_train.values.ravel(),
                early_stopping_rounds=100,
                    early_stopping_rounds=100,
                    eval_set=[(X_nphi_test, Y_nphi_test)],
                    verbose=100)
      """Predicting DTS on training, validation, and test sets and computing metrics
      """
      train_pred = estimator.predict(X_nphi_train)
      val_pred = estimator.predict(X_nphi_val)
      test_pred = estimator.predict(X_nphi_test)

      varinace_train = explained_variance_score(Y_nphi_train, train_pred)
      max_eror_train = max_error(Y_nphi_train, train_pred)
      ms_error_train = np.sqrt(mean_squared_error(Y_nphi_train, train_pred))
      mabs_error_train = mean_absolute_error(Y_nphi_train, train_pred)
      r2_train = r2_score(Y_nphi_train, train_pred)

      varinace_val = explained_variance_score(Y_nphi_val, val_pred)
      max_eror_val = max_error(Y_nphi_val, val_pred)
      ms_error_val = np.sqrt(mean_squared_error(Y_nphi_val, val_pred))
      mabs_error_val = mean_absolute_error(Y_nphi_val, val_pred)
      r2_val = r2_score(Y_nphi_val, val_pred)

      varinace_test = explained_variance_score(Y_nphi_test, test_pred)
      max_eror_test = max_error(Y_nphi_test, test_pred)
      ms_error_test = np.sqrt(mean_squared_error(Y_nphi_test, test_pred))
      mabs_error_test = mean_absolute_error(Y_nphi_test, test_pred)
      r2_test = r2_score(Y_nphi_test, test_pred)

      rme_errors.append(ms_error_test) # storing root mean squared error for model comparition

      print('''--------TRAINING SET METRICS--------
      explianed varianve {},
      maximum error {},
      root mean squared error {},

      maximum absolute error {},
      R2 {}'''.format(varinace_train, max_eror_train, ms_error_train, mabs_error_train, r2_train))

      print('''-------VALIDATION SET METRICS-------
      explianed varianve {},
      maximum error {},
      root mean squared error{},
      maximum absolute error {},
      R2 {}'''.format(varinace_val, max_eror_val, ms_error_val, mabs_error_val, r2_val))

      print('''-----------TEST SET METRICS----------
      explianed varianve {},
      maximum error {},
      root mean squared error {},
      maximum absolute error {},
      R2 {}'''.format(varinace_test, max_eror_test, ms_error_test, mabs_error_test, r2_test))

      i += 1

    # selecting bet model to perform imputation
    selected_model_index = rme_errors.index(min(rme_errors))
    print('TRAINING BEST MODEL TO PERFORM NPHI IMPUTATION {}'.format(estimator_names[selected_model_index]))
    model2000 = estimators[selected_model_index]
    model2000.fit(X_nphi_train,
              Y_nphi_train.values.ravel(),
              early_stopping_rounds=100,
              eval_set=[(X_nphi_test, Y_nphi_test)],
              verbose=100) # fitting bets ML regressor

# filling nan values before predicting NPHI
X_train_NPHI = traindata.drop(['LITHO', 'DTS', 'DTS_pred', 'NPHI'], axis=1)
X_train_NPHI2 = X_train_NPHI.apply(lambda x: x.fillna(x.median()), axis=0)

X_test_NPHI = testdata.drop(['LITHO', 'DTS', 'DTS_pred', 'NPHI'], axis=1)
X_test_NPHI2 = X_test_NPHI.apply(lambda x: x.fillna(x.median()), axis=0)

X_hidden_NPHI = hiddendata.drop(['LITHO', 'DTS', 'DTS_pred', 'NPHI'], axis=1)
X_hidden_NPHI2 = X_hidden_NPHI.apply(lambda x: x.fillna(x.median()), axis=0)

# predicting NPHI on complete datasets
traindata['NPHI_pred'] = model2000.predict(X_train_NPHI2)
testdata['NPHI_pred'] = model2000.predict(X_test_NPHI2)
hiddendata['NPHI_pred'] = model2000.predict(X_hidden_NPHI2)

# inputing nan values in NPHI by ML predictions
traindata['NPHI_COMB'] = traindata['NPHI']
traindata['NPHI_COMB'].fillna(traindata['NPHI_pred'], inplace=True)

testdata['NPHI_COMB'] = testdata['NPHI']
testdata['NPHI_COMB'].fillna(testdata['NPHI_pred'], inplace=True)

hiddendata['NPHI_COMB'] = hiddendata['NPHI']
hiddendata['NPHI_COMB'].fillna(hiddendata['NPHI_pred'], inplace=True)

# predicting RHOB

print('----------------------------------------PREDINCTING RHOB----------------------------------------')

"""Getting data containing RHOB readings and
spliting into a new training and validation subsets.
"""

traindata_rhob = traindata[traindata.RHOB.notna()]
X_rhob = traindata_rhob.drop(['LITHO', 'DTS', 'DTS_pred', 'NPHI', 'NPHI_pred', 'RHOB'], axis=1)
Y_rhob = traindata_rhob['RHOB']

"""Inputing remanent features by their median.
"""

X_rhob_inp = X_rhob.apply(lambda x: x.fillna(x.median()), axis=0) #imputation
X_rhob_train, X_rhob_val, Y_rhob_train, Y_rhob_val = train_test_split(X_rhob_inp,
                                                                      Y_rhob,
                                                                      test_size=0.3,
                                                                      random_state=42) # train-validation split
"""Getting RHOB testing set from open test set
"""
testdata_rhob = testdata[testdata.RHOB.notna()]
X_rhob = testdata_rhob.drop(['LITHO', 'DTS', 'DTS_pred', 'NPHI', 'NPHI_pred', 'RHOB'], axis=1)
X_rhob_test = X_rhob.apply(lambda x: x.fillna(x.median()), axis=0)
Y_rhob_test = testdata_rhob['RHOB']

print('Training set sahpe {}, validation set shape {}, and test shape {}'.format(X_rhob_train.shape,
                                                                                X_rhob_val.shape,
                                                                                X_rhob_test.shape))
"""Testing regressors
"""
i = 0
rme_errors = []
for estimator in estimators:

  print('----------------------{} MODEL----------------------'.format(estimator_names[i]))
  estimator.fit(X_rhob_train,
                Y_rhob_train.values.ravel(),
                early_stopping_rounds=100,
                eval_set=[(X_rhob_test, Y_rhob_test)],
                verbose=100)

  """Predicitng RHOB on training, validation, and test sets and computing metrics
  """
  train_pred = estimator.predict(X_rhob_train)
  val_pred = estimator.predict(X_rhob_val)
  test_pred = estimator.predict(X_rhob_test)

  varinace_train = explained_variance_score(Y_rhob_train, train_pred)
  max_eror_train = max_error(Y_rhob_train, train_pred)
  ms_error_train = np.sqrt(mean_squared_error(Y_rhob_train, train_pred))
  mabs_error_train = mean_absolute_error(Y_rhob_train, train_pred)
  r2_train = r2_score(Y_rhob_train, train_pred)

  varinace_val = explained_variance_score(Y_rhob_val, val_pred)
  max_eror_val = max_error(Y_rhob_val, val_pred)
  ms_error_val = np.sqrt(mean_squared_error(Y_rhob_val, val_pred))
  mabs_error_val = mean_absolute_error(Y_rhob_val, val_pred)
  r2_val = r2_score(Y_rhob_val, val_pred)


  varinace_test = explained_variance_score(Y_rhob_test, test_pred)
  max_eror_test = max_error(Y_rhob_test, test_pred)
  ms_error_test = np.sqrt(mean_squared_error(Y_rhob_test, test_pred))
  mabs_error_test = mean_absolute_error(Y_rhob_test, test_pred)
  r2_test = r2_score(Y_rhob_test, test_pred)

  rme_errors.append(ms_error_test) # storing root mean squared error for model comparition

  print('''--------TRAINING SET METRICS--------
  explianed varianve {},
  maximum error {},
  root mean squared error {},
  maximum absolute error {},
  R2 {}'''.format(varinace_train, max_eror_train, ms_error_train, mabs_error_train, r2_train))

  print('''-------VALIDATION SET METRICS-------
  explianed varianve {},
  maximum error {},
  root mean squared error{},
  maximum absolute error {},
  R2 {}'''.format(varinace_val, max_eror_val, ms_error_val, mabs_error_val, r2_val))

  print('''-----------TEST SET METRICS----------
  explianed varianve {},
      maximum error {},
      root mean squared error {},
      maximum absolute error {},
      R2 {}'''.format(varinace_test, max_eror_test, ms_error_test, mabs_error_test, r2_test))

      i += 1

    #selecting best model to perform imputation
    selected_model_index = rme_errors.index(min(rme_errors))
    print('TRAINING BEST MODEL TO PERFORM RHOB IMPUTATION {}'.format(estimator_names[selected_model_index]))
    model4000 = estimators[selected_model_index]

    model4000.fit(X_rhob_train,
                  Y_rhob_train.values.ravel(),
                  early_stopping_rounds=100,
                  eval_set=[(X_rhob_test, Y_rhob_test)],
                  verbose=100) #fitting best ML regressor

    # filling nan values before predicting RHOB
    X_train_RHOB = traindata.drop(['LITHO', 'DTS', 'DTS_pred', 'NPHI', 'NPHI_pred', 'RHOB'], axis=1)
    X_train_RHOB2 = X_train_RHOB.apply(lambda x: x.fillna(x.median()), axis=0)

    X_test_RHOB = testdata.drop(['LITHO', 'DTS', 'DTS_pred', 'NPHI', 'NPHI_pred', 'RHOB'], axis=1)
    X_test_RHOB2 = X_test_RHOB.apply(lambda x: x.fillna(x.median()), axis=0)

    X_hidden_RHOB = hiddendata.drop(['LITHO', 'DTS', 'DTS_pred', 'NPHI', 'NPHI_pred', 'RHOB'], axis=1)
    X_hidden_RHOB2 = X_hidden_RHOB.apply(lambda x: x.fillna(x.median()), axis=0)

    # predicting RHOB on complete datasets
    traindata['RHOB_pred'] = model4000.predict(X_train_RHOB2)
    testdata['RHOB_pred'] = model4000.predict(X_test_RHOB2)
    hiddendata['RHOB_pred'] = model4000.predict(X_hidden_RHOB2)


    # imputing nan values in RHOB by ML predictions
    traindata['RHOB_COMB'] = traindata['RHOB']
    traindata['RHOB_COMB'].fillna(traindata['RHOB_pred'], inplace=True)

    testdata['RHOB_COMB'] = testdata['RHOB']
    testdata['RHOB_COMB'].fillna(testdata['RHOB_pred'], inplace=True)

    hiddendata['RHOB_COMB'] = hiddendata['RHOB']
    hiddendata['RHOB_COMB'].fillna(hiddendata['RHOB_pred'], inplace=True)

    # predicting DTC
    print('---------------------------------------PREDINCTING DTC----------------------------------------')

    """Getting data containing DTC readings and
    spliting into a new training and validation subsets.
    """
    traindata_dtc = traindata[traindata.DTC.notna()]
    X_dtc = traindata_dtc.drop(['LITHO', 'DTS', 'DTS_pred', 'NPHI', 'NPHI_pred', 'RHOB', 'RHOB_pred', 'DTC'], axis=1)
    Y_dtc = traindata_dtc['DTC']

    """Inputing remanent features by their median.
    """

    X_dtc_inp = X_dtc.apply(lambda x: x.fillna(x.median()), axis=0) #imputation
    X_dtc_train, X_dtc_val, Y_dtc_train, Y_dtc_val = train_test_split(X_dtc_inp,
                                                                    Y_dtc,
                                                                    test_size=0.3,
                                                                    random_state=42) #train_validation split
"""Getting DTC testing set from open test set
"""
testdata_dtc = testdata[testdata.DTC.notna()]
X_dtc = testdata_dtc.drop(['LITHO', 'DTS', 'DTS_pred', 'NPHI', 'NPHI_pred', 'RHOB', 'RHOB_pred', 'DTC'], axis=1)
X_dtc_test = X_dtc.apply(lambda x: x.fillna(x.median()), axis=0)
Y_dtc_test = testdata_dtc['DTC']

print('Training set sahpe {}, validation set shape {}, and test shape {}'.format(X_dtc_train.shape,
                                                                               X_dtc_val.shape,
                                                                               X_dtc_test.shape))

"""Testing regressors
"""
i = 0
rme_errors = []
for estimator in estimators:

  print('----------------------{} MODEL----------------------'.format(estimator_names[i]))
  estimator.fit(X_dtc_train,
                Y_dtc_train.values.ravel(),
                early_stopping_rounds=100,
                eval_set=[(X_dtc_test, Y_dtc_test)],
                verbose=100)

  train_pred = estimator.predict(X_dtc_train)
  val_pred = estimator.predict(X_dtc_val)
  test_pred = estimator.predict(X_dtc_test)

  varinace_train = explained_variance_score(Y_dtc_train, train_pred)
  max_eror_train = max_error(Y_dtc_train, train_pred)
  ms_error_train = np.sqrt(mean_squared_error(Y_dtc_train, train_pred))
  mabs_error_train = mean_absolute_error(Y_dtc_train, train_pred)
  r2_train = r2_score(Y_dtc_train, train_pred)

  varinace_val = explained_variance_score(Y_dtc_val, val_pred)
  max_eror_val = max_error(Y_dtc_val, val_pred)
  ms_error_val = np.sqrt(mean_squared_error(Y_dtc_val, val_pred))
  mabs_error_val = mean_absolute_error(Y_dtc_val, val_pred)
  r2_val = r2_score(Y_dtc_val, val_pred)

  varinace_test = explained_variance_score(Y_dtc_test, test_pred)
  max_eror_test = max_error(Y_dtc_test, test_pred)
  ms_error_test = np.sqrt(mean_squared_error(Y_dtc_test, test_pred))
  mabs_error_test = mean_absolute_error(Y_dtc_test, test_pred)
  r2_test = r2_score(Y_dtc_test, test_pred)

  rme_errors.append(ms_error_test) # storing root mean squared error for model comparition

  print('''--------TRAINING SET METRICS--------
  explianed varianve {},
  maximum error {},
  root mean squared error {},
  maximum absolute error {},
  R2 {}'''.format(varinace_train, max_eror_train, ms_error_train, mabs_error_train, r2_train))

```

```
    print('''-------VALIDATION SET METRICS-------
    explianed varianve {},
    maximum error {},
    root mean squared error{},
    maximum absolute error {},
    R2 {}'''.format(varinace_val, max_eror_val, ms_error_val, mabs_error_val, r2_val))

    print('''-----------TEST SET METRICS----------
    explianed varianve {},
    maximum error {},
    root mean squared error {},
    maximum absolute error {},
    R2 {}'''.format(varinace_test, max_eror_test, ms_error_test, mabs_error_test, r2_test))

    i += 1
  # selecting bet model to perform imputation
  selected_model_index = rme_errors.index(min(rme_errors))
  print('TRAINING BEST MODEL TO PERFORM DTC IMPUTATION {}'.format(estimator_names[selected_model_index]))
  model3000 = estimators[selected_model_index]
  model3000.fit(X_dtc_train,
                Y_dtc_train.values.ravel(),
                early_stopping_rounds=100,
                eval_set=[(X_dtc_test, Y_dtc_test)],
                verbose=100) # fitting best ML regressor

  # filling nan values before predicting DTC
  X_train_DTC = traindata.drop(['LITHO', 'DTS', 'DTS_pred', 'NPHI', 'NPHI_pred', 'RHOB', 'RHOB_pred', 'DTC'], axis=1)
  X_train_DTC2 = X_train_DTC.apply(lambda x: x.fillna(x.median()), axis=0)

  X_test_DTC = testdata.drop(['LITHO', 'DTS', 'DTS_pred', 'NPHI', 'NPHI_pred', 'RHOB', 'RHOB_pred', 'DTC'], axis=1)
  X_test_DTC2 = X_test_DTC.apply(lambda x: x.fillna(x.median()), axis=0)


  X_hidden_DTC = hiddendata.drop(['LITHO', 'DTS', 'DTS_pred', 'NPHI', 'NPHI_pred', 'RHOB', 'RHOB_pred', 'DTC'], axis=1)
  X_hidden_DTC2 = X_hidden_DTC.apply(lambda x: x.fillna(x.median()), axis=0)

  # predict DTC on complete datasets
  traindata['DTC_pred'] = model3000.predict(X_train_DTC2)
  testdata['DTC_pred'] = model3000.predict(X_test_DTC2)
  hiddendata['DTC_pred'] = model3000.predict(X_hidden_DTC2)

  # imputing nan values in DTC by ML predictions
  traindata['DTC_COMB'] = traindata['DTC']
  traindata['DTC_COMB'].fillna(traindata['DTC_pred'], inplace=True)

  testdata['DTC_COMB'] = testdata['DTC']
  testdata['DTC_COMB'].fillna(testdata['DTC_pred'], inplace=True)

  hiddendata['DTC_COMB'] = hiddendata['DTC']
  hiddendata['DTC_COMB'].fillna(hiddendata['DTC_pred'], inplace=True)


  """Creating additional well log-based features
  """

  print('-------------------------------Creating additional features-------------------------------')

  # training Set
```

```
    traindata['S_I'] = traindata.RHOB * (1e6/traindata.DTS_COMB) # s-impedance
    traindata['P_I'] = traindata.RHOB * (1e6/traindata.DTC) #p-impedance
    traindata['DT_R'] = traindata.DTC / traindata.DTS_COMB # slowness ratio
    traindata['G'] = ((1e6/traindata.DTS_COMB)**2) * traindata.RHOB #Shear modulus
    traindata['K'] = (((1e6/traindata.DTC)**2) * traindata.RHOB) - (4 * traindata.G/3) # bulk modulus
    traindata['MD_TVD'] = -(traindata.DEPTH_MD/traindata.Z_LOC) #MD-TVD ratio

    # test Set
    testdata['S_I'] = testdata.RHOB * (1e6/testdata.DTS_COMB)
    testdata['P_I'] = testdata.RHOB * (1e6/testdata.DTC)
    testdata['DT_R'] = testdata.DTC / testdata.DTS_COMB
    testdata['G'] = ((1e6/testdata.DTS_COMB)**2) * testdata.RHOB
    testdata['K'] = (((1e6/testdata.DTC)**2) * testdata.RHOB) - (4 * testdata.G/3)
    testdata['MD_TVD'] = -(testdata.DEPTH_MD/testdata.Z_LOC)

    # hidden Set
    hiddendata['S_I'] = hiddendata.RHOB * (1e6/hiddendata.DTS_COMB)
    hiddendata['P_I'] = hiddendata.RHOB * (1e6/hiddendata.DTC)
    hiddendata['DT_R'] = hiddendata.DTC / hiddendata.DTS_COMB
    hiddendata['G'] = ((1e6/hiddendata.DTS_COMB)**2) * hiddendata.RHOB
    hiddendata['K'] = (((1e6/hiddendata.DTC)**2) * hiddendata.RHOB) - (4 * hiddendata.G/3)
    hiddendata['MD_TVD'] = -(hiddendata.DEPTH_MD/hiddendata.Z_LOC)

    # print column names held on datasets
    print('Features included in the datasets: {}'.format(traindata.columns))
    return traindata, testdata, hiddendata
```

8.1.6 Data Normalization (input_norm.py)

```

"""Data Normalization

This script standardize the training, open test, and hidden test sets
after the datasets have been pre-process and machine-learning augmented.
It provides the datastes ready to be used in any machine-learning model.

"""

def normalization(traindata, testdata, hiddendata):

  """Returns the starndardize training, open test, and hidden test
  dataframes once they have been pre-processed and augmented.

  Parameters
  ----------
  traindata: Dataframe
    Augmented trainig dataframe.
  testdata: Dataframe
    Augmented open test dataframe.
  hiddendata: Dataframe
    Augmented hidden test dataframe.

  Returns
  ----------
  cleaned_traindata: Dataframe
    Starndardized training dataframe.
  cleaned_testdata: Dataframe
    Starndardized open test dataframe.
  cleaned_hiddendata: Dataframe
    Starndardized hidden test dataframe.
  """


  import pandas as pd
  from sklearn.preprocessing import StandardScaler

  """The features that were not augmented
  by ML are inputed by the median and then standardized.
  """

  train_features = traindata.drop(['LITHO'], axis=1);   train_labels = traindata['LITHO']
  test_features = testdata.drop(['LITHO'], axis=1);     test_labels = testdata['LITHO']
  hidden_features = hiddendata.drop(['LITHO'], axis=1); hidden_labels = hiddendata['LITHO']

  #Imputng features by median
  train_features_inp = train_features.apply(lambda x: x.fillna(x.median()), axis=0)
  test_features_inp = test_features.apply(lambda x: x.fillna(x.median()), axis=0)
  hidden_features_inp = hidden_features.apply(lambda x: x.fillna(x.median()), axis=0)

  """Normalizing features on each dataset
  """

  n = train_features_inp.shape[1]
  std = StandardScaler()
```

```
x_train_std = train_features_inp.copy()
x_test_std = test_features_inp.copy()
x_hidden_std = hidden_features_inp.copy()

x_train_std.iloc[:,:n] = std.fit_transform(x_train_std.iloc[:,:n])
x_test_std.iloc[:,:n] = std.transform(x_test_std.iloc[:,:n])
x_hidden_std.iloc[:,:n] = std.transform(x_hidden_std.iloc[:,:n])

"""Concatenating features and targets
"""

cleaned_traindata = pd.concat([x_train_std, train_labels], axis=1)
cleaned_testdata = pd.concat([x_test_std, test_labels], axis=1)
cleaned_hiddendata = pd.concat([x_hidden_std, hidden_labels], axis=1)

return cleaned_traindata, cleaned_testdata, cleaned_hiddendata
```

8.2 Appendix B – Machine and Deep Learning Models Python Code

8.2.1 Logistic Regression (LR_model.py)

```

"""Logistic regression machine-learning model

This script trains a logistic regression machine learning model and test it on the
open test and hidden test dataset. The function retuns the lithofacies predictions
obtained for the training, open test, and hidden test sets.
"""

def run_LR(train_norm, test_norm, hidden_norm):

  """Returns the predicted lithology classes for the training,
  open test, and hidden test obtained by Logistic Regression.

  Parameters
  ----------
  cleaned_traindata: Dataframe
    Starndardized training dataframe.
  cleaned_testdata: Dataframe
    Starndardized open test dataframe.
  cleaned_hiddendata: Dataframe
    Starndardized hidden test dataframe.

  Returns
  ----------
  train_pred_lr: one-dimentional array
    Predicted lithology classes obtained from the training dataset.
  test_pred_lr: one-dimentional array
    Predicted lithology classes obtained from the open test dataset.
  hidden_pred_lr: one-dimentional array
    Predicted lithology classes obtained from the hidden test dataset.

  """

  from sklearn.model_selection import train_test_split
  from sklearn.linear_model import LogisticRegression

  # selected features to be used while training
  features_selected_lr = ['DTS_COMB', 'G', 'P_I', 'GR','NPHI_COMB',
                          'DTC', 'RHOB', 'DT_R', 'Z_LOC', 'S_I','K'
                          ]

  x_train = train_norm[features_selected_lr]
  y_train = train_norm['LITHO']

  x_test = test_norm[features_selected_lr]
  y_test = test_norm['LITHO']

  x_hidden = hidden_norm[features_selected_lr]
  y_hidden = hidden_norm['LITHO']

  x_train_strat, X2, y_train_strat, Y2 = train_test_split(x_train,
                                                          y_train,
                                                          train_size=0.1,
                                                          shuffle=True,
                                                          stratify=y_train,
                                                          random_state=0)
    # difining model with optimal hyper-parameters
    model_lr = LogisticRegression(C=0.1,
                                  solver='saga',
                                  max_iter=4000,
                                  verbose=1
                                  )

    # fitting a logistic regression model
    model_lr.fit(x_train_strat[features_selected_lr], y_train_strat)

    # predicting
    train_pred_lr = model_lr.predict(x_train[features_selected_lr])
    test_pred_lr = model_lr.predict(x_test[features_selected_lr])
    hidden_pred_lr = model_lr.predict(x_hidden[features_selected_lr])

    return train_pred_lr, test_pred_lr, hidden_pred_lr
```

8.2.2 K-Nearest Neighbors (KNN_model.py)

```

"""K-Nearest Neighbors machine-learning model

This script receives the clean datasets and trains a K-nearest neighbor
machine-learning model and test it on the clean open test and hidden test
datasets. The function retuns the lithofacies predictions obtained for the
training, open test, and hidden test sets.
"""

def run_KNN(train_norm, test_norm, hidden_norm):

  """Returns the predicted lithology classes for the training,
  open test, and hidden test obtained by K-nearesr Neighbors.

  Parameters
  ----------
  cleaned_traindata: Dataframe
    Starndardized training dataframe.
  cleaned_testdata: Dataframe
    Starndardized open test dataframe.
  cleaned_hiddendata: Dataframe
    Starndardized hidden test dataframe.

  Returns
  ----------
  train_pred_knn: one-dimentional array
    Predicted lithology classes obtained from the training dataset.
  test_pred_knn: one-dimentional array
    Predicted lithology classes obtained from the open test dataset.
  hidden_pred_knn: one-dimentional array
    Predicted lithology classes obtained from the hidden test dataset.
  """

  from sklearn.model_selection import train_test_split
  from sklearn import neighbors

```

```
    # selected features to be used while training
    selectedfeatures_knn = ['GR', 'FORMATION_encoded', 'GROUP_encoded', 'NPHI_COMB', 'RHOB',
                            'X_LOC', 'BS', 'CALI', 'SP', 'WELL_encoded', 'Z_LOC', 'DT_R', 'DEPTH_MD',
                            'DTC', 'Cluster']

    x_train = train_norm[selectedfeatures_knn]
    y_train = train_norm['LITHO']

    x_test = test_norm[selectedfeatures_knn]
    y_test = test_norm['LITHO']

    x_hidden = hidden_norm[selectedfeatures_knn]
    y_hidden = hidden_norm['LITHO']

    x_train_strat, X2, y_train_strat, Y2 = train_test_split(x_train,
                                                            y_train,
                                                            train_size=0.1,
                                                            shuffle=True,
                                                            stratify=y_train,
                                                            random_state=0
                                                            )
    # defining KNN mdoel with optimal hyper-parameters
    model_knn = neighbors.KNeighborsClassifier(n_neighbors=80,
                                            weights='distance',
                                            metric='manhattan'
                                            )
    # fitting a logistic regression model
    model_knn.fit(x_train_strat[selectedfeatures_knn], y_train_strat)

    # predicting
    train_pred_knn = model_knn.predict(x_train[selectedfeatures_knn])
    test_pred_knn = model_knn.predict(x_test[selectedfeatures_knn])
    hidden_pred_knn = model_knn.predict(x_hidden[selectedfeatures_knn])

    return train_pred_knn, test_pred_knn, hidden_pred_knn
```

8.2.3 Support Vector Machines (SVM_model.py)

```

"""Support Vector Machines machine-learning model

This script receives the clean datasets and trains a SVM machine-learning
model and test it on the clean open test and hidden test datasets.
The function returns the lithofacies predictions obtained for the training,
open test, and hidden test sets.
"""

def run_SVM(train_norm, test_norm, hidden_norm):

  """Returns the predicted lithology classes for the training,
  open test, and hidden test obtained by SVM.

  Parameters
  ----------
  cleaned_traindata: Dataframe
```

```
      Starndardized training dataframe.
    cleaned_testdata: Dataframe
      Starndardized open test dataframe.
    cleaned_hiddendata: Dataframe
      Starndardized hidden test dataframe.

    Returns
    ----------
    train_pred_svm: one-dimentional array
      Predicted lithology classes obtained from the training dataset.
    test_pred_svm: one-dimentional array
      Predicted lithology classes obtained from the open test dataset.
    hidden_pred_svm: one-dimentional array
      Predicted lithology classes obtained from the hidden test dataset.
    """

    from sklearn.model_selection import train_test_split
    from sklearn.svm import SVC

    x_train = train_norm.drop(['LITHO'], axis=1)
    y_train = train_norm['LITHO']

    x_test = test_norm.drop(['LITHO'], axis=1)
    y_test = test_norm['LITHO']

    x_hidden = hidden_norm.drop(['LITHO'], axis=1)
    y_hidden = hidden_norm['LITHO']

    x_train_strat, X2, y_train_strat, Y2 = train_test_split(x_train,
                                                            y_train,
                                                            train_size=0.1,
                                                            shuffle=True,
                                                            stratify=y_train,
                                                            random_state=0
                                                            )
    # Defining SVM model with optimal hyper-parameters
    model_svm = SVC(kernel='rbf',
                    C=0.5,
                    cache_size=5000,
                    decision_function_shape='ovr'
                    )
    # fitting SVM model
    model_svm.fit(x_train_strat, y_train_strat)

    # predicting
    train_pred_svm = model_svm.predict(x_train)
    test_pred_svm = model_svm.predict(x_test)
    hidden_pred_svm = model_svm.predict(x_hidden)

    return train_pred_svm, test_pred_svm, hidden_pred_svm
```

8.2.4 Decision Tree (DT_model.py)

```

"""Decision Tree machine-learning model

This script receives the clean datasets and trains a decision tree machine
learning model and test it on the clean open test and hidden test datasets.
The function returns the lithofacies predictions obtained for the training,
open test, and hidden test sets.
"""

def run_DT(train_norm, test_norm, hidden_norm):

  """Returns the predicted lithology classes for the training,
  open test, and hidden test obtained by a decision tree.

  Parameters
  ----------
  cleaned_traindata: Dataframe
    Starndardized training dataframe.
  cleaned_testdata: Dataframe
    Starndardized open test dataframe.
  cleaned_hiddendata: Dataframe
    Starndardized hidden test dataframe.

  Returns
  ----------
  train_pred_dtp: one-dimentional array
    Predicted lithology classes obtained from the training dataset.
  open_pred_dtp: one-dimentional array
    Predicted lithology classes obtained from the open test dataset.
  hidden_pred_dtp: one-dimentional array
    Predicted lithology classes obtained from the hidden test dataset.
  """

  from sklearn.model_selection import train_test_split
  from sklearn.tree import DecisionTreeClassifier

  x_train = train_norm.drop(['LITHO'], axis=1)
  y_train = train_norm['LITHO']

  x_test = test_norm.drop(['LITHO'], axis=1)
  y_test = test_norm['LITHO']

  x_hidden = hidden_norm.drop(['LITHO'], axis=1)
  y_hidden = hidden_norm['LITHO']

  x_train_strat, X2, y_train_strat, Y2 = train_test_split(x_train,
                                                          y_train,
                                                          train_size=0.1,
                                                          shuffle=True,
                                                          stratify=y_train,
                                                          random_state=0
                                                          )
  # defining DT model after pruning
  tunned_dt = DecisionTreeClassifier(max_depth=15,
                                     ccp_alpha=0.002
                                     )

    # fitting the decision tree model
    tunned_dt.fit(x_train_strat, y_train_strat)

    # predicting
    train_pred_dtp = tunned_dt.predict(x_train)
    open_pred_dtp = tunned_dt.predict(x_test)
    hidden_pred_dtp = tunned_dt.predict(x_hidden)

    return train_pred_dtp, open_pred_dtp, hidden_pred_dtp
```

8.2.5 Random Forest (RF_model.py)

```
"""Random Forest machine-learning model

This script receives the clean datasets and trains a random forest machine
learning model and test it on the clean open test and hidden test datasets.
The function returns the lithofacies predictions obtained for the training,
open test, and hidden test sets.
"""

def run_RF(train_norm, test_norm, hidden_norm):

  """Returns the predicted lithology classes for the training,
  open test, and hidden test obtained by a random forest.

  Parameters
  ----------
  cleaned_traindata: Dataframe
    Starndardized training dataframe.
  cleaned_testdata: Dataframe
    Starndardized open test dataframe.
  cleaned_hiddendata: Dataframe
    Starndardized hidden test dataframe.

  Returns
  ----------
  train_pred_rf: one-dimentional array
    Predicted lithology classes obtained from the training dataset.
  open_pred_rf: one-dimentional array
    Predicted lithology classes obtained from the open test dataset.
  hidden_pred_rf: one-dimentional array

    Predicted lithology classes obtained from the hidden test dataset.
  """

  from sklearn.model_selection import train_test_split
  from sklearn.ensemble import RandomForestClassifier

  # selected features to be used while training
  features_selected_rf = ['RDEP', 'GR', 'NPHI_COMB', 'G', 'P_I', 'S_I', 'DTC', 'DTS_COMB',
                          'RSHA', 'DT_R', 'RHOB', 'K', 'DCAL', 'Y_LOC', 'GROUP_encoded',
                          'WELL_encoded', 'FORMATION_encoded', 'DEPTH_MD', 'Z_LOC', 'CALI',
                          'X_LOC', 'RMED', 'PEF', 'SP', 'MD_TVD', 'ROP', 'DRHO']

  x_train = train_norm[features_selected_rf]
  y_train = train_norm['LITHO']
```

```

    x_test = test_norm[features_selected_rf]
    y_test = test_norm['LITHO']

    x_hidden = hidden_norm[features_selected_rf]
    y_hidden = hidden_norm['LITHO']

    x_train_strat, X2, y_train_strat, Y2 = train_test_split(x_train,
                                                            y_train,
                                                            train_size=0.5,
                                                            shuffle=True,
                                                            stratify=y_train,
                                                            random_state=0)
    # defining a RF model with the optimal hyper-parameters
    model_rf = RandomForestClassifier(n_estimators=350,
                                      bootstrap=False,
                                      max_depth=45,
                                      max_features='sqrt'
                                      )

    # fitting the random forest model
    model_rf.fit(x_train_strat[features_selected_rf], y_train_strat.values.ravel())

    # predicting
    train_pred_rf = model_rf.predict(x_train[features_selected_rf])
    open_pred_rf = model_rf.predict(x_test[features_selected_rf])
    hidden_pred_rf = model_rf.predict(x_hidden[features_selected_rf])

    return train_pred_rf, open_pred_rf, hidden_pred_rf
```

8.2.6 Categorical Gradient Boosting (CatBoost_model.py)

```

"""Categorical gradient boosting tree-based machine-learning model

This script receives the clean datasets and trains a categorical tree gradient
boosting machine learning model and test it on the clean open test and hidden
test datasets. The function returns the lithofacies predictions obtained for
the training, open test, and hidden test sets.
"""

def run_CatBoost(train_norm, test_norm, hidden_norm):

  """Returns the predicted lithology classes for the training,
  open test, and hidden test obtained by a categorical tree-based
  gradient boosting model, CAT.

  Parameters
  ----------
  cleaned_traindata: Dataframe
    Starndardized training dataframe.
  cleaned_testdata: Dataframe
    Starndardized open test dataframe.
  cleaned_hiddendata: Dataframe
    Starndardized hidden test dataframe.

  Returns
  ----------
  train_pred_cat1: one-dimentional array
    Predicted lithology classes obtained from the training dataset.
  open_pred_cat1: one-dimentional array
    Predicted lithology classes obtained from the open test dataset.
  hidden_pred_cat1: one-dimentional array
    Predicted lithology classes obtained from the hidden test dataset.
  """

  from sklearn.model_selection import StratifiedKFold
  from catboost import CatBoostClassifier
  from sklearn.metrics import accuracy_score
  import pandas as pd
  import numpy as np

  # selected features to be used while training
  selected_features_catboost = ['GR', 'NPHI_COMB', 'DTC', 'DTS_COMB','RHOB',
                                'Y_LOC', 'GROUP_encoded', 'WELL_encoded',
                                'FORMATION_encoded', 'DEPTH_MD', 'Z_LOC', 'CALI',
                                'X_LOC', 'RMED', 'SP', 'MD_TVD']

  x_train = train_norm[selected_features_catboost]
  y_train = train_norm['LITHO']

  x_test = test_norm[selected_features_catboost]
  y_test = test_norm['LITHO']

  x_hidden = hidden_norm[selected_features_catboost]
  y_hidden = hidden_norm['LITHO']

  """ The model is trained on 10 stratified k-folds, also uses the open set as
```

```
validation set to avoid overfitting and a 100-round early stopping callback.

The model uses a multi-soft_probability objective function which returns the
probabilities predicted for each class. This probabilities are computed and
stacked by using each k-fold to give the final prediction.
"""

split = 10
kf = StratifiedKFold(n_splits=split, shuffle=True)

train_prob_cat1 = np.zeros((len(x_train), 12))
open_prob_cat1 = np.zeros((len(x_test), 12))
hidden_prob_cat1 = np.zeros((len(x_hidden), 12))

catboost_model1 = CatBoostClassifier(iterations=1000,
                             learning_rate=0.1,
                             depth = 6,
                             l2_leaf_reg = 300,
                             #border_count = 128,
                             #bagging_temperature = 10,
                             grow_policy = 'SymmetricTree',
                             task_type='GPU',
                             verbose=100)
i = 1
for (train_index, test_index) in kf.split(x_train, y_train):
  X_train, X_test = x_train.iloc[train_index], x_train.iloc[test_index]
  Y_train, Y_test = y_train.iloc[train_index], y_train.iloc[test_index]

  catboost_model1.fit(X_train,
                      Y_train.values.ravel(),
                      early_stopping_rounds=100,
                      eval_set=[(X_test, Y_test)],

                      verbose=100
                      )

  prediction = catboost_model1.predict(X_test)
  print('Fold accuracy:', accuracy_score(Y_test, prediction))

  print(f'-----------------------FOLD {i}--------------------')

  # staking predicted probabilities
  train_prob_cat1 += catboost_model1.predict_proba(x_train)
  open_prob_cat1 += catboost_model1.predict_proba(x_test)
  hidden_prob_cat1 += catboost_model1.predict_proba(x_hidden)

  i += 1

# getting final predicted classes
train_prob_cat1 = pd.DataFrame(train_prob_cat1/split)
train_pred_cat1 = np.array(pd.DataFrame(train_prob_cat1).idxmax(axis=1))

open_prob_cat1 = pd.DataFrame(open_prob_cat1/split)
open_pred_cat1 = np.array(pd.DataFrame(open_prob_cat1).idxmax(axis=1))

hidden_prob_cat1 = pd.DataFrame(hidden_prob_cat1/split)
hidden_pred_cat1 = np.array(pd.DataFrame(hidden_prob_cat1).idxmax(axis=1))

return train_pred_cat1, open_pred_cat1, hidden_pred_cat1
```

8.2.7 Extreme Gradient Boosting (XGB_model.py)

```python

"""eXtreme gradient boosting tree-based machine-learning model

This script receives the clean datasets and trains an extreme gradient boosting
tree-based machine learning model and test it on the clean open test and hidden
test datasets. The function returns the lithofacies predictions obtained for
the training, open test, and hidden test sets.
"""

def run_XGB(train_norm, test_norm, hidden_norm):

  """Returns the predicted lithology classes for the training,
  open test, and hidden test obtained by a extreme gradient boosting
  tree-based model, XGB.

  Parameters
  ----------
  cleaned_traindata: Dataframe
    Starndardized training dataframe.
  cleaned_testdata: Dataframe
    Starndardized open test dataframe.
  cleaned_hiddendata: Dataframe
    Starndardized hidden test dataframe.

  Returns
  ----------
  train_pred_xgb1: one-dimentional array
    Predicted lithology classes obtained from the training dataset.
  open_pred_xgb1: one-dimentional array
    Predicted lithology classes obtained from the open test dataset.
  hidden_pred_xgb1: one-dimentional array
    Predicted lithology classes obtained from the hidden test dataset.

  """

  from xgboost import XGBClassifier
  from sklearn.model_selection import StratifiedKFold
  import numpy as np
  import pandas as pd
  from sklearn.metrics import accuracy_score

  # selected features to be used while training
  selected_fetures_xgb = ['RDEP', 'GR', 'NPHI_COMB', 'G', 'P_I', 'DTC', 'DTS_COMB', 'RSHA',
                          'DT_R', 'RHOB', 'K', 'DCAL', 'Y_LOC', 'Cluster', 'GROUP_encoded',
                          'WELL_encoded', 'FORMATION_encoded', 'DEPTH_MD', 'Z_LOC', 'CALI', 'BS',
                          'X_LOC', 'RMED', 'PEF', 'SP', 'MD_TVD', 'RMIC', 'DRHO']

  x_train = train_norm[selected_fetures_xgb]
  y_train = train_norm['LITHO']

  x_test = test_norm[selected_fetures_xgb]
  y_test = test_norm['LITHO']

  x_hidden = hidden_norm[selected_fetures_xgb]
  y_hidden = hidden_norm['LITHO']

```

```
"""The model is trained on 10 stratified k-folds, also uses the open set as
validation set to avoid overfitting and a 100-round early stopping callback.

The model uses a multi-soft_probability objective function which returns the
probabilities predicted for each class. This probabilities are computed and
stacked by using each k-fold to give the final prediction.

"""

split = 10
kf = StratifiedKFold(n_splits=split, shuffle=True)

train_prob_xgb1 = np.zeros((len(x_train), 12))
open_prob_xgb1 = np.zeros((len(x_test), 12))
hidden_prob_xgb1 = np.zeros((len(x_hidden), 12))

xgbmodel_noarg = XGBClassifier(n_estimators=1000, max_depth=4,
                               booster='gbtree', objective='multi:softprob',
                               learning_rate=0.075, random_state=42,
                               subsample=1, colsample_bytree=1,
                               tree_method='gpu_hist', predictor='gpu_predictor',
                               verbose=2020, reg_lambda=1500
                               )
i = 1
for (train_index, test_index) in kf.split(x_train, y_train):
  X_train, X_test = x_train.iloc[train_index], x_train.iloc[test_index]
  Y_train, Y_test = y_train.iloc[train_index], y_train.iloc[test_index]

  xgbmodel_noarg.fit(X_train,
                     Y_train.values.ravel(),
                     early_stopping_rounds=100,
                     eval_set=[(X_test, Y_test)],

                     verbose=100
                     )

  prediction = xgbmodel_noarg.predict(X_test)
  print('Fold accuracy:', accuracy_score(Y_test, prediction))

  print(f'----------------------FOLD {i}--------------------')

  # stacking probabilities
  train_prob_xgb1 += xgbmodel_noarg.predict_proba(x_train)
  open_prob_xgb1 += xgbmodel_noarg.predict_proba(x_test)
  hidden_prob_xgb1 += xgbmodel_noarg.predict_proba(x_hidden)

  i += 1

# final lithology class prediction
train_prob_xgb1 = pd.DataFrame(train_prob_xgb1/split)
train_pred_xgb1 = np.array(pd.DataFrame(train_prob_xgb1).idxmax(axis=1))

open_prob_xgb1 = pd.DataFrame(open_prob_xgb1/split)
open_pred_xgb1 = np.array(pd.DataFrame(open_prob_xgb1).idxmax(axis=1))

hidden_prob_xgb1 = pd.DataFrame(hidden_prob_xgb1/split)
hidden_pred_xgb1 = np.array(pd.DataFrame(hidden_prob_xgb1).idxmax(axis=1))

return train_pred_xgb1, open_pred_xgb1, hidden_pred_xgb1
```

8.2.8 Neural Network (NN_model.py)

```

"""Neural Network model

This script receives the clean datasets and trains neural network and test it
on the clean open test and hidden test datasets. The function returns the
lithofacies predictions obtained for the training, open test, and hidden test sets.
"""

def run_NN(train_norm, test_norm, hidden_norm):

  """Returns the predicted lithology classes for the training,
  open test, and hidden test obtained by a bayesian optimized
  two-hidden layer neural network.

  Parameters
  ----------
  cleaned_traindata: Dataframe
    Starndardized training dataframe.
  cleaned_testdata: Dataframe
    Starndardized open test dataframe.
  cleaned_hiddendata: Dataframe
    Starndardized hidden test dataframe.

  Returns
  ----------
  train_nn2: one-dimentional array
    Predicted lithology classes obtained from the training dataset.
  open_nn2: one-dimentional array
    Predicted lithology classes obtained from the open test dataset.
  hidden_nn2: one-dimentional array
    Predicted lithology classes obtained from the hidden test dataset.
  """


 # importing dependencies
 import numpy as np
 import pandas as pd
 import tensorflow as tf
 from tensorflow import keras
 from tensorflow.keras.models import Sequential
 from tensorflow.keras.layers import Dense, Activation
 from tensorflow.keras.layers import InputLayer, Input
 from tensorflow.keras.layers import Dense, Dropout, Activation
 from tensorflow.keras.callbacks import EarlyStopping
 from tensorflow.keras.optimizers import SGD

 # selected features to be used while training
 features_selected_nn = ['GROUP_encoded', 'GR', 'NPHI_COMB', 'Y_LOC', 'RHOB',
                      'DEPTH_MD', 'FORMATION_encoded', 'Z_LOC', 'WELL_encoded', 'X_LOC',
                      'RMED', 'CALI', 'DTC', 'MD_TVD', 'DT_R',
                      'PEF', 'RDEP', 'DTS_COMB', 'G', 'SP',
                      'Cluster', 'K', 'P_I', 'DRHO', 'DCAL'
                        ]
 """hyper-parameters optimized through a Bayesian optimization process.
 """

 learning_rate = 0.1
```

```
num_layers = 2
num_nodes = 512
activation = 'sigmoid'

"""Structuring a two-hidden layer feed fordward neural network
"""

x_train_nn = tf.convert_to_tensor(train_norm[features_selected_nn])
x_test_nn = tf.convert_to_tensor(test_norm[features_selected_nn])
x_hidden_nn = tf.convert_to_tensor(hidden_norm[features_selected_nn])

opt_model = Sequential()
opt_model.add(InputLayer(input_shape=(x_train_nn.shape[1])))
opt_model.add(Dropout(0.1))

opt_model.add(Dense(num_nodes,
                    activation=activation,
                    kernel_initializer='random_normal')) #input layer

opt_model.add(Dropout(0.7)) # drop out layer for regularization

opt_model.add(Dense(num_nodes,
                    activation=activation,
                    kernel_initializer='random_normal')) # hidden layer 1

opt_model.add(Dense(12,
                    activation='softmax',
                    kernel_initializer='random_normal')) # hidden layer 2

# setting stochastic gradint descent optimizer
optimizer = SGD(learning_rate=learning_rate, momentum=0.1)

opt_model.compile(optimizer=optimizer,
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy']
                  ) # compiling model

monitor = EarlyStopping(monitor='val_loss',
                        min_delta=1e-3,
                        patience=50,
                        verbose=1,
                        mode='auto',
                        restore_best_weights=True
                        ) # early stopping callback to avoid overfitting

"""Fitting neural network while validation on the open test set
"""

histories = opt_model.fit(x_train_nn,
                          train_norm['LITHO'],
                          batch_size = 256,
                          validation_data = (x_test_nn, test_norm['LITHO']),
                          callbacks = [monitor],
                          verbose=1,
                          epochs=100
                          )

```

```
""" Predicting lithology classes
"""

nn_train_prob = opt_model.predict(x_train_nn)
train_nn2 = np.array(pd.DataFrame(nn_train_prob).idxmax(axis=1))

nn_open_prob = opt_model.predict(x_test_nn)
open_nn2 = np.array(pd.DataFrame(nn_open_prob).idxmax(axis=1))

nn_hidden_prob = opt_model.predict(x_hidden_nn)
hidden_nn2 = np.array(pd.DataFrame(nn_hidden_prob).idxmax(axis=1))

return train_nn2, open_nn2, hidden_nn2
```

8.3 Appendix C – Neural network Bayesian parameter optimization (Bayes_opt.py)

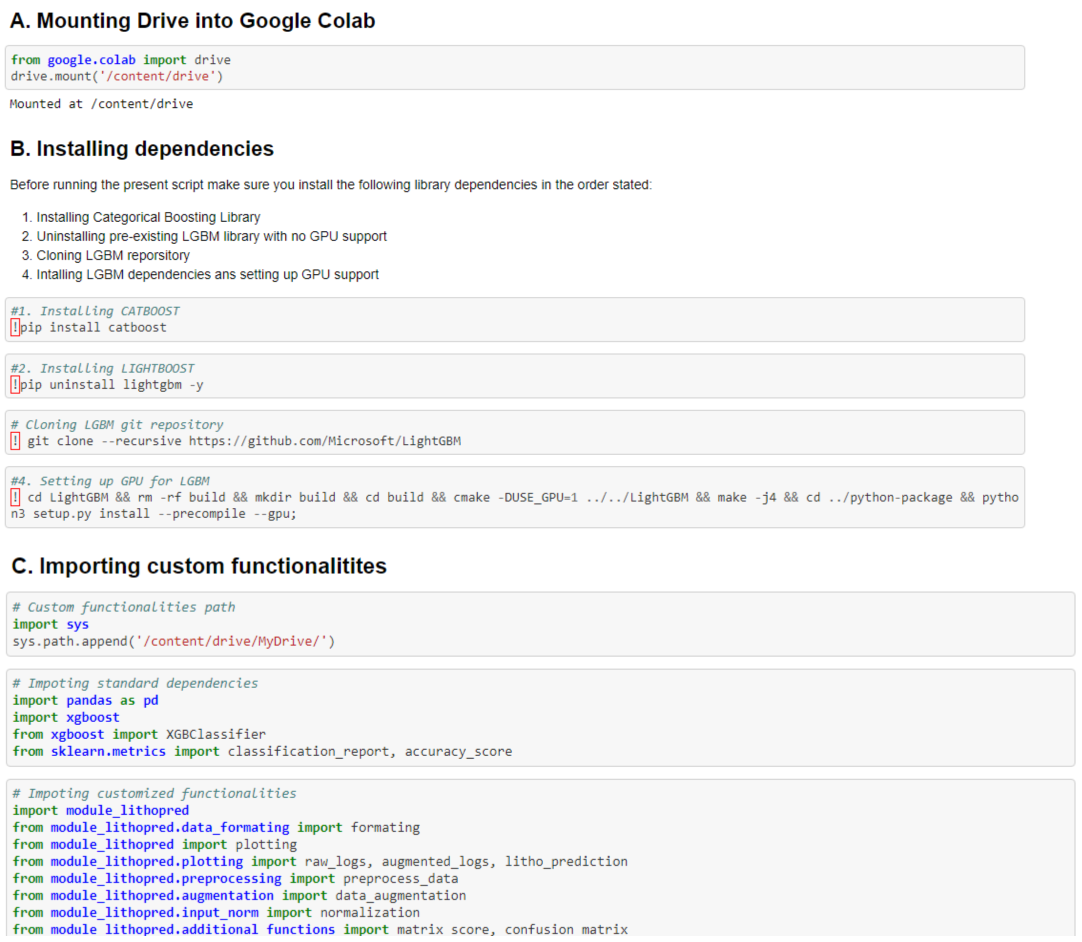
```
"""1. Setting search ranges for the parameters of interest.
"""

dim_learning_rate = Real(low=1e-4,
                         high=1e-1,
                         prior='log-uniform',
                         name='learning_rate')
dim_num_dense_layers = Integer(low=1,
                               high=5,
                               name='num_dense_layers')
dim_num_dense_nodes = Integer(low=64,
                              high=512,
                              name='num_dense_nodes')
dim_activation = Categorical(categories=['relu', 'sigmoid'], name='activation')

dimensions = [dim_learning_rate, dim_num_dense_layers,
              dim_num_dense_nodes, dim_activation] #Defining a list with all the parameters

defaut_parameters = [0.1, 2, 128, 'relu'] #Setting initial parameters

"""2. Defining a funtion top log the training process to visualize.
"""

def log_dir_name(learning_rate, num_dense_layers,
                 num_dense_nodes, activation):
    s = "./19_logs/lr_{0:.0e}_layers_{1}_nodes_{2}_{3}/" #Directory name

    log_dir = s.format(learning_rate, num_dense_layers,
                       num_dense_nodes, activation) #Directory name + parameters
    return log_dir



  """3. Defining a neural netwotk create model function
  """

def create_model(learning_rate, num_dense_layers, num_dense_nodes, activation):
  """Returns a tensor flow sequential fully connected neural network.
  It uses a stochastic gradien descent SGD optimizer that will be used to find it global minima.
    Parameters
    ----------
    learning_rate: int
      Step size to be taken by the optimizer towards its minima.
     num_dense_layers: int
      Number of hidden layers to be included in the model.
     num_dense_nodes: int
      Number of neurons to be inlcuded in each hidden layer.
     activation: str
      Activation function, either 'relu' or 'sigmoid.

    Returns
    ----------
    model:
      Neural network ready to be trained.
    """
  model = Sequential()
  num_features = x_train_nn.shape[1]
  model.add(layers.InputLayer(input_shape=(num_features, )))

  for i in range(num_dense_layers):                     #Adding hidden layers
    name = 'layer_dense_{0}'.format(i+1)
    model.add(Dense(num_dense_nodes,                    #Input layer
                    activation=activation,
                    name=name,
                    kernel_initializer='random_normal',  #Random normal weight initialization
                    bias_initializer='zeros'))           #Zeros bias initialization

  model.add(Dense(12,                                   #Output layer - 12 outputs
                  activation='softmax',                 #Softmax activation fuction
                  kernel_initializer='random_normal',   #Random normal weight initialization
                  bias_initializer='zeros'))            #Zeros bias initialization

  opt = SGD(learning_rate=learning_rate,                #Adam optimizer
            momentum=0.1)                               #momentum to help covergence

  model.compile(optimizer=opt,                          #Compiling the model
                loss='sparse_categorical_crossentropy',  #Sparse categorical crossentropy loss
                metrics=['accuracy'])
  return model



"""4. Defining optimization fitness function
"""
path_best_model = '19_best_model.h5' #Path where accuracy history will be stored
best_accuracy = 0 #Initializing global accuracy
validation_data = (x_test_nn, y_test) # Setting validations data

def fitness(learning_rate, num_dense_layers,
            num_dense_nodes, activation):
  """Fintion to be iterated several times by calling the create_model function and fitting the sequential fully
  connected neural network on a different set of parameters for 7 epochs on each, then it stores the best result
  and parameters on the assigned directory.
    Parameters
    ----------
    learning_rate: int
      Step size to be taken by the optimizer towards its minima.
     num_dense_layers: int
      Number of hidden layers to be included in the model.
     num_dense_nodes: int
      Number of neurons to be inlcuded in each hidden layer.
     activation: str
      Activation function, either 'relu' or 'sigmoid.

    Returns
    ----------
    -accuracy:
      The negative accuracy obtained on each set of hyper-parameters.
      The minus only alows us to treat the optimization as a minimization problem by using scikit optimizer skopt.
    """

    #Displaying selected hyper-parameters
    print("""learning rate: {0:.1e},
              num_dense_layers: {},
              num_dense_nodes: {},
              activation: {}""".format(learning_rate, num_dense_layers, num_dense_nodes, activation))

    #Calling create_model function
    model = create_model(learning_rate=learning_rate,
                         num_dense_layers=num_dense_layers,
                         num_dense_nodes=num_dense_nodes,
                         activation=activation)

    #Storing parameters on the assigned directory
    log_dir = log_dir_name(learning_rate, num_dense_layers,
                           num_dense_nodes, activation)

    #Defining a call back to be called during training to avoid overfitting
    callback_log = TensorBoard(log_dir=log_dir,
                               histogram_freq=0,
                               write_graph=True,
                               write_grads=False,
                               write_images=False)

    #Fitting the model for 7 epochs while validating on the open test data
    history = model.fit(x= x_train_nn,
                        y= y_train,
                        epochs=7,
                        batch_size=256,
                        validation_data=validation_data,
                        callbacks=[callback_log])

    # Displaying the validation accuracy after the 7th epoch
    accuracy = history.history['val_accuracy'][-1]
    print("Accuracy: {0:.2%}".format(accuracy))

    #Updating and storing the global accuracy if the selected hyper-parameters achieves to do so
    global best_accuracy
    if accuracy > best_accuracy:
        model.save(path_best_model)
        best_accuracy = accuracy

    #Clearing the model from memory before runnijng the next model
    del model
    K.clear_session()

    return -accuracy
# This function exactly comes from :Hvass-Labs, TensorFlow-Tutorials


"""5. Running optimization
"""
optimization = gp_minimize(func=fitness,
                            dimensions=dimensions,
                            acq_func='EI', # Expected Improvement.
                            n_calls=75,
                            x0=defaut_parameters)
```

```
optimization = gp_minimize(func=fitness,
                           dimensions=dimensions,
                           acq_func='EI', # Expected Improvement.
                           n_calls=75,
                           x0=defaut_parameters)

plot_convergence(optimization) #Displaying best set of hyper-paremeters
sorted(zip(optimization.func_vals, optimization.x_iters)) #Plotting convergence
```

8.4 Appendix D – Execution Python Code (Execution.py)

The current appendix shows how to set the environment necessary to run the functionalities and models included in appendices A and B. In addition, the scrip includes the sequential steps that must be taken in order to call each functionality needed and visualize each model's lithology prediction. Moreover, due to the extensiveness and repetitiveness involved in the process of calling each machine-learning model running function, only the best performing model, XGB, is included as an example for the present appendix.

To see the complete Execution.py file, please refer to the GitHub repository direction stated at the beginning of section 8.



A. Mounting Drive into Google Colab

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

B. Installing dependencies

Before running the present script make sure you install the following library dependencies in the order stated:

1. Installing Categorical Boosting Library
2. Uninstalling pre-existing LGBM library with no GPU support
3. Cloning LGBM reporsitory
4. Intalling LGBM dependencies ans setting up GPU support

```
#1. Installing CATBOOST
!pip install catboost
```

```
#2. Installing LIGHTBOOST
!pip uninstall lightgbm -y
```

```
# Cloning LGBM git repository
! git clone --recursive https://github.com/Microsoft/LightGBM
```

```
#4. Setting up GPU for LGBM
! cd LightGBM && rm -rf build && mkdir build && cd build && cmake -DUSE_GPU=1 ../../LightGBM && make -j4 && cd ../python-package && pytho
n3 setup.py install --precompile --gpu;
```

C. Importing custom functionalitites

```
# Custom functionalities path
import sys
sys.path.append('/content/drive/MyDrive/')
```

```
# Impoting standard dependencies
import pandas as pd
import xgboost
from xgboost import XGBClassifier
from sklearn.metrics import classification_report, accuracy_score
```

```
# Impoting customized functionalities
import module_lithopred
from module_lithopred.data_formating import formating
from module_lithopred import plotting
from module_lithopred.plotting import raw_logs, augmented_logs, litho_prediction
from module_lithopred.preprocessing import preprocess_data
from module_lithopred.augmentation import data_augmentation
from module_lithopred.input_norm import normalization
from module_lithopred.additional_functions import matrix_score, confusion_matrix
```

D. Importing data ¶

```
# Setting datasets directories
directory = '/content/drive/MyDrive/Thesis_data/'
```

```
"""Imporitng training, open test, and hidden test sets
"""
raw_training = pd.read_csv(directory + 'train.csv', sep=';') # rawtraining dataset

test_data = pd.read_csv(directory + 'test.csv', sep=';')
test_labels = pd.read_csv(directory + 'test_target.csv', sep=';')
raw_test = pd.merge(test_data, test_labels, on=['WELL', 'DEPTH_MD']) #open test dataset

raw_hidden = pd.read_csv(directory + 'hidden_test.csv', sep=';')  #hidden test dataset
```

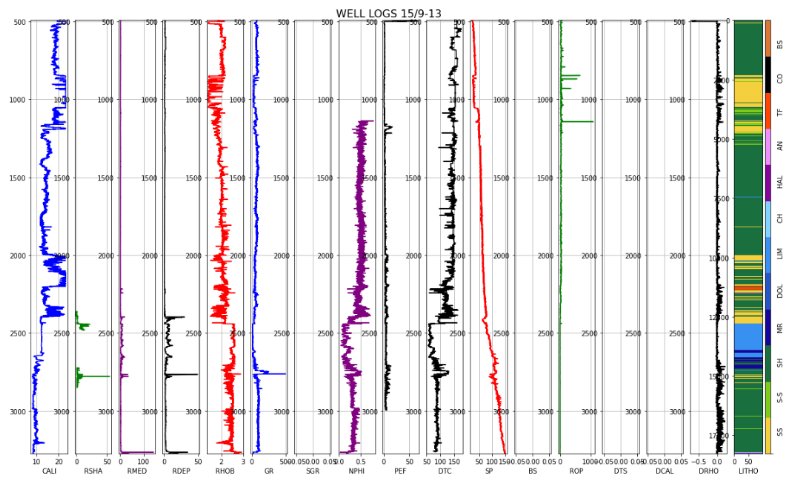
```
#Data formating

""" Calling formating function, which renames columns and maps lithofacies
classes values from 0 to 11, and drops intepretation confidence column.
"""

training_form, test_form, hidden_form = formating(raw_training, raw_test, raw_hidden)
```

```
#Inspecting raw data
display(training_form.head())
```

```
#Inspecting raw data
display(training_form.head())
```

	WELL	DEPTH_MD	X_LOC	Y_LOC	Z_LOC	GROUP	FORMATION	CALI	RSHA	RMED	RDEP	RHOB	GR	SGR	NPH
0	15/9-13	494.528	437641.96875	6470972.5	-469.501831	NORDLAND GP.	NaN	19.480835	NaN	1.611410	1.798681	1.884186	80.200851	NaN	NaN
1	15/9-13	494.680	437641.96875	6470972.5	-469.653809	NORDLAND GP.	NaN	19.468800	NaN	1.618070	1.795641	1.889794	79.262886	NaN	NaN
2	15/9-13	494.832	437641.96875	6470972.5	-469.805786	NORDLAND GP.	NaN	19.468800	NaN	1.626459	1.800733	1.896523	74.821999	NaN	NaN
3	15/9-13	494.984	437641.96875	6470972.5	-469.957794	NORDLAND GP.	NaN	19.459282	NaN	1.621594	1.801517	1.891913	72.878922	NaN	NaN
4	15/9-13	495.136	437641.96875	6470972.5	-470.109772	NORDLAND GP.	NaN	19.453100	NaN	1.602679	1.795299	1.880034	71.729141	NaN	NaN

```
""" Formated well logs visualization by calling raw_logs
customized function. Only the fist well displayed.

For plotting additionall wells change the range(0, 1).

See raw_logs.py for further details about the function.
"""
for i in range(0, 1):
  raw_logs(training_form, i)
```



** Only one well displayed for visualization.

E. Data Pre-processing

```
""" Preprocessing involves dropping unncesary cols, encoding categorical variables,
and incorporating well location clustering as feature prior to feature augmentation.

See perpocessing.py for further deatils.
"""

traindata, testdata, hiddendata = preprocess_data(training_form, test_form, hidden_form)
```

```
#Checking data structure befrore augmentation
display(traindata.head())
```

	Cluster	DEPTH_MD	X_LOC	Y_LOC	Z_LOC	CALI	RSHA	RMED	RDEP	RHOB	GR	NPHI	PEF	DTC	SP
0	0	494.528	437641.96875	6470972.5	-469.501831	19.480835	NaN	1.611410	1.798681	1.884186	80.200851	NaN	20.915468	161.131180	24.61
1	0	494.680	437641.96875	6470972.5	-469.653809	19.468800	NaN	1.618070	1.795641	1.889794	79.262886	NaN	19.383013	160.603470	23.89
2	0	494.832	437641.96875	6470972.5	-469.805786	19.468800	NaN	1.626459	1.800733	1.896523	74.821999	NaN	22.591518	160.173615	23.91
3	0	494.984	437641.96875	6470972.5	-469.957794	19.459282	NaN	1.621594	1.801517	1.891913	72.878922	NaN	32.191910	160.149429	23.79
4	0	495.136	437641.96875	6470972.5	-470.109772	19.453100	NaN	1.602679	1.795299	1.880034	71.729141	NaN	38.495632	160.128342	24.10

F. Feature augmentation by Machine-learning

```
"""First, each regressor takes as training data the 80% of the features
where the log being predicted is present, later each regresor is validated
on the remanent 20% of this data, and tested on the open test set.

Second, each regression model predictcs and imputes the predicted value
where the predicted log readings were originally missing on the training,
open, and hidden sets.

Finally, some other additional features are created.

See argumentation.py for further details.
"""

training_aug, test_aug, hidden_aug = data_augmentation(traindata, testdata, hiddendata)
```

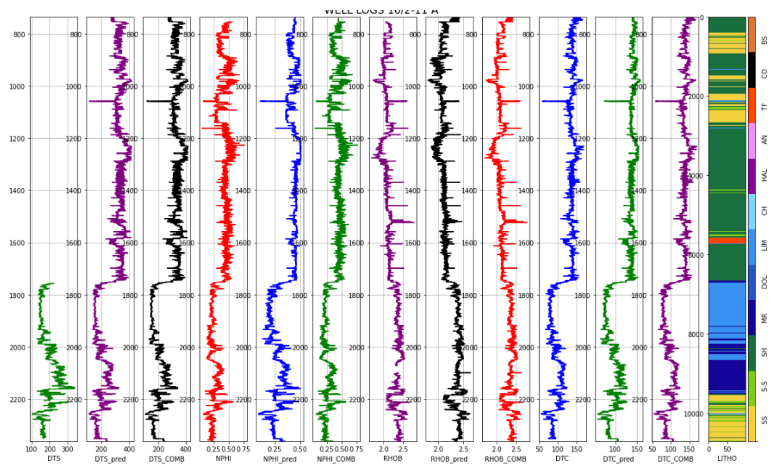
```
-----------------------------------------PREDINCTING DTS-----------------------------------------
----------------------EXTREME BOOST REGRESSOR MODEL----------------------
[22:15:02] WARNING: /workspace/src/objective/regression_obj.cu:152: reg:linear is now deprecated in favor of reg:squarederror.
[0]	validation_0-rmse:185.641
Will train until validation_0-rmse hasn't improved in 100 rounds.
[100]	validation_0-rmse:18.0697
[200]	validation_0-rmse:18.2223
Stopping. Best iteration:
[143]	validation_0-rmse:16.1557

--------TRAINING SET METRICS--------
    explianed varianve 0.9429831306549395,
    maximum error 320.7644653285937,
    root mean squared error 17.016241900143637,
    maximum absolute error 9.933835157182124,
```

```
# Inspecting dataframe after augmentation

display(training_aug.head())
```

	Cluster	DEPTH_MD	X_LOC	Y_LOC	Z_LOC	CALI	RSHA	RMED	RDEP	RHOB	GR	NPHI	PEF	DTC	SP
0	0	494.528	437641.96875	6470972.5	-469.501831	19.480835	NaN	1.611410	1.798681	1.884186	80.200851	NaN	20.915468	161.131180	24.61
1	0	494.680	437641.96875	6470972.5	-469.653809	19.468800	NaN	1.618070	1.795641	1.889794	79.262886	NaN	19.383013	160.603470	23.89
2	0	494.832	437641.96875	6470972.5	-469.805786	19.468800	NaN	1.626459	1.800733	1.896523	74.821999	NaN	22.591518	160.173615	23.91
3	0	494.984	437641.96875	6470972.5	-469.957794	19.459282	NaN	1.621594	1.801517	1.891913	72.878922	NaN	32.191910	160.149429	23.79
4	0	495.136	437641.96875	6470972.5	-470.109772	19.453100	NaN	1.602679	1.795299	1.880034	71.729141	NaN	38.495632	160.128342	24.10

```
"""Concatenating the actual, predited, and augmented well logs for plotting
"""

cols_needed = ['DTS', 'DTS_pred', 'DTS_COMB', 'NPHI', 'NPHI_pred', 'NPHI_COMB', 'RHOB',
               'RHOB_pred', 'RHOB_COMB', 'DTC', 'DTC_pred', 'DTC_COMB', 'LITHO']

train_predicted_logs = pd.concat((raw_training[['WELL', 'DEPTH_MD']],
                                  training_aug[cols_needed].reset_index()), axis=1)

test_predicted_logs = pd.concat((raw_test[['WELL', 'DEPTH_MD']],
                                 test_aug[cols_needed].reset_index()), axis=1)

hidden_predicted_logs = pd.concat((raw_hidden[['WELL', 'DEPTH_MD']],
                                   hidden_aug[cols_needed].reset_index()), axis=1)
```

```
""" Actual, predicted, and augmented well logs visualization by calling
augmented_logs customized function. Only the fist well displayed.

For plotting additionall wells change the range(0, 1).

See augmented_logs.py for further details about the function.
"""

for i in range(10, 11):
  augmented_logs(train_predicted_logs, i)
```



** Only one well displayed for feature augmentation visualization.

G. Data Normalization

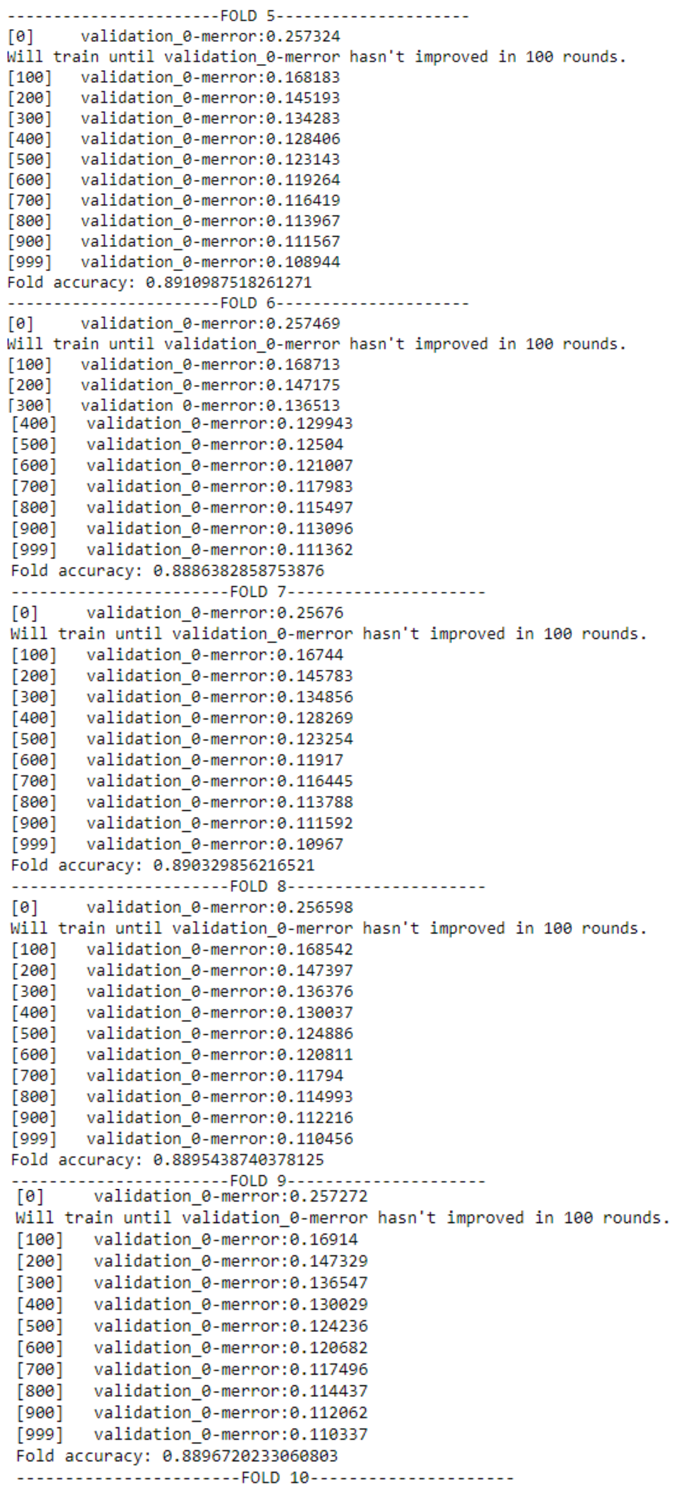
```
"""First, the features that were not augmented by machine-learning
are inputed by median inputation technique before normalizing

Later, a Standard Scaler is used to standardize the datasets.

See normalization.py for further details.
"""

train_norm, test_norm, hidden_norm = normalization(training_aug, test_aug, hidden_aug)
```

H. Machine learning models' results

A8. EXTREME GRADIENT BOOSTING MODEL (XGB)

```python
"""Predicting lithofacies by using the eXtreme Gradient Boosting model (XGB)
by calling run_XGB function.

See XGB_model.py for further details about the function.
"""
from module_lithopred.ML_models.XGB_model import run_XGB

train_pred_xgb, test_pred_xgb, hidden_pred_xgb = run_XGB(train_norm, test_norm, hidden_norm)
```

```
[0]	validation_0-merror:0.255724
Will train until validation_0-merror hasn't improved in 100 rounds.
[100]	validation_0-merror:0.166336
[200]	validation_0-merror:0.144799
[300]	validation_0-merror:0.134222
[400]	validation_0-merror:0.127405
[500]	validation_0-merror:0.123236
[600]	validation_0-merror:0.118896
[700]	validation_0-merror:0.116025
[800]	validation_0-merror:0.113394
[900]	validation_0-merror:0.110618
[999]	validation_0-merror:0.108601
Fold accuracy: 0.8914243242319653
----------------------FOLD 1--------------------
[0]	validation_0-merror:0.254915
Will train until validation_0-merror hasn't improved in 100 rounds.
[100]	validation_0-merror:0.16621
[200]	validation_0-merror:0.146329
[300]	validation_0-merror:0.136035
[400]	validation_0-merror:0.129012
[500]	validation_0-merror:0.123852
[600]	validation_0-merror:0.120213
[700]	validation_0-merror:0.116522
[800]	validation_0-merror:0.113925
[900]	validation_0-merror:0.111652
[999]	validation_0-merror:0.109841
Fold accuracy: 0.8901589905254974
----------------------FOLD 2--------------------
[0]	validation_0-merror:0.254692
Will train until validation_0-merror hasn't improved in 100 rounds.
[100]	validation_0-merror:0.167671
[200]	validation_0-merror:0.147004
[300]	validation_0-merror:0.136197
[400]	validation_0-merror:0.129636
[500]	validation_0-merror:0.125219
[600]	validation_0-merror:0.120947
[700]	validation_0-merror:0.117684
[800]	validation_0-merror:0.114762
[900]	validation_0-merror:0.112626
[999]	validation_0-merror:0.110465
Fold accuracy: 0.8895353307532614
----------------------FOLD 3--------------------
[0]	validation_0-merror:0.254308
Will train until validation_0-merror hasn't improved in 100 rounds.
[100]	validation_0-merror:0.168363
[200]	validation_0-merror:0.147414
[300]	validation_0-merror:0.136411
[400]	validation_0-merror:0.130063
[500]	validation_0-merror:0.124869
[600]	validation_0-merror:0.121272
[700]	validation_0-merror:0.118401
[800]	validation_0-merror:0.115548
[800]	validation_0-merror:0.115548
[900]	validation_0-merror:0.113438
[999]	validation_0-merror:0.111336
Fold accuracy: 0.8887237187208994
----------------------FOLD 4--------------------
[0]	validation_0-merror:0.253385
Will train until validation_0-merror hasn't improved in 100 rounds.
[100]	validation_0-merror:0.167397
[200]	validation_0-merror:0.146107
[300]	validation_0-merror:0.135778

[400]	validation_0-merror:0.129294
[500]	validation_0-merror:0.123972
[600]	validation_0-merror:0.120161
[700]	validation_0-merror:0.117146
[800]	validation_0-merror:0.114489
[900]	validation_0-merror:0.112344
[999]	validation_0-merror:0.11049
Fold accuracy: 0.889518244184159
```



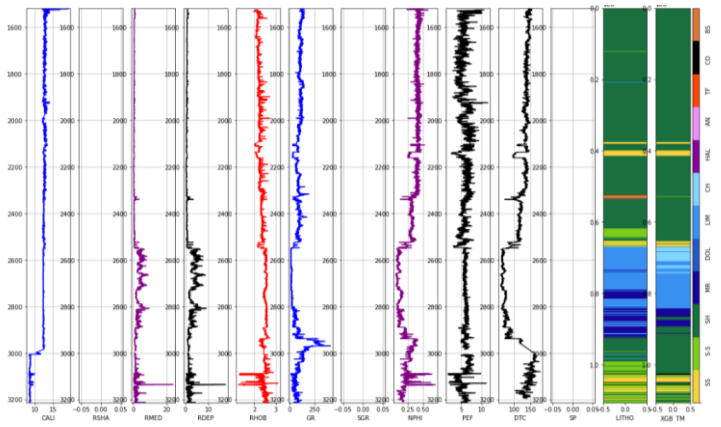
```
-----------------------FOLD 5---------------------
[0]     validation_0-merror:0.257324
Will train until validation_0-merror hasn't improved in 100 rounds.
[100]   validation_0-merror:0.168183
[200]   validation_0-merror:0.145193
[300]   validation_0-merror:0.134283
[400]   validation_0-merror:0.128406
[500]   validation_0-merror:0.123143
[600]   validation_0-merror:0.119264
[700]   validation_0-merror:0.116419
[800]   validation_0-merror:0.113967
[900]   validation_0-merror:0.111567
[999]   validation_0-merror:0.108944
Fold accuracy: 0.8910987518261271
-----------------------FOLD 6---------------------
[0]     validation_0-merror:0.257469
Will train until validation_0-merror hasn't improved in 100 rounds.
[100]   validation_0-merror:0.168713
[200]   validation_0-merror:0.147175
[300]   validation_0-merror:0.136513
[400]   validation_0-merror:0.129943
[500]   validation_0-merror:0.12504
[600]   validation_0-merror:0.121007
[700]   validation_0-merror:0.117983
[800]   validation_0-merror:0.115497
[900]   validation_0-merror:0.113096
[999]   validation_0-merror:0.111362
Fold accuracy: 0.8886382858753876
-----------------------FOLD 7---------------------
[0]     validation_0-merror:0.25676
Will train until validation_0-merror hasn't improved in 100 rounds.
[100]   validation_0-merror:0.16744
[200]   validation_0-merror:0.145783
[300]   validation_0-merror:0.134856
[400]   validation_0-merror:0.128269
[500]   validation_0-merror:0.123254
[600]   validation_0-merror:0.11917
[700]   validation_0-merror:0.116445
[800]   validation_0-merror:0.113788
[900]   validation_0-merror:0.111592
[999]   validation_0-merror:0.10967
Fold accuracy: 0.890329856216521
-----------------------FOLD 8---------------------
[0]     validation_0-merror:0.256598
Will train until validation_0-merror hasn't improved in 100 rounds.
[100]   validation_0-merror:0.168542
[200]   validation_0-merror:0.147397
[300]   validation_0-merror:0.136376
[400]   validation_0-merror:0.130037
[500]   validation_0-merror:0.124886
[600]   validation_0-merror:0.120811
[700]   validation_0-merror:0.11794
[800]   validation_0-merror:0.114993
[900]   validation_0-merror:0.112216
[999]   validation_0-merror:0.110456
Fold accuracy: 0.8895438740378125
-----------------------FOLD 9---------------------
[0]     validation_0-merror:0.257272
Will train until validation_0-merror hasn't improved in 100 rounds.
[100]   validation_0-merror:0.16914
[200]   validation_0-merror:0.147329
[300]   validation_0-merror:0.136547
[400]   validation_0-merror:0.130029
[500]   validation_0-merror:0.124236
[600]   validation_0-merror:0.120682
[700]   validation_0-merror:0.117496
[800]   validation_0-merror:0.114437
[900]   validation_0-merror:0.112062
[999]   validation_0-merror:0.110337
Fold accuracy: 0.8896720233060803
-----------------------FOLD 10---------------------
```

```
""""Plotting the classification report for the hidden test set
and the matrix penalty score.
"""

print('----------------------HIDDEN SET REPORT--------------------')
print('Hidden set penalty matrix score:', matrix_score(hidden_norm.LITHO.values, hidden_pred_xgb))
print('Hidden set report:', classification_report(hidden_norm.LITHO, hidden_pred_xgb))
```

```
----------------------HIDDEN SET REPORT--------------------
Hidden set penalty matrix score: -0.4471167593976977
Hidden set report:                precision    recall  f1-score   support

           0       0.76      0.86      0.80     14045
           1       0.66      0.42      0.52     12283
           2       0.88      0.94      0.91     71827
           3       0.27      0.31      0.29      4396
           4       0.12      0.05      0.07       287
           5       0.73      0.51      0.60      8374
           6       0.70      0.73      0.71      2905
           7       0.99      0.99      0.99      6498
           8       0.73      0.65      0.69       597
           9       0.55      0.44      0.49       941
          10       0.79      0.71      0.75       244

    accuracy                           0.82    122397
   macro avg       0.65      0.60      0.62    122397
weighted avg       0.81      0.82      0.81    122397
```

```
"""Storing XGB predctions into a copy of the formated datasets.
"""
train_xgb_res = training_form.copy()
test_xgb_res = test_form.copy()
hidden_xgb_res = hidden_form.copy()

train_xgb_res['XGB_TM'] = train_pred_xgb
test_xgb_res['XGB_TM'] = test_pred_xgb
hidden_xgb_res['XGB_TM'] = hidden_pred_xgb
```

```
""" Plotting hidden test set well logs, actual and predicted litholofacies by calling
litho_prediction customized function. Only the fist well is displayed.

For plotting additionall wells change the range(0, 1).

See litho_prediction.py for further details about the function.
"""

for i in range(0, 1):
  litho_prediction(hidden_xgb_res, i, 1)
```



**** Only well 15/9-23 belonging to the hidden test set is used for results visualization. Refer to Execution.ipynb to visualize the lithofacies prediction obtained by XGB for every well included in the open test and hidden test sets.

8.5 Appendix E – Experimentation Python code (Experimentation.ipynb)

Considering the extensiveness of the experimentation code, it was not included in the current endorsement. However, if any detail regarding, statistical visualization, feature selection, and hyper-parameter tuning that leaded to the final machine-learning models included in the present study is needed, this file as well as the other python appendices included in the current study can be found open sourced on GitHub.

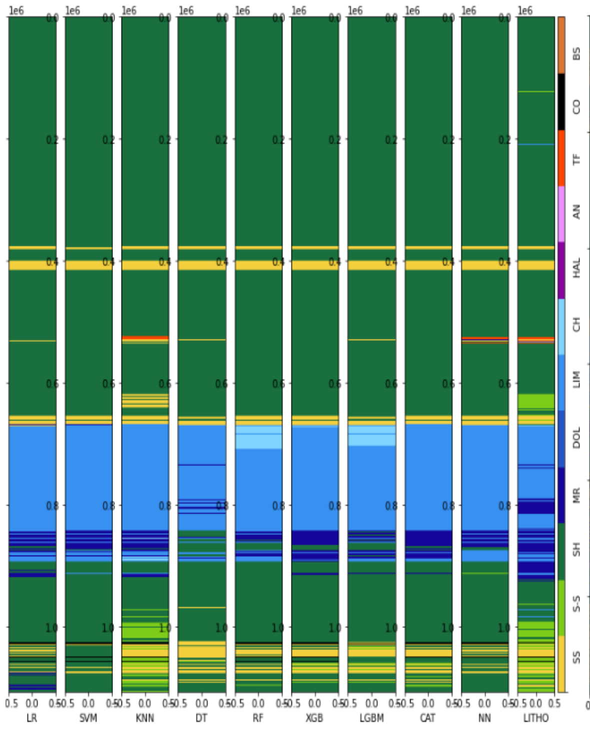
Experimentation.ipynb GitHub location:

https://github.com/JohnMasapantaPozo/Machine-and-Deep-Learning-Applied-to-Geosciences

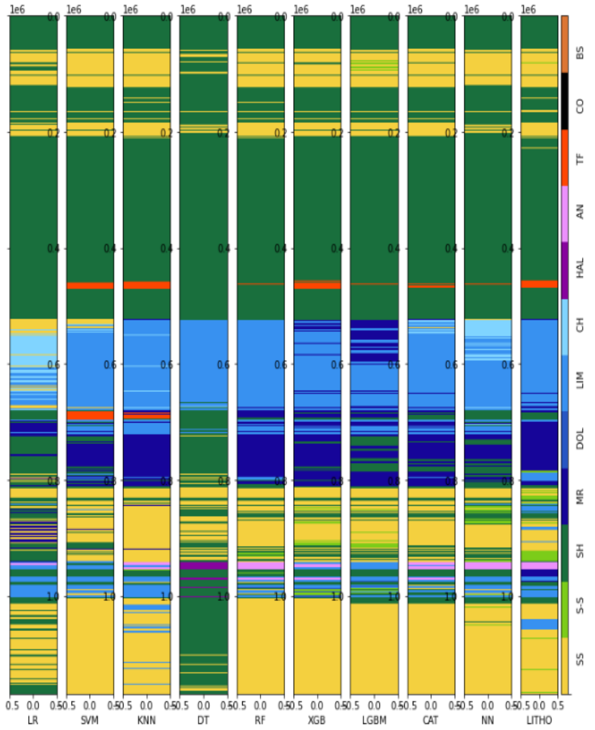
8.6 Appendix F – Lithology prediction results

8.6.1 Hidden test dataset

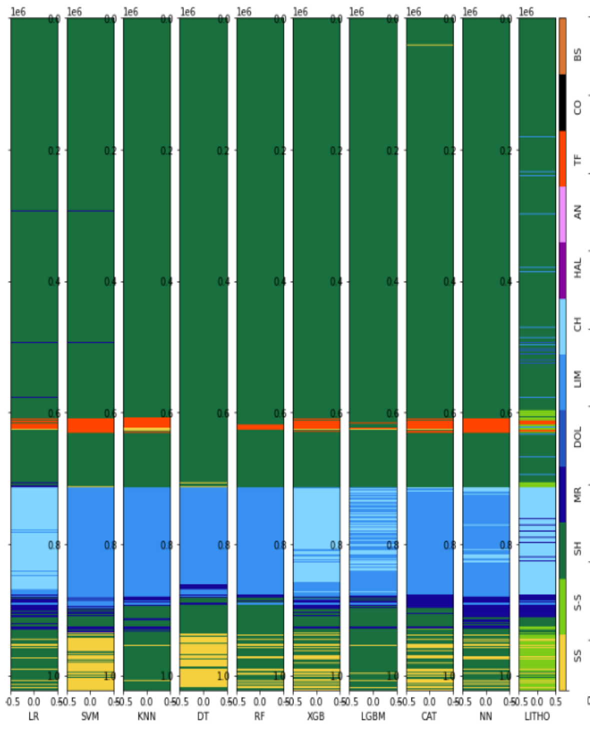
Well 15/9-23



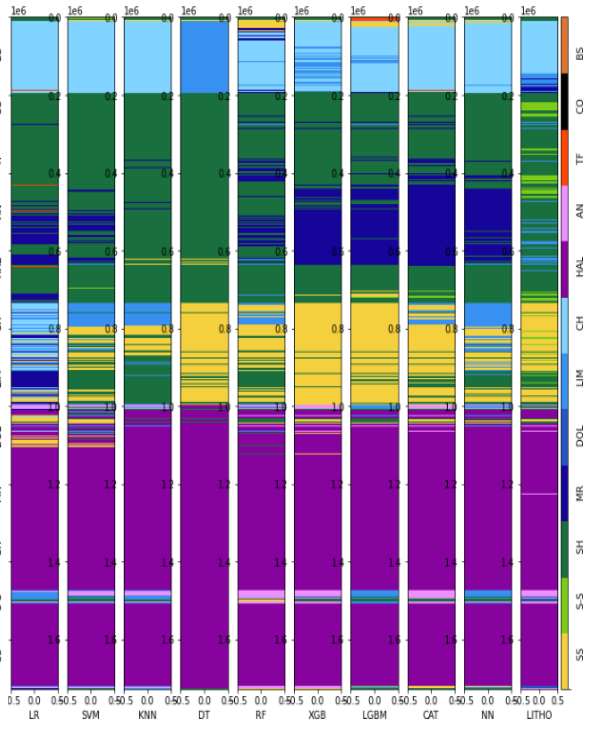
Well 16/2-7



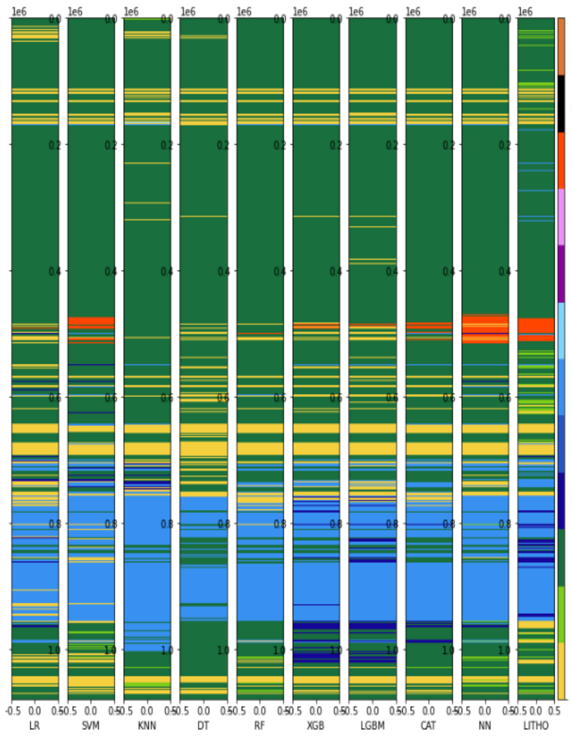
Well 16/7-6



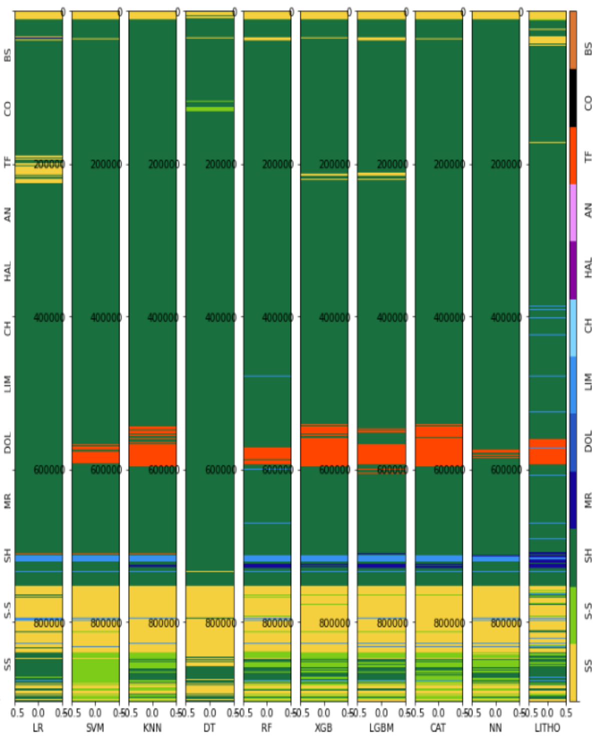
Well 17/4-1



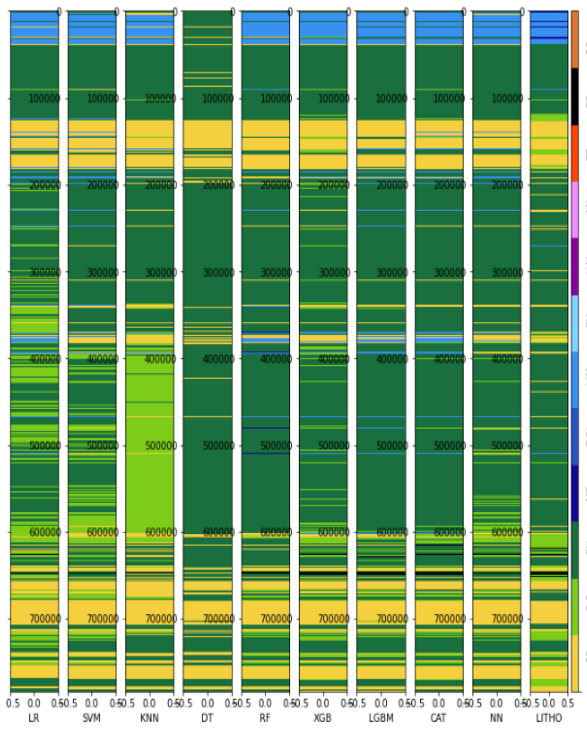
Well 25/10-9



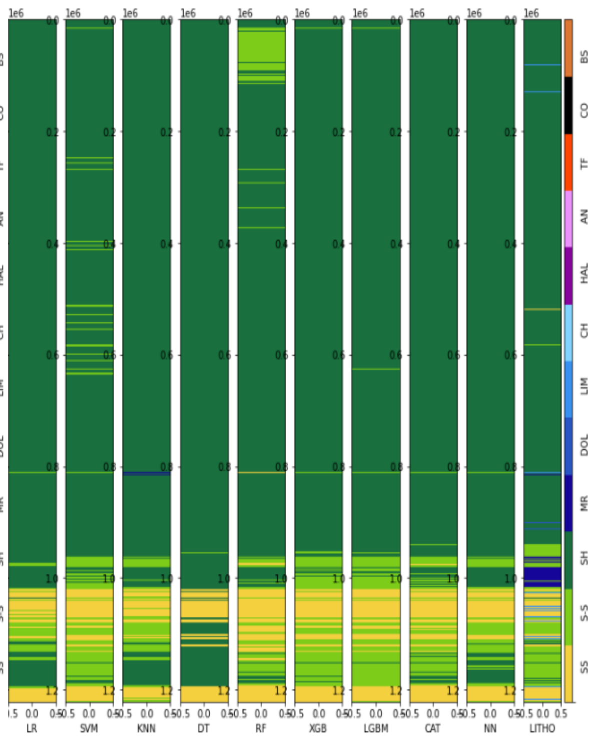
Well 31/2-10

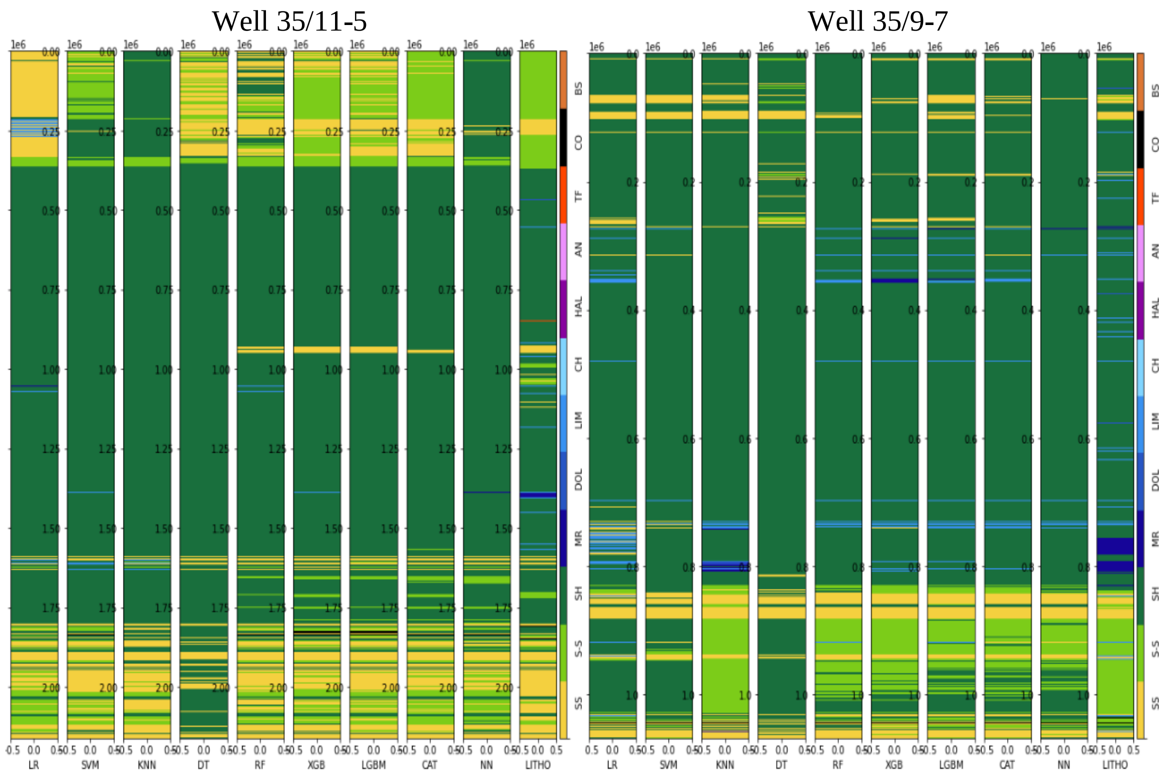


Well 31/2-21-S

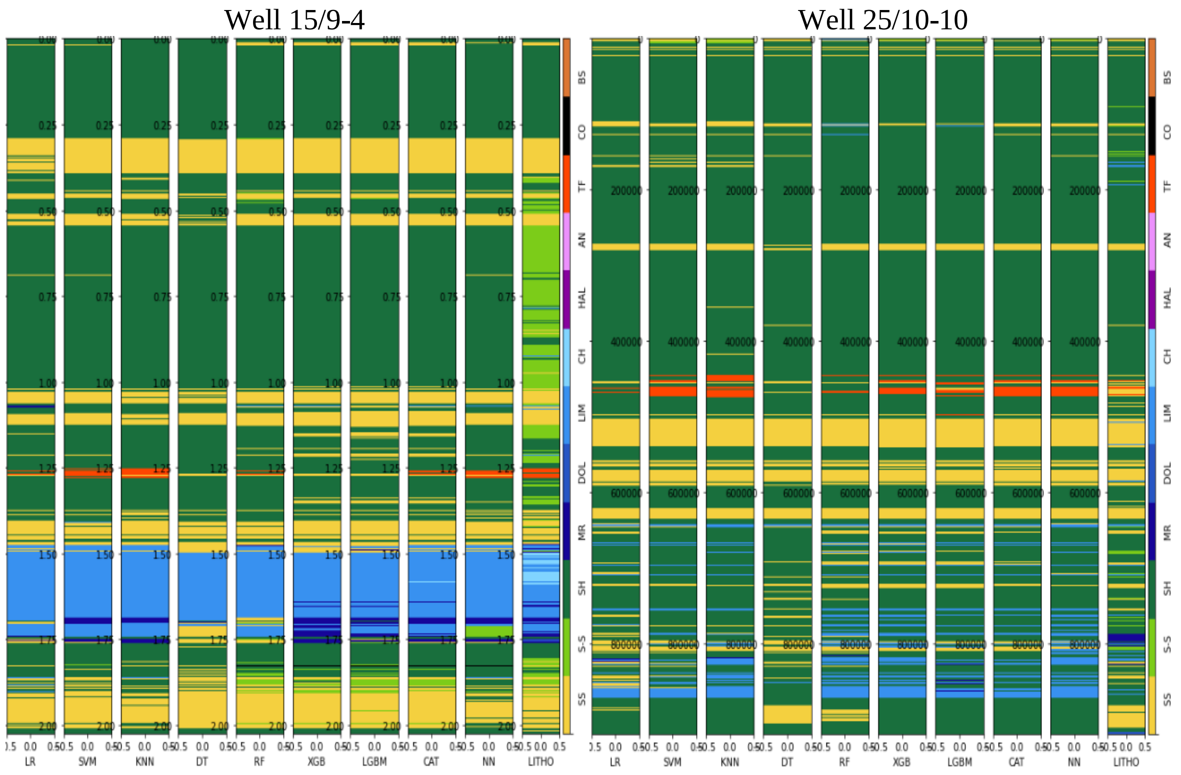


Well 34/3-2S

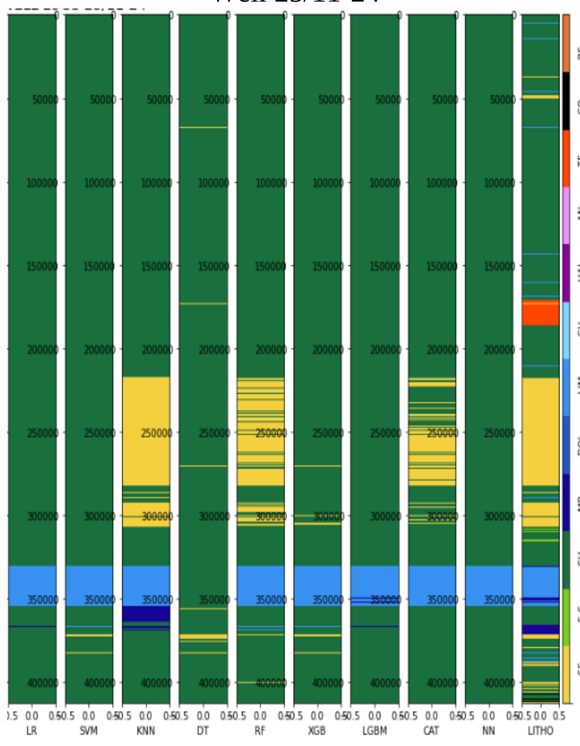




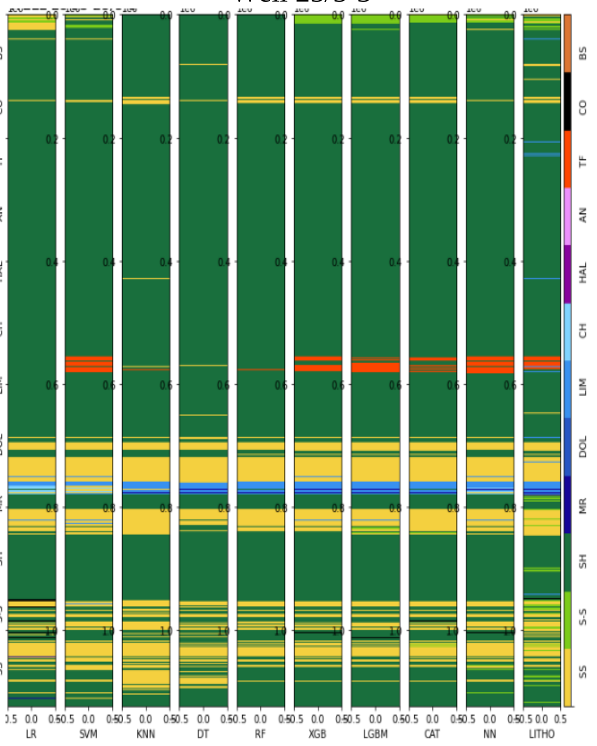
8.6.2 Open test dataset



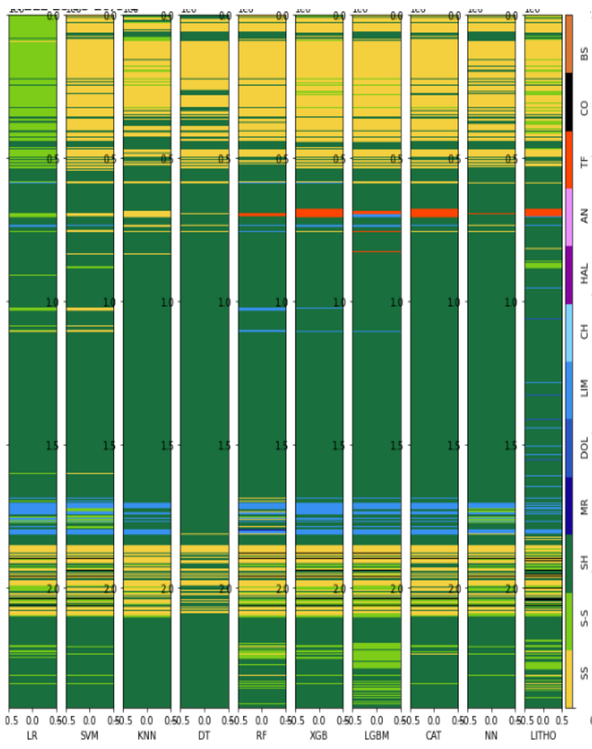
Well 25/11-24



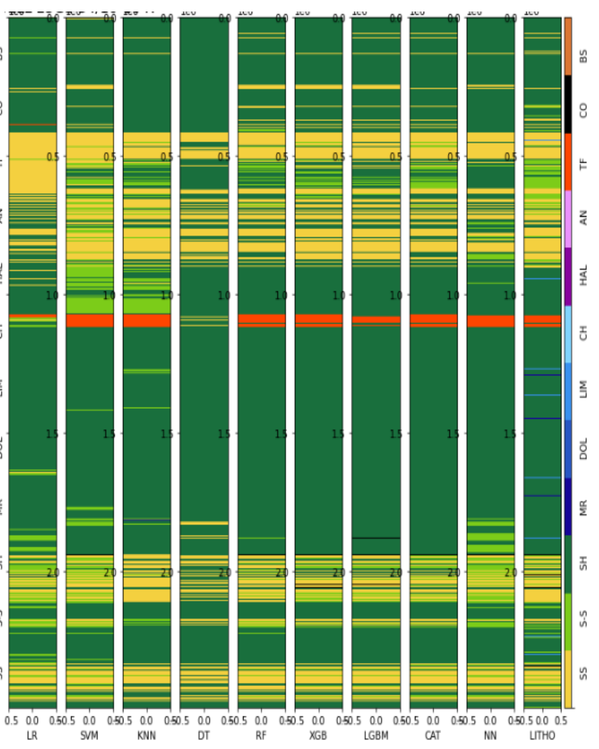
Well 25/5-3



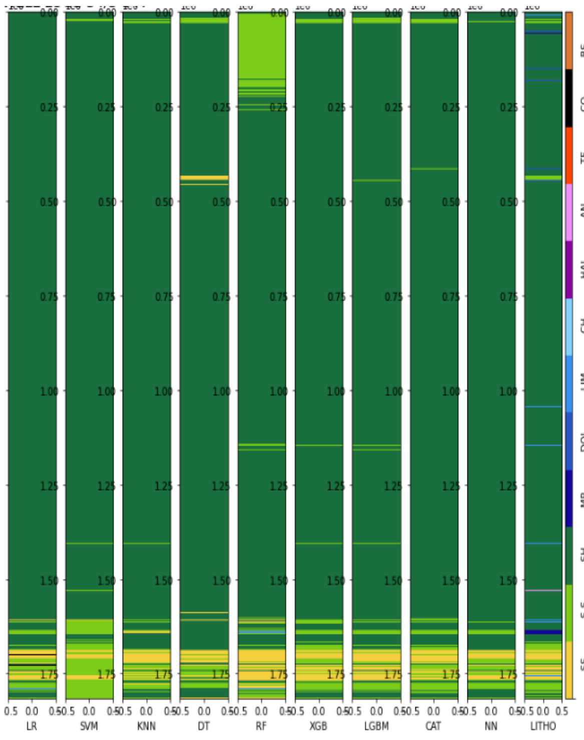
Well 29/3-1



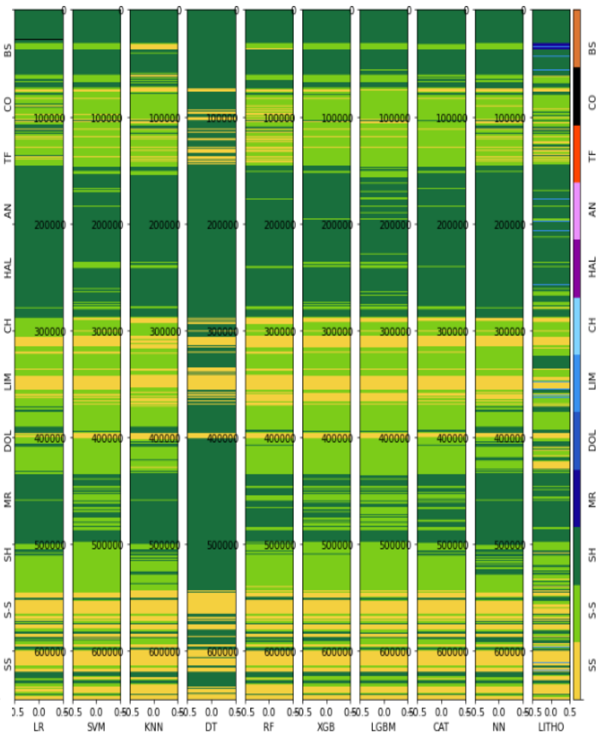
Well 34/10-16R



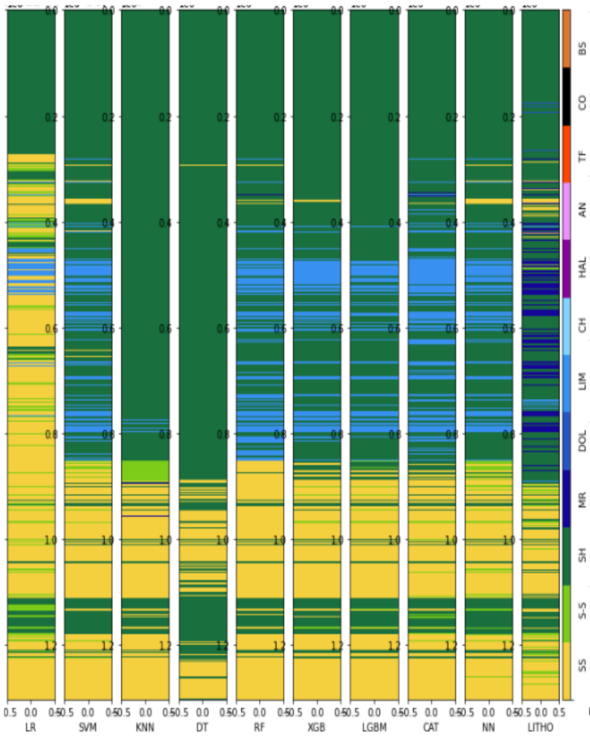
Well 34/3-3A



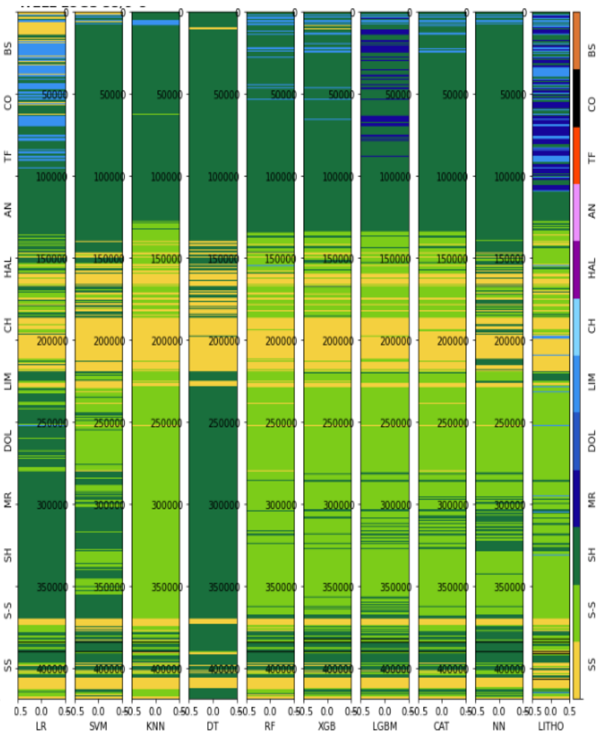
Well 34/6-1S



Well 34/3-3A



Well 34/6-1S



8.7 Appendix G – Open set classification histograms

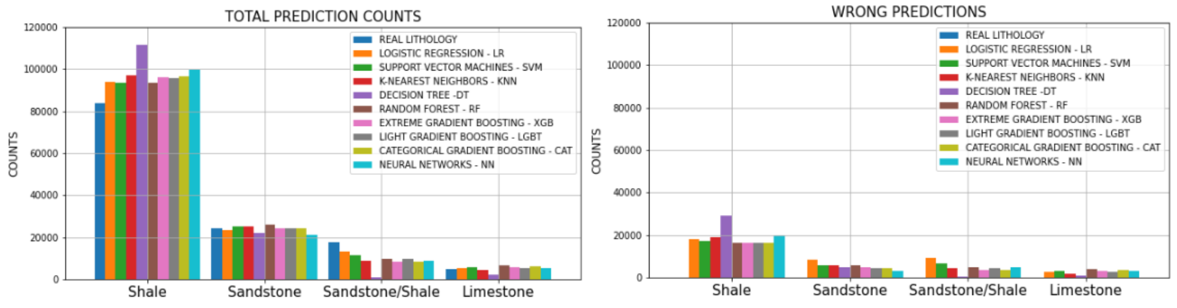


Figure 87 Open test set prediction histograms: Total predictions count (left) and wrong predictions counts (right) for shale, sandstone, shaly-sandstone, and limestone.

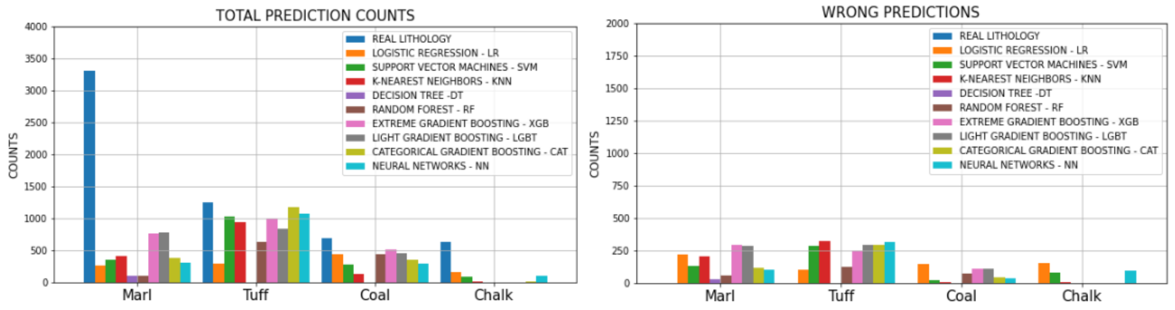


Figure 88 Open test set prediction histograms: Total predictions count (left) and wrong predictions counts (right) for marl, tuff, coal, and chalk.

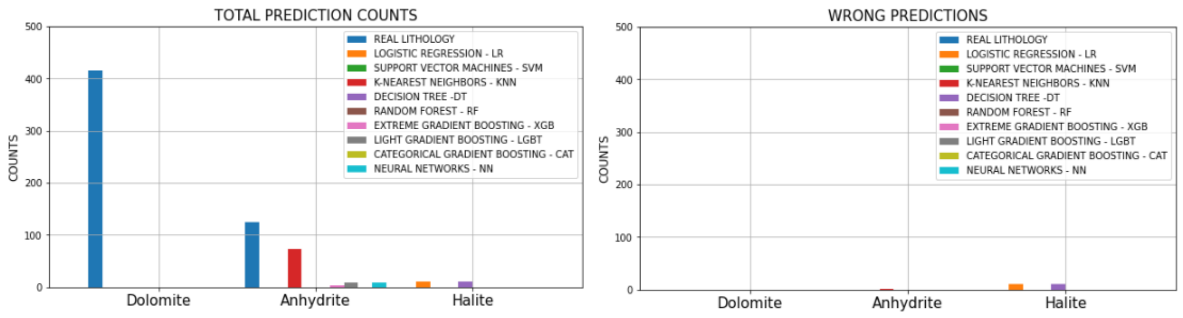
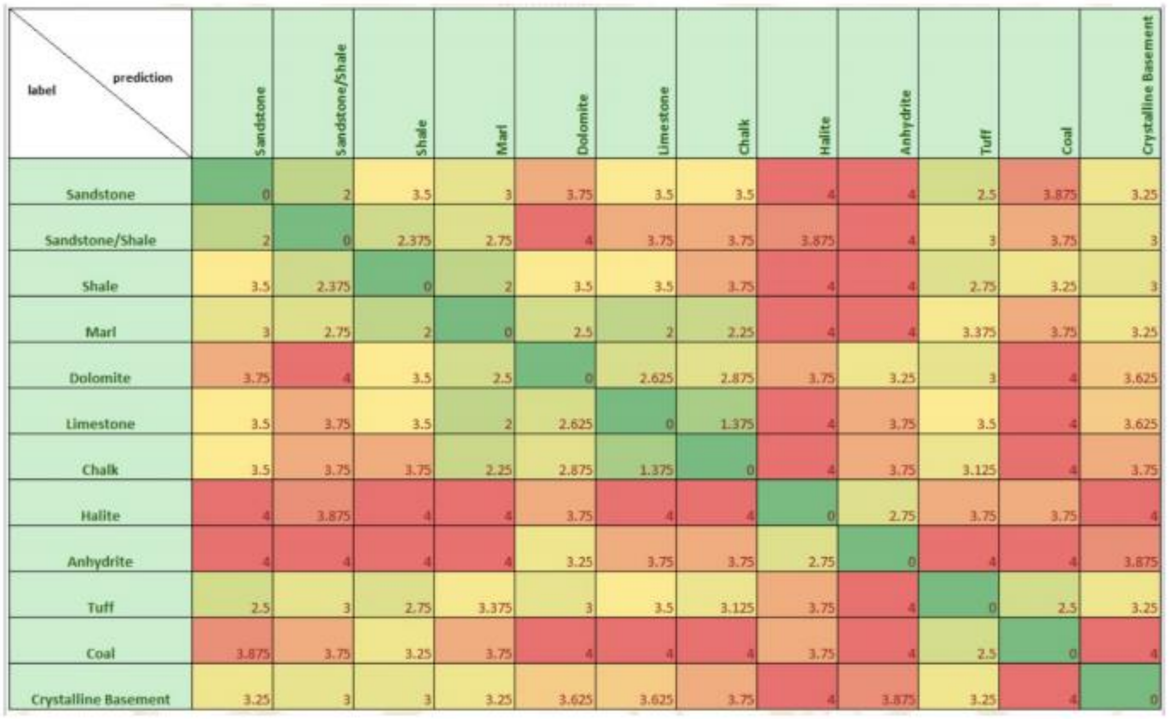


Figure 89 Open test set prediction histograms: Total predictions count (left) and wrong predictions counts (right) for dolomite, anhydrite, and halite.

8.8 Appendix H – FORCE penalty matrix



label \ prediction	Sandstone	Sandstone/Shale	Shale	Marl	Dolomite	Limestone	Chalk	Halite	Anhydrite	Tuff	Coal	Crystalline Basement
Sandstone	0	2	3.5	3	3.75	3.5	3.5	4	4	2.5	3.875	3.25
Sandstone/Shale	2	0	2.375	2.75	4	3.75	3.75	3.875	4	3	3.75	3
Shale	3.5	2.375	0	2	3.5	3.5	3.75	4	4	2.75	3.25	3
Marl	3	2.75	2	0	2.5	2	2.25	4	4	3.375	3.75	3.25
Dolomite	3.75	4	3.5	2.5	0	2.625	2.875	3.75	3.25	3	4	3.625
Limestone	3.5	3.75	3.5	2	2.625	0	1.375	4	3.75	3.5	4	3.625
Chalk	3.5	3.75	3.75	2.25	2.875	1.375	0	4	3.75	3.125	4	3.75
Halite	4	3.875	4	4	3.75	4	4	0	2.75	3.75	3.75	4
Anhydrite	4	4	4	4	3.25	3.75	3.75	2.75	0	4	4	3.875
Tuff	2.5	3	2.75	3.375	3	3.5	3.125	3.75	4	0	2.5	3.25
Coal	3.875	3.75	3.25	3.75	4	4	4	3.75	4	2.5	0	4
Crystalline Basement	3.25	3	3	3.25	3.625	3.625	3.75	4	3.875	3.25	4	0

Figure 90 Appendix H - FORCE penalty matrix NPD, (2021)

8.9 Appendix I – Categorical gradient boosting explanation

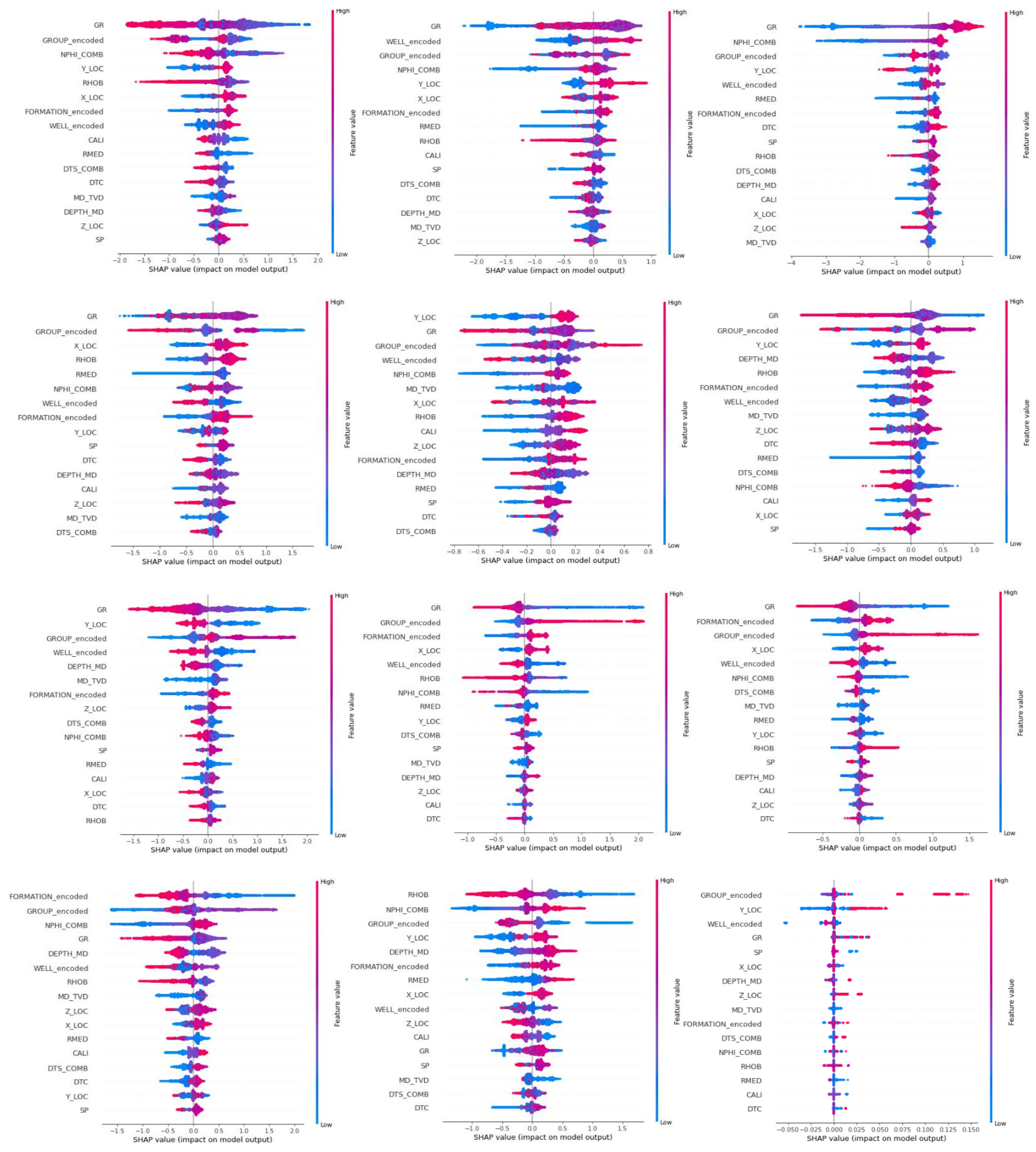


Figure 91 Appendix I - Categorical Boosting Classifier: SHAP values impact on each lithology prediction. Sandstone (0), shale-sandstone (1), shale (2), marl (3), dolomite (4), limestone (5), chalk (6), halite (7), anhydrite (8), tuff (9), coal (10), basement (11). Figures ordered from top left to right down.

8.10 Appendix J – Light gradient boosting explanation

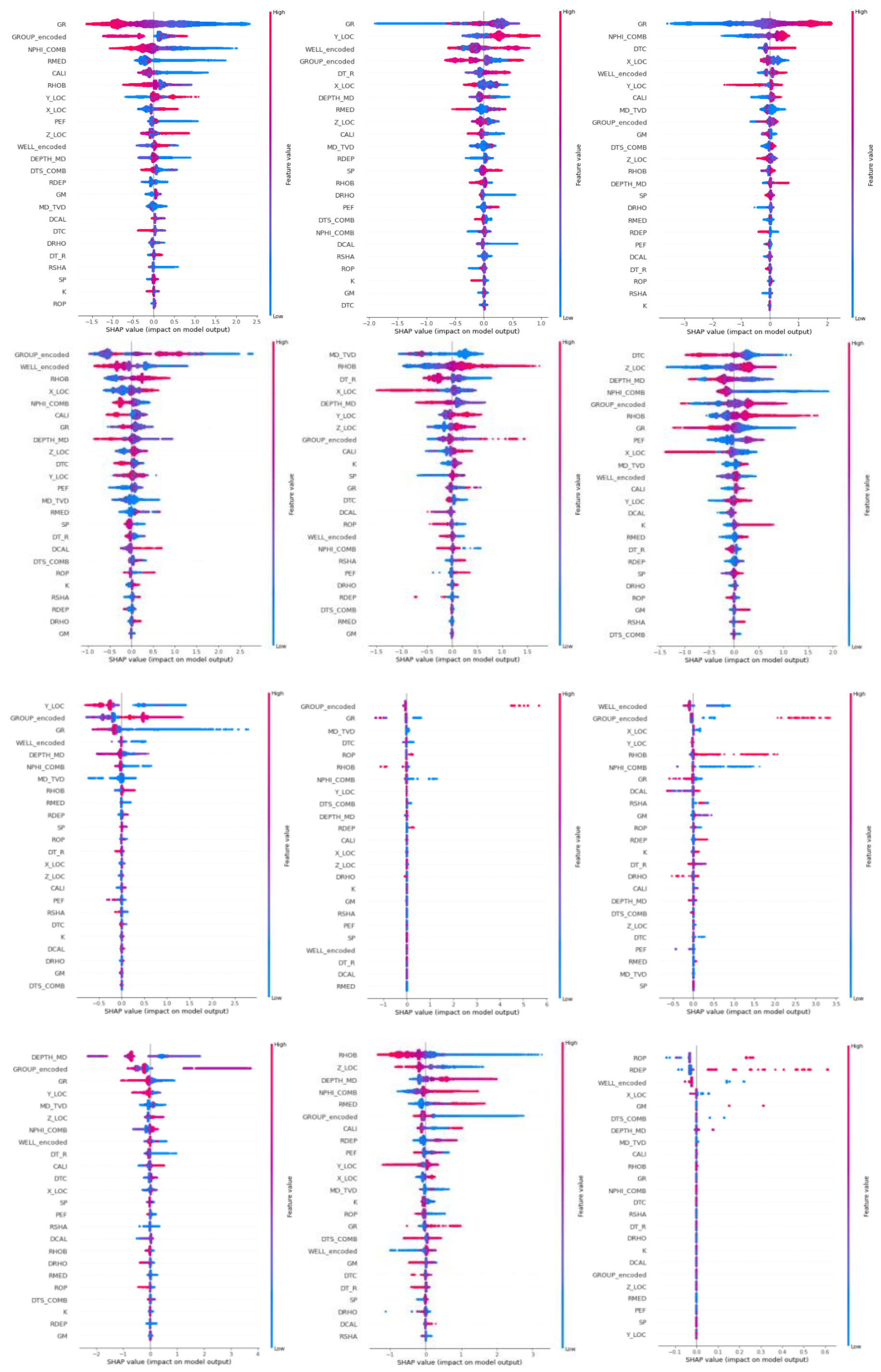


Figure 92 Appendix J - Light Boosting Classifier: SHAP values impact on each lithology prediction. Sandstone (0), shale-sandstone (1), shale (2), marl (3), dolomite (4), limestone (5), chalk (6), halite (7), anhydrite (8), tuff (9), coal (10), basement (11). Figures ordered from top left to right down.

8.11 Appendix K – Extreme gradient boosting explanation

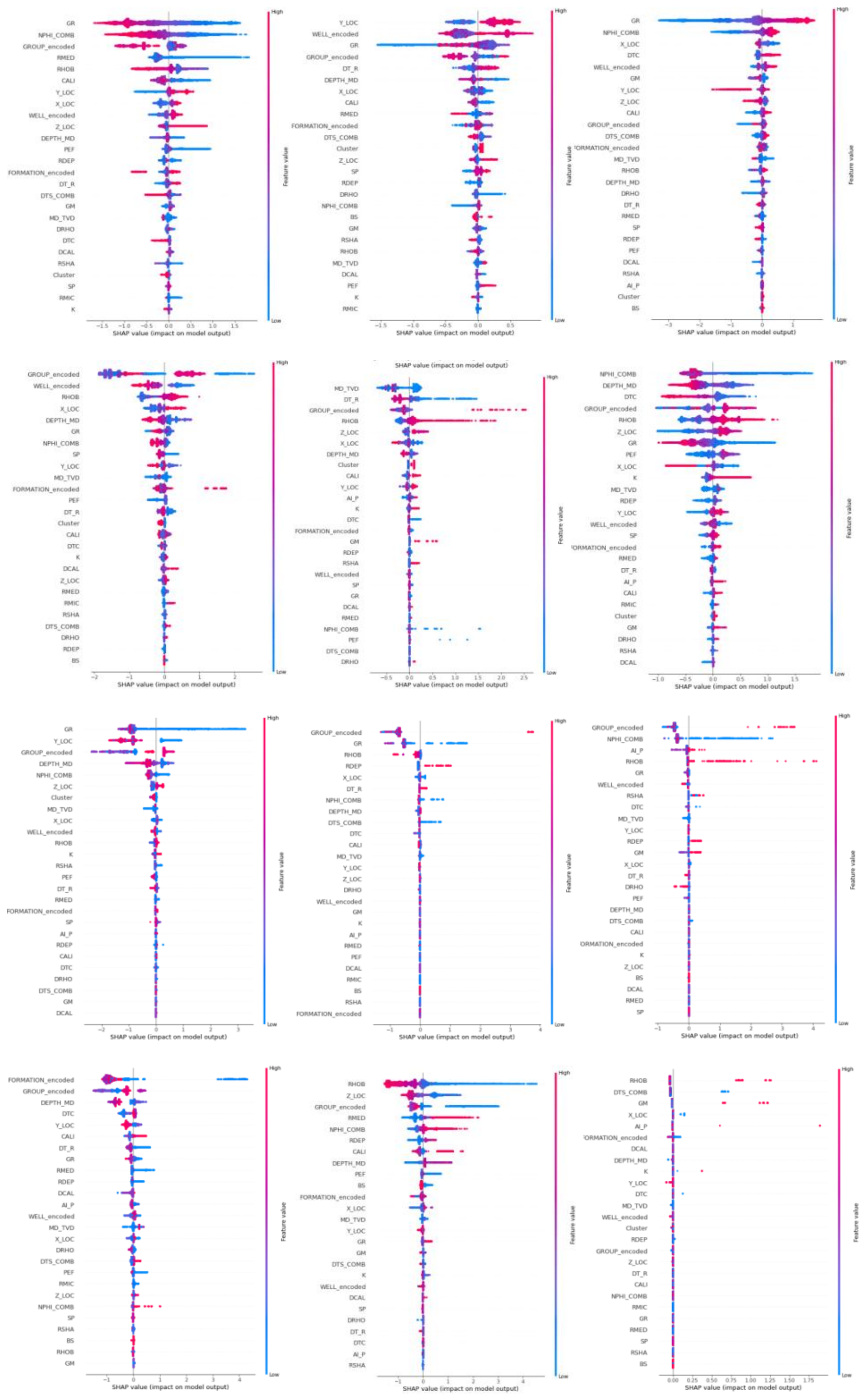


Figure 93 Appendix K - Extreme Boosting Classifier: SHAP values impact on each lithology prediction. Sandstone (0), shale-sandstone (1), shale (2), marl (3), dolomite (4), limestone (5), chalk (6), halite (7), anhydrite (8), tuff (9), coal (10), basement (11). Figures ordered from top left to right down.