Alma Mater Studiorum · Università di Bologna

Scuola di Ingegneria e Architettura

Dipartimento di Informatica - Scienza e Ingegneria (DISI)

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea

in

Artificial Intelligence

Developing a New Approach for Machine Learning Explainability combining Local and Global Model-Agnostic Approaches

Relatore:
Prof. Federico Chesani

Correlatore:
PhD. Giorgio Visani

Candidato:
Vincenzo Maria Stanzione

Anno Accademico: 2020/2021

Sessione III

Alla mia famiglia

Contents

Abstract

The last couple of past decades have seen a new flourishing season for the Artificial Intelligence, in particular for Machine Learning (ML). This is reflected in the great number of fields that are employing ML solutions to overcome a broad spectrum of problems. However, most of the last employed ML models have a black-box behavior. This means that given a certain input, we are not able to understand why one of these models produced a certain output or made a certain decision. Most of the time, we are not interested in knowing what and how the model is thinking, but if we think of a model which makes extremely critical decisions or takes decisions that have a heavy result on people's lives, in these cases explainability is a duty. A great variety of techniques to perform global or local explanations are available. One of the most widespread is Local Interpretable Model-Agnostic Explanations (LIME), which creates a local linear model in the proximity of an input to understand in which way each feature contributes to the final output. However, LIME is not immune from instability problems and sometimes to incoherent predictions. Furthermore, as a local explainability technique, LIME needs to be performed for each different input that we want to explain. In this work, we have been inspired by the LIME approach for linear models to craft a novel technique. In combination with the Model-based Recursive Partitioning (MOB), a brand-new score function to assess the quality of a partition and the usage of Sobol quasi-Montecarlo sampling, we developed a new global model-agnostic explainability technique we called Global-Lime. Global-Lime is capable of giving a global understanding

of the original ML model, through an ensemble of spatially not overlapped hyperplanes, plus a local explanation for a certain output considering only the corresponding linear approximation. The idea is to train the black-box model and then supply along with it its explainable version.

List of Figures

List of Tables

Introduction

The last couple of past decades have seen a new flourishing season for the Artificial Intelligence, in particular for Machine Learning and all its inherent fields. Thanks to a more consistent availability of computational power and in general the spreading of on-premise cloud computing solutions, several companies and researchers have brought their attention to this spectrum of topics. This is reflected in a wide variety of applications and a countless number of fields: from medicine to finance, from news recommendation to online retailing, from autonomous driving to natural language processing applications. In each one of these fields machines proved to be effective and of great support for human beings, also for critical decisions.

However, most of the last employed Machine Learning models have a black-box behavior. This means that given a certain input, we are not able to understand why one of these models produced a certain output or made a certain decision. Neural network-based models in all their declinations are well-known examples of these behaviors. Nevertheless, handling a black-box is not necessarily a bad thing. Most of the time, we are not interested in knowing what and how the model is thinking, but we are focused solely on the output. For example, a news recommendation system will not need a transparent model in the majority of its deployment, even if understanding the model more deeply would be interesting. But if we think of a model which makes extremely critical decisions (e.g. it handles the security system of a nuclear power plant) or takes decisions that have a heavy result on people's lives (e.g. it decides whether you

are eligible for obtaining a loan), in these cases, explainability is a duty. Even before the deployment phase, being capable of understanding why a model is behaving in a certain manner can be useful to predict how it will perform in a real-world scenario.

Several Machine Learning models are not explainable, but a great variety of techniques to perform global or local explanations (in respect to a prediction) are available. One of the most widespread is Local Interpretable Model-Agnostic Explanations (LIME), which creates a local linear model in the proximity of an input to understand in which way each feature contributes to the final output. However, LIME is not immune from instability problems and sometimes to incoherent predictions. Furthermore, as a local explainability technique LIME needs to be performed for each different input that we want to explain, resulting in a higher usage of computing power, mostly if we want to interpret multiple instances.

In this work, we have been inspired by the LIME approach for linear models to craft a novel technique. In combination with the Model-based Recursive Partitioning (MOB), a brand-new score function to assess the quality of a partition and the usage of Sobol quasi-Montecarlo sampling, we developed a new global model-agnostic explainability technique we called Global-Lime. Global-Lime is capable of giving a global understanding of the original Machine Learning model, through an ensemble of spatially not overlapped hyperplanes, plus a local explanation for a certain output considering only the corresponding linear approximation. The idea is to train the black-box model and then supply along with it its explainable version.

The work is structured in 4 chapters:

1. **Explainability**: here a brief overview of Machine Learning and its most important paradigms is given. Then, it is reported what we mean by "Explainability", what are interpretable models and which are the main interpretability approaches. Finally, we focus on LIME, the technique taken as reference, describing its functioning, pro and cons.

2. **Global-Lime**: it is the core chapter of the whole thesis. The theory behind the new explainability technique is described in all its parts, from MOB approach for partitioning to Sobol sampling. Afterwards, the Global-Lime macro-algorithm is described.

3. **Model Architecture**: in this part, we expose how the model has been implemented and which precautions have been taken during the drafting of MOB and Global-Lime. Later, several possible mathematical and code optimizations are discussed, with their advantages and drawbacks.

4. **Testing and Results**: it is the final chapter in which we report the results obtained during the testing phase. We performed tests on continuous and discontinuous ad-hoc functions, to assess the mathematical rigour behind our thesis. Finally, we tested Global-Lime on two real datasets, one numerical and one containing also categorical variables, witnessing a coherent and sufficiently robust behaviour.

Chapter 1

Explainability

Explainability is defined as: "the usage of methods and models that make the behavior and predictions of machine learning systems clear and understandable to humans" [1]. Explainability is one of the hottest topics in AI (Artificial Intelligence) in the last few years [2], due to several reasons and necessities that are born thanks to the spreading of these technologies, especially ML(Machine Learning) and Deep Learning, in a wide range of fields.

In fact, despite the widespread adoption of AI solutions, ML-based models remain mostly black boxes. Given a certain input to the model, we obtain the desired output, actually without any knowledge of its internal workings. Despite the power of the AI-based approaches, black-box behavior is inconvenient and disadvantageous if we want to know how the model "thinks" or why it has predicted a class instead of another. Recently, it is something that also EU decided to regulate [3] because of the importance of AI trust in highly critical situations. Furthermore, understanding the reasons behind predictions is quite important in assessing also its trust, which is fundamental if one plans to take actions based on the predictions of a model or simply wants to have insights into the model dynamics.

In this chapter, we will briefly introduce what are ML and Deep Learning, what is explainability and why is important, and then we will discuss the most

important explainability techniques with their pros and cons.

1.1 Machine Learning

Usually, two definitions of Machine Learning are offered. Arthur Samuel described it as [4]: “the field of study that gives computers the ability to learn without being explicitly programmed.” Tom Mitchell provides a more modern and formal definition [5]: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." In detail, it is seen as a part of artificial intelligence, whose aim is to design and exploit algorithms and models that improve automatically through experience and data, in order to make predictions and actions, without being explicitly programmed to do so.

Nowadays ML is used in a wide variety of applications such as computer vision, spam filtering, classification of items, speech recognition, market prediction, weather prediction, medical diagnosis, etc. The core of all ML is to be capable of generalizing from experience, in particular, the ability to perform consistently also on new unseen examples after a training phase [6]. ML approaches are usually divided into 4 categories, depending on the nature of the feedback mechanism for the learning process.

1.1.1 Supervised Learning

Supervised learning is the process of learning a mathematical model that tries to map an input to an output based on the submitted training set of labeled data, i.e. containing (input, output) tuples.
The inferred function is used to predict output values on unseen data. An example of usage could be: we want to predict a person’s height using his weight, age, and gender. The training data will be composed of tuples having people’s weight, age, gender info along with their real heights to discover the

relationship between height and the other features in the dataset.
Furthermore, we can identify two subproblems under the supervised spectrum (Figure 1.1.1):

- **classification**: the model learns to label an observation based on its variable values. For example, if an email is spam or not analyzing the occurring keywords.
- **regression**: the model tries to estimate the relationship between the independent variables (inputs) and the dependent variables (outputs), using a certain mathematical model, then forecasts new outcomes. It is important to understand that there are no fixed labels to assign in this case, but more a computed value. For example, given the size in squared meters m^2, the zone, and the floor of a flat, we want to estimate the hypothetical selling prize and renting revenue.

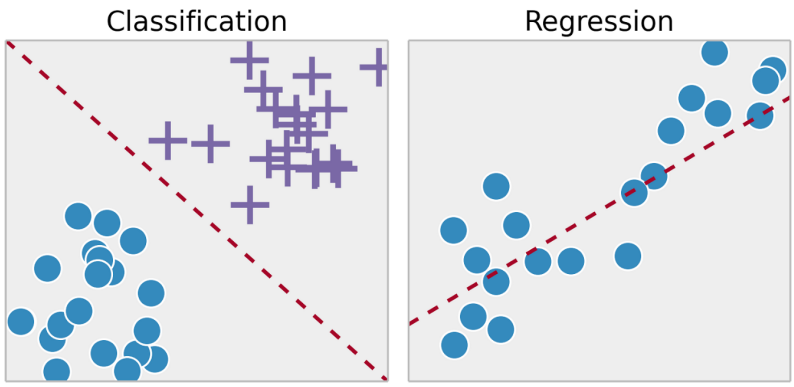


Figure 1.1.1: Classification and Regression [7].

1.1.2 Semi-supervised Learning

Similarly to supervised learning, semi-supervised learning, also known as active learning, is a technique that involves a small number of labeled examples but also a large number of unlabeled examples. The algorithm can interactively query

the user to label chosen data points. There are scenarios in which unlabeled data is abundant but manual labeling is expensive, so the learner samples some significant examples to submit to the user [8].

The evidence of an effective semi-supervised learning algorithm is that it can achieve better performance than a supervised learning algorithm fit only on the labeled training examples of the dataset.

Semi-supervised learning may be used or may contrast inductive and transductive learning[1]. Inductive learning refers to a learning algorithm that learns from labeled training data and generalizes to new data. Transductive learning refers to learning from labeled training data and generalizing it to available unlabeled (training) data. Both types of learning tasks may be performed by a semi-supervised approach.

1.1.3 Unsupervised Learning

Unsupervised learning is a type of machine learning in which the algorithm is not provided with any pre-assigned labels or scores for the training data, whose aim is to look for previously undetected patterns with minimal human supervision. So it will self-discover any naturally occurring patterns in the given data, in contrast to what is done in supervised learning, where a human gives a list of assignable labels [9].
The main unsupervised approaches are:

- **clustering**: is a grouping process in which the members of a group (i.e. a cluster) are more similar to each other than the members of the other clusters (Figure 1.1.2).
- **dimensionality reduction**: the process of selecting T features in a dataset, removing for example highly correlated variables, or the creation of new features that are smaller in number in respect to the original ones. The new features summarize the information carried by the previous ones.

[1] We will not cover in details inductive and transductive learning in this introduction to ML because is out of our scope.

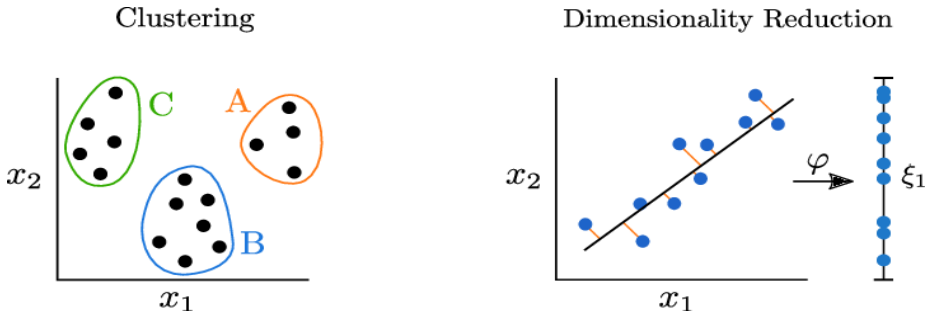


Figure 1.1.2: Unsupervised Learning [10].

1.1.4 Reinforcement Learning

Reinforcement learning is an area of machine learning concerned with how agents ought to take actions in an environment to maximize the notion of cumulative reward. A reinforcement learning ecosystem is composed of (Figure 1.1.3):

- **agent**: the entity which performs the assigned tasks.
- **environment**: the logical space in which the agent completes its tasks.
- **action**: a move the agent can do which results in a change of the status of the environment and a returned reward.
- **reward**: a negative or positive remuneration given by the environment, depending on its status and the previous actions.

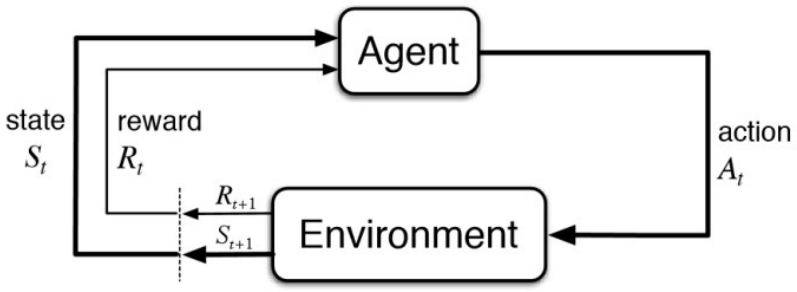


Figure 1.1.3: Reinforcement Learning.

1.2 Deep Learning

Deep Learning is a subset of ML in which techniques that involve the usage of neural networks and representation learning are leveraged, to perform ML tasks[2]. Recently, thanks to the increasing computing power available, Deep Learning approaches led to outstanding results in a variety of fields. For example, Alpha Go [11] successfully mastered the board game Go, reaching superhuman performances and being capable of defeating the world champion[3].

Artificial Neural Networks (ANNs) are the core of Deep Learning. ANNs are computing structures inspired by the functioning of the brain: a collection of nodes called neurons are connected using weighted connections, to produce an output given a certain input.
A Deep Neural Network (DNN) is a particular ANN composed of multiple layers (input, hidden, output) of neurons between input and output (Figure 1.2.1). The advantage of DNN is the ability to create a non-linear relationship between input and output that can theoretically approximate any mapping.

However, Deep Learning models behave such as black boxes. Because of that, Deep Learning does not thrive as the principal solution in fields which require critical decisions such as financial and medical fields, nevertheless the adoption is steadily growing.

1.3 Overview on Explainability of ML

As we stated in the previous paragraphs, despite the widespread adoption of ML models, most of them are black boxes. A naive consideration would be why explainable models are needed if we are just interested in the output. Several times we are not interested in the internal mechanism inside the box, e.g. the

[2]The most successful tasks are: computer vision, speech recognition, natural language processing, machine translation, bioinformatics, drug design, medical image analysis, board games and so on.

[3]It is important to keep in mind that before 2015, the best Go programs only managed to reach amateur levels and Go was considered unapproachable due to its intrinsic complexity.

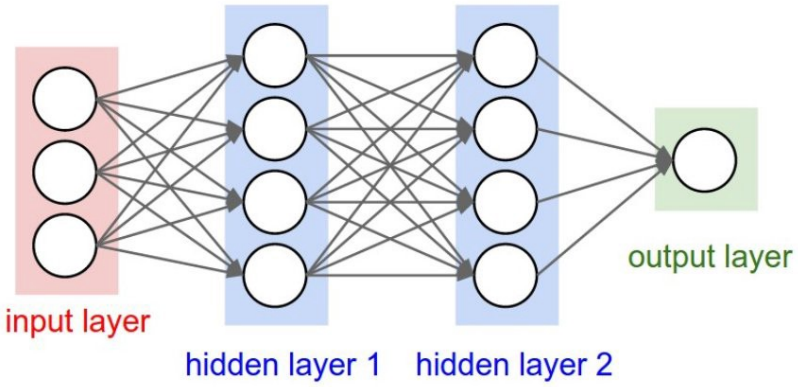


Figure 1.2.1: Deep Neural Network.

model has great metrics scores yet. The answer is that from explainability derives trust, so understanding the reasons behind predictions is fundamental in assessing trust on a model [12]. Trust in a model is essential if one plans to take action based on a prediction or when choosing whether to deploy a new model. If a model is not trusted, the users will not use or rely on it, because it is difficult to understand on what grounds a decision has been taken. Furthermore, as the Right to Explanation [13] and the GDPR [14] state, each individual affected by The algorithm's decisions have the right to know the model's rationale.
We can define trust from two different points of view:

- **trusting a prediction**: a user trusts an individual prediction sufficiently to take some action based on it. It is a vital feature for decision making in critical situations like medical diagnosis or bomb detection. The decision cannot be made upon "blind" faith in the model.

- **trusting a model**: whether the user trusts a whole model to behave in reasonable ways if deployed in the real world, which can be significantly different from the depicted one in the training set.

1.3.1 Methods for Interpretability

ML is often difficult to interpret due to the complexity of the mathematical structures inside the model. However, in the ML domain, a group of intrinsically explainable models exists. In order to achieve explainability one could exploit one of these transparent models, with simple formulas to understand , or to use post-hoc techniques to convey the knowledge of a black box model in an understandable representation, such as a simpler surrogate interpretable model.
More strictly, we can define different criteria for classifying interpretability methods [15]:

- **intrinsic interpretability**: selecting and training a machine learning model that is intrinsically interpretable due to its simple structure (e.g. Decision Tree, Linear Regression).
- **post-hoc or post-model methods**: application of interpretability methods after the training of the model. They provide insight into what the model has learned, without changing the underlying model.
- **pre-model methods**: independent of the model. They can be used before the selection of the model (e.g PCA, t-SNE, clustering).

Our goal is to explain a prediction or the whole model, which means presenting textual or visual artifacts that provide a qualitative understanding of the relationship between the instance's components (e.g. words in a text, patches in an image) and the model's prediction, so human-understandable. Usually, interpretability methods can also be classified by the outcome of the prediction model. A not exhaustive list could be:

- **Feature summary statistics**: refer to a summary of each feature statistic that affects the model predictions.
- **Feature summary visualization**: methods that can only be visualized and could not be meaningfully presented in the form of a table.

- **Data point interpretability**: methods that return data points to make the model interpretable.
- **Intrinsically interpretable**: interpretable by internal model parameters of feature summary statistics.

1.3.2 Local and Global Interpretability

In general, if a method can explain a single prediction, we refer to those methods as local, whilst if it explains the entire model behavior as global [1]. Local interpretability can be achieved if we design a model which explains why a specific decision has been made, then explanations are valid only in a tight neighborhood of the selected point.

On the other hand, globally interpretable models offer transparency about what is going on inside a model on an abstract level [16]. Global methods describe how features affect the prediction on average[4].

It is important to consider the fact that through a selected group of local explanations, it is possible to obtain a global view of the overall behavior of a model. So achieving a sort of global explainability, leveraging local explanations.

1.3.3 Model-agnostic and Model-specific Interpretability

Model-specific interpretation methods are limited to specific models and derive explanations by examining internal model parameters [16]. However, it is very annoying in a situation in which we are testing various models and we would prefer to have a standard criterion to interpret them In this context, we understand that separating the explanations from the ML model leads to some important advantages. The major one is flexibility. Model-agnostic interpretation methods are methods that can be applied to any type of model, without changing its structure. With model-agnostic techniques, developers are free to

[4]However, several of these methods (e.g. Global-Lime) can be used for local and global explanation both)

use any machine learning model they prefer, because potentially the interpretation methods can be applied to any model. Interpretation and model become independent.

By model flexibility we mean that the interpretation technique is valid for every ML model [12].

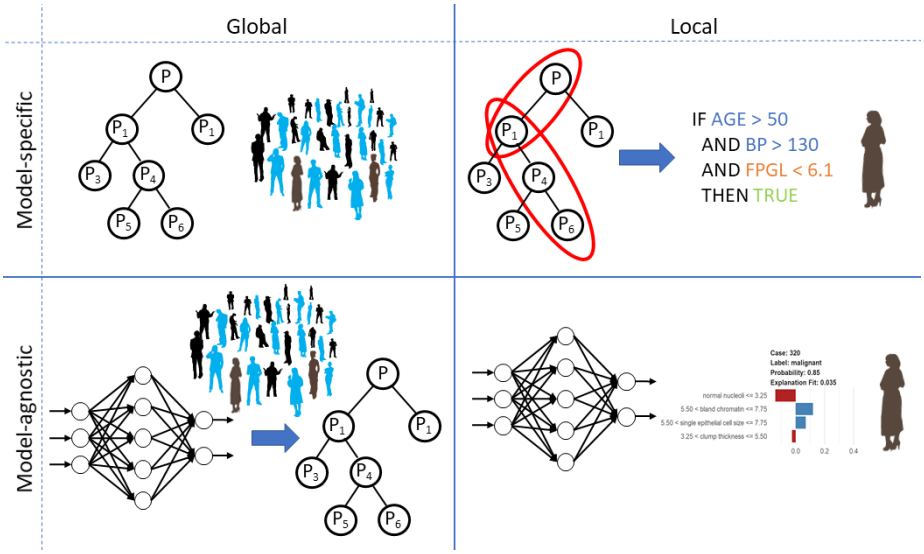


Figure 1.3.1: Explainability techniques summary [15].

1.4 Interpretable Models

As we discussed in section 1.3.1, the most straightforward approach to guarantee interpretability is to use a subset of interpretable algorithms. For completeness, in this section, a brief overview of these methods will be given[5].

1.4.1 Linear Regression

By definition, a linear regression is a weighted sum of the independent variables to predict a dependent variable, this relationship between input and output

[5]There are several interpretable techniques not reported, we will cover only the ones that will be of interest for the next sections.

makes the interpretation easy to understand.

$$y = \beta_0 + \beta_1 x_1 + ... + \beta_n x_n + \epsilon \tag{1.4.1}$$

Or similarly:

$$y = \boldsymbol{X}\beta + \epsilon \tag{1.4.2}$$

The predicted output is a weighted sum of n features, plus an intercept (β_0) and the error ϵ which follows a Gaussian distribution. Hence, we can think the β weights as the importance of a certain variable to compute y. We estimate the optimal weights by minimizing a certain objective function e.g. mean squared errors, but various methods can be used.

1.4.2 Logistic Model and Probit Model

Linear models are easy to use and simple to interpret, but the predictions of a linear model are not bounded, potentially they span through all the real numbers. For example, this could be tedious if one would have to model a probability (from 0 to 1). To overcome this limit, models like Probit (equation 1.4.4) and Logistic (equation 1.4.3) are used, where $\Phi(\cdot)$ is the Cumulative Distribution Function of a standard Gaussian N(0, 1) (Figure 1.4.1).

Both of them transform the probability into a new real variable, using a bijective function. In this way, we can model the relation between the variables in a non-linear way, with an increase in the representation capacity [17].

$$y = \frac{e^{\mathbf{X}^T\beta}}{1 + e^{\mathbf{X}^T\beta}} \tag{1.4.3}$$

$$y = \Phi(\mathbf{X}^T\beta) \tag{1.4.4}$$

An additional perk of Logistic Regression, when compared to Probit, is its interpretability: the parameters derived from the best curve's estimation, can be

regarded as odds ratio, more precisely $\frac{P(Y=1|X=x)}{P(Y=0|X=0)}$. Some disadvantages of logistic regressions are:

- the interpretation is harder than a linear regression.
- it suffers from complete separation: if a feature perfectly separates two classes, the logistic regression weight for that feature will never converge because the optimal weight would be infinite [18].

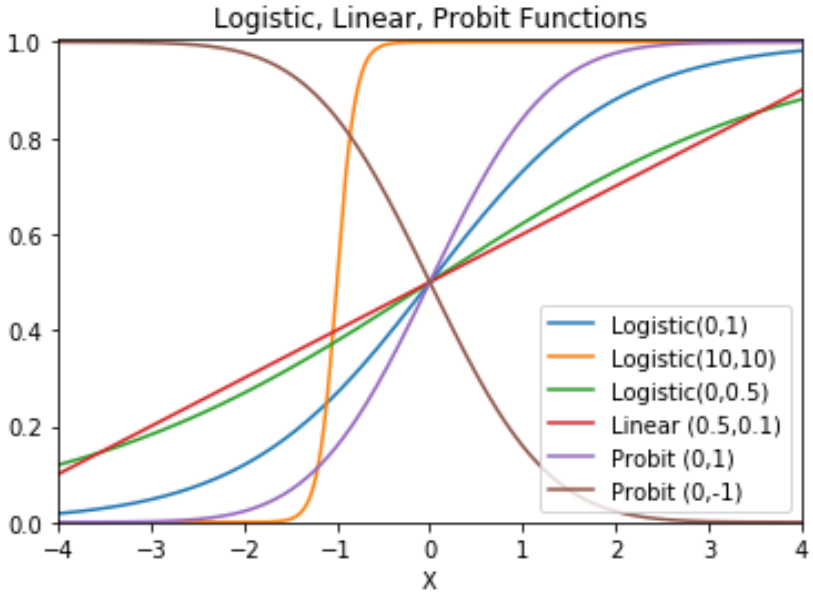


Figure 1.4.1: Logistic model and Probit model [17], on the y-axis the probability $y = P(Y = 1|X = x)$.

1.4.3 Generalized Linear Models (GLM) and Generalized Additive Models (GAM)

Understanding a prediction made by a linear regression model is simple due to its intrinsic clarity: a prediction is modeled as a weighted sum of the features, then we know which are the dominant features. However, simplicity is also the

biggest weakness of linear models. In real-world data, most of the time the features might interact, the outcome might not be linear or might not have a Gaussian distribution. The solution for outcomes that do not follow a Gaussian distribution could be GLM, whilst for a not linear relationship between the features could be GAM.

Linear regression models can be extended to support other types of outcomes (non-gaussian outcomes) through GLM. GLM mathematically links the weighted sum of the features with the mean value of the conditional distribution assumed for Y using a link function g, which can be chosen depending on the type of outcome [1]. E_Y is a probability distribution from the exponential family[6].

$$g(E_Y(y|x)) = \beta_0 + \beta_1 x_1 + ... + \beta_p x_p \tag{1.4.5}$$

In this way, we can consider a linear model as a special case of GLM, in which g is the identity function. The chosen distribution together with the link function determines how to interpret the weights of the model. First, we need to invert g and then consider the contribution of E_Y.

$$E_Y(y|x) = g^{-1}(\beta_0 + \beta_1 x_1 + ... + \beta_p x_p) \tag{1.4.6}$$

Obviously, this leads to a harder interpretation of the β vector.

Instead, GAM models are used to model a non linear relationship between features (Figure 1.4.2). For example, the 100m lap-time and the height/weight of a sprinter, intuitively, are not linearly linked. More weight in proportion to height could mean more muscles and then better speed, but at a certain point adding weight becomes disadvantageous and can worsen performance. Also being tall enough is an advantage, but being too tall is a problem if we think about the acceleration phase. If the model was linear, then it would have meant that increasing of 100kg the weight of an athlete is good for the lap time (obviously very hard to think). One could think to bypass the problem

[6]We will not cover in details this topic. It is sufficient to know that is a set of distributions that can be written with the same parametrized formula, including and exponent, the mean and variance of the distribution [19].

considering the height/weight ratio, but the problems are only postponed. In fact, if the relationship between input and output is not linear, could be harsh to rearrange the features to obtain the desired effect.

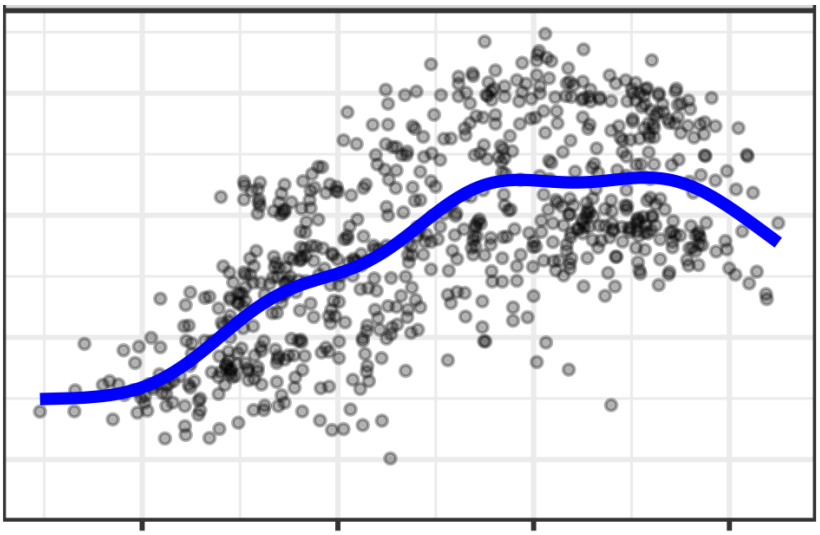


Figure 1.4.2: Example of a GAM on a non-linear output.

GAM relax the linear regression assumption that the output is $\sum_{i=0}^{d} \beta_i x_i + \beta_0$ and assumes the output is a sum of arbitrary functions $f_j(x_j)$. g and E_Y are what we defined for the GLM case.

$$g(E_Y(y|x)) = \beta_0 + f_1(x_1) + f_2(x_2) + ... + f_p(x_p) \tag{1.4.7}$$

The output is still a sum of features effects but with the possibility of a non-linear relationship between between each feature and the target variable Y[7].

1.4.4 Decision Tree

Linear regression and logistic regression models fail in situations where the relationship between features and outcome is nonlinear or where features interact

[7]Most modifications of the linear model make the model less interpretable and inherently more complex. Although, the power of GLM and GAM, our preference will go to linear models.

with each other [1].

Decision trees (Figure 1.4.3) are capable of overcoming this problem. The dataset is split multiple times, according to certain criteria and tests on the attributes. At the end of this process, we obtain a group of leaves, which are grouped elements with the same features. To predict the outcome in each leaf node, the average outcome of the training data in this node is used.

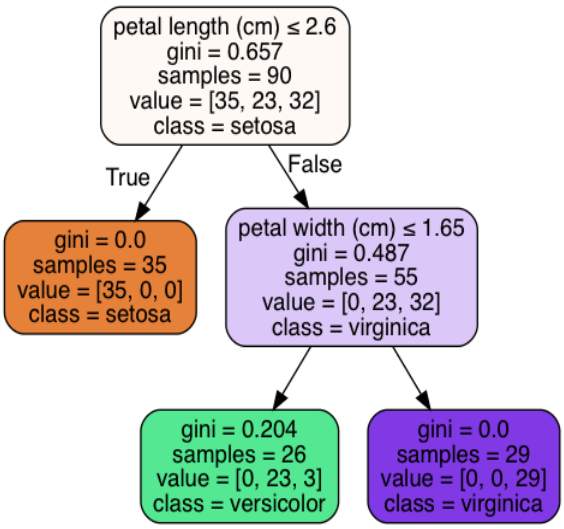


Figure 1.4.3: Example of a decision tree on the iris dataset.

The interpretation of these trees is simple. Starting from the root node, we traverse the tree, following the test that respects the characteristics of the selected point. Once reached the leaf node, the output is predicted. In this way, we are conscious in every moment of why the model has predicted a class because the chain of tests followed is known.

However, decision trees can create over-complex trees that do not generalize the data well (overfitting). Furthermore, decision trees can be unstable because small variations in the data might result in a completely different tree being generated. This leads also to the consideration that predictions are neither smooth nor continuous with respect to the X variables, because of the constant approximation process of subsetting the dataset [20].

1.5 Local Model-Agnostic Techniques

As we said, model agnostic techniques guarantee flexibility. In this section we will discuss one of the major techniques for local explainability. In particular, we will focus on Local interpretable model-agnostic explanations as the most inspiring for our new technique described in chapter 2. There are several techniques such as SHAP values[8], but we will maintain the focus on LIME, which is the technique we referred as the base for our new approach.

1.5.1 Local interpretable Model-agnostic Explanations (LIME)

LIME is an explanation technique that explains the predictions of any classifier in an interpretable and faithful manner, by learning an interpretable model locally i.e. around the prediction [12]. An explanation needs to use a human-friendly representation to be useful, so in most cases we will retain only the most important variables in a locality of the individual to interpret, reducing the dimensionality of the explanation .

We can define $g \in G$ a model in the class of interpretable models, which has to approximate the original model f. We define $\Omega(g)$ as a measure of the complexity of the explanation. Given x and z points of a model f to be explained, we define $\pi_x(z)$ as a proximity measure between z to x, so a locality of z from x. Let $\mathcal{L}(f, g, \pi_x)$ be a measure of the error of the approximation g in respect to f. Most of the time a weighted squared loss is used as $\mathcal{L}$. The idea behind LIME is to minimize $\mathcal{L}(f, g, \pi_x)$ while keeping $\Omega(g)$ low enough. Than find the optimal g:

$$\xi(x) = \arg\min_{g \in G} \mathcal{L}(f, g, \pi_x) + \Omega(g) \tag{1.5.1}$$

We want to approximate the ML function f, in a locality of x. To do that, we sample points from all over the f domain, weighted by π_x, usually implemented

[8]Giving a short overall, SHAP interpret the impact of having a certain value for a given feature in comparison to the prediction we would make if that feature took some baseline value.

as a gaussian kernel:

$$\pi_x(z) = e^{-\frac{||x-z||^2}{\sigma^2}} \tag{1.5.2}$$

The approximation g is obtained via K-LASSO, a combination of feature selection and Lasso regression. LIME chooses only the K most important features because explanations with hundreds of variables could be chaotic and harsh to understand. Theoretically, we may choose any kind of interpretable model as a surrogate model e.g. Decision Trees, Logistic Regression, GLM, etc.

Thinking about a 2-dimensional example, we are building the best linear equation which approximates the ML function f in a certain neighborhood (Figure 1.5.1).

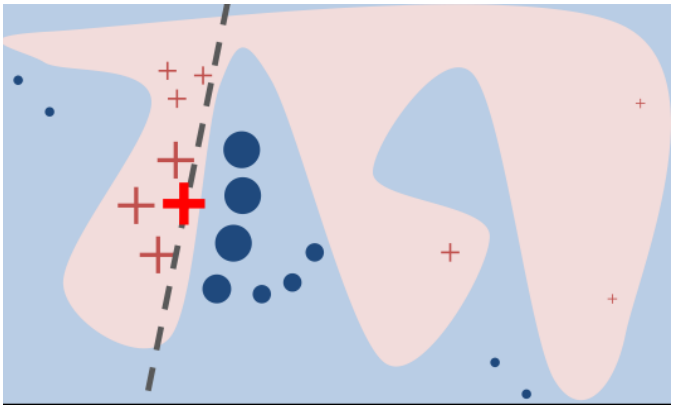


Figure 1.5.1: LIME functioning intuition [12].

An interesting example is LIME applied to image data. One could just highlight the super-pixels with positive influence regarding a specific class (presence-absence of a characteristic). This approach gives an immediate intuition as to why the model thinks an image contains a certain object or is of a certain class (Figure 1.5.2). This kind of visualization enhances trust in the classifier (even if the prediction is wrong), as it shows how the model is reasoning.

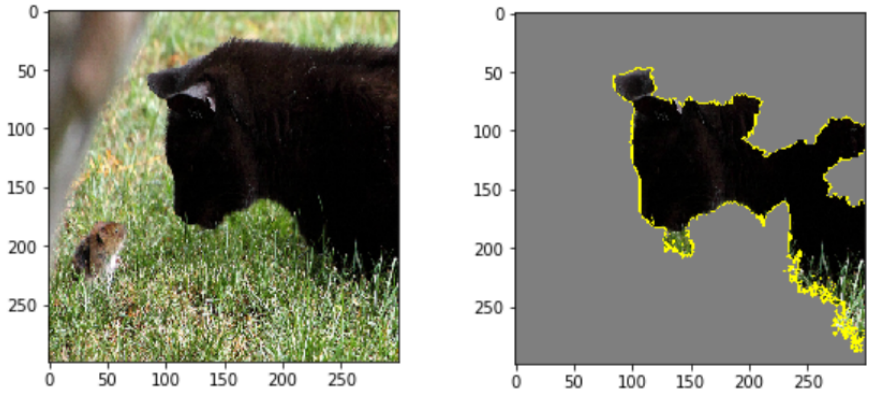


Figure 1.5.2: Why does this image contain a cat? LIME explanation using superpixels.

1.5.2 Generation and Weighting Step

To generate the local approximation around a point x, LIME generates n points x'_i all over the R^d space[9], then also in faraway regions from x. Essentially we are generating a new dataset in which the $\mathbf{X}$ are given by the generated x'_i. To obtain the $\mathbf{Y}$ values, we use the ML function f: we compute $f(x'_i)$, in this way we obtain $\mathbf{Y} = f(\mathbf{X})$. Hence, we are simply plugging each x'_i to the surface of f. A 1D example is shown if Figure 1.5.3.

Moreover, we need a fair way to generate $\mathbf{X}$. LIME standardizes all the features with $\frac{x-\mu(x)}{\sigma(x)}$, then we sample from each variable separately, as if the variable is gaussian. In this way, we will have more points near the variable mean and fewer far away from it. For Categorical variables the approach is similar: we sample the category ID randomly, the probability of obtaining a given category is the same as in the original dataset.

Once the points are generated, we want to weight them referring to their distance to x, since we are not interested in far-away points. Like shown in 1.5.2, LIME uses a Gaussian kernel. If we define the distance as $D(x'_i, x) = ||x'_i - x||$

[9]Some works tried to generate points only close to reference point x, but still present issues [21] [22].

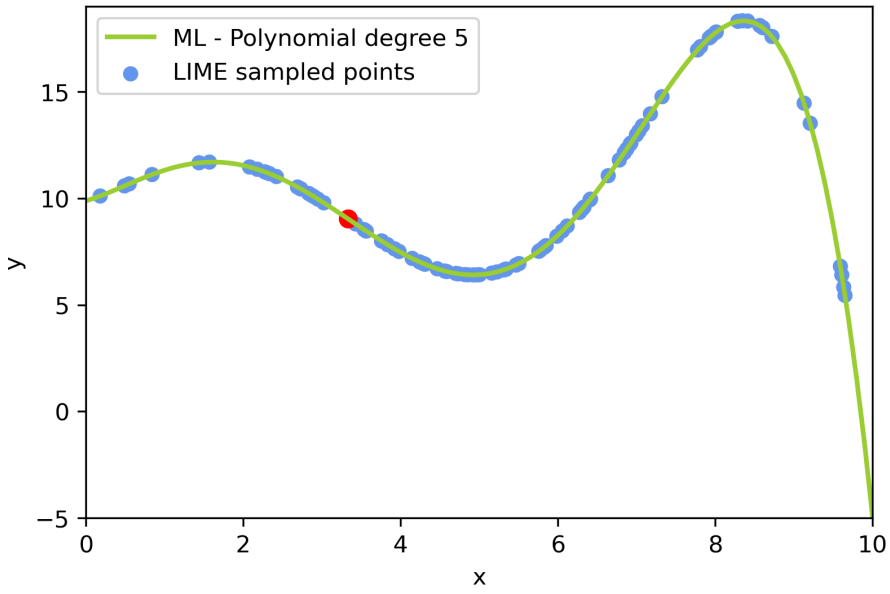


Figure 1.5.3: LIME Sampling.

and kw the kernel width:

$$\pi_x(x'_i) = e^{-\frac{||x'_i - x||^2}{kw}} \tag{1.5.3}$$

$\pi_x(x'_i)$ returns a value between $[0, 1]$, the higher the closer to the reference point x. The only tunable parameter is kw, which defines the radius of the circle drawn around the reference point.

1.5.3 LIME Issues

LIME is a powerful method: is one of the few that works for tabular data, text, and images and has the flexibility of all the model-agnostic methods. On the other hand, it is not immune from problems. One of them is the intrinsic instability from two points of view:

- when we submit very close samples, we would expect to obtain similar explanations.

- when we perform different explanations on the same sample, we would expect to have the same result.

First of all, we define the notion of robustness as something concerning the prediction's explanation concerning changes in the input leading to that prediction. It means that if the input being explained is slightly modified, then we expect the explanation to change a little, according to the modification [23]. Keeping this definition, LIME is not considered robust: two very close points can have completely different explanations.

Furthermore, LIME is unstable because repeated explanations, with equal settings, may have different outcomes. This is due to the generation step, which creates a different dataset for each LIME explanation [24].

Chapter 2

Global-Lime

The goal of Global-Lime is to create a global model-agnostic explanation technique inspired by the LIME approach on local explanations, both in regression and classification. The idea is to create an ensemble of interpretable models (linear regressions) spatially separated, that approximate well an unknown curvy ML model function $f(\mathbf{X})$; with $\mathbf{X} \in R^d$ d-dimensional space and $Y = f(\mathbf{X}) \in R$. In regressions, we consider Y as our continuous variable predicted by the ML model, in classification Y can be regarded as the probability of class C, namely $P(X = C)$.

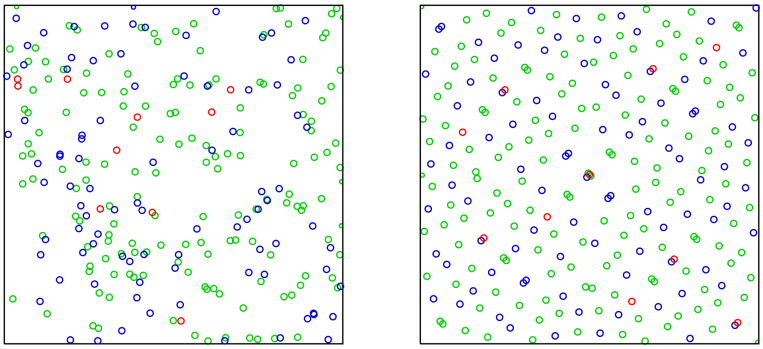
The macro-steps are:

1. **Generation Step**: generate points all over the domain of the ML model, contained in the R^d space.

2. **Clustering**: create clusters exploiting generated points. In each one, a linear regression is fitted to approximate the points in the cluster. The final result is a group of clusters, each one containing a regression, that "covers" well the ML f surface.

2.1 Generation Step

Differently from LIME, in Global-Lime we want to uniformly sample in the fairest way possible the whole space. Not existing a reference point, here there is no notion of weight like in LIME. To create a lattice to cover the space we need n points for each variable, then if d is the number of dimensions n^d points, which is unfeasible with a high number of variables. We can use a sampling strategy that covers well the ML function with fewer uncorrelated points and that guarantees more evenly sampling, such as Sobol sequences (Figure 2.1.1) [25].



(a) 2D Pseudorandom sequence.

(b) 2D Sobol sequence.

Figure 2.1.1: Sampling Examples.

Sobol sequences are quasi-random low-discrepancy sequences. It means that are sequences of values with the property that for all values of M, a subsequence $x_1, ..., x_M$ has a low discrepancy. The discrepancy of a sequence is low if the proportion of points in the sequence falling into an arbitrary set B is close to proportional to the measure of B, as would happen on average (but not for particular samples) in the case of an equidistributed sequence. These sequences use a base of two to form successively finer uniform partitions of the unit interval and then reorder the coordinates in each dimension.

Here we report the mono-dimensional case. We start with initialization numbers m_i, at least a tuple of two (m_1, m_2) randomly chosen real numbers. However, each assumed m_i must satisfy two criteria:

- it must be an odd integer.
- it must be less than 2^i.

Practically $m_1 < 2$, $m_2 < 2^2$, $m_3 < 2^3$ and so on. However, this does not mean that any integer that satisfies these two criteria would generate a good quality low discrepancy sequence[1]. The values of the next m_k can be derived by (m_1, m_2) with a formula starting from the chosen primitive polynomial. For simplicity, here we do not report it.

Then, we create direction numbers $v_i = \frac{m_i}{2^i}$ and gray code numbers g_n, obtained from the index n of the number we want to generate in the series: $G(n) = n \oplus \lfloor n/2 \rfloor$, with g_n its binary representation. Sequence number are generated as:

$$x^n = g_1 v_1 \oplus g_2 v_2 \oplus g_3 v_3 ... \tag{2.1.1}$$

An example of sampling for a 3D function is shown in Figure 2.1.2.

2.2 Partitioning (Clustering)

The idea of this phase is to group a certain number of points, to create partitions (or clusters) in which a criterion, such as R^2, is locally optimized, leveraging an approach inspired by MOB (Model-based Recursive partitioning) [27]. The R^2 score or the coefficient of determination provides a measure of how well-observed outcomes are replicated by a model, so how much the approximated function is "close" to the real one in terms of outcomes. Here we are searching for the best linear approximation in each partition, weighting complexity, and performance. It is clear that if we create very fine-grained partitions, the approximated model will be very tight to the real one, but the results will be a

[1]The quality of the random number depends on the set of initialization numbers. There are a few studies that give the right initialization numbers [26].

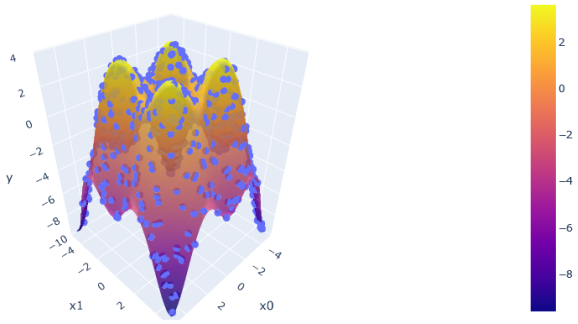


Figure 2.1.2: Example of sampling for 3D function $y = x_0 sinx_0 + x_1 sinx_1$.

complete fragmentation of the domain and an incredibly rigid and slow performance model. Furthermore, all these considerations are bound to the number of available sampled points.

The MOB algorithm is structured in these 4 macro-steps [27]:

1. fit a parametric model to a data set.
2. test for parameter instability over a set of partitioning variables.
3. if there is some overall parameter instability, split the model with respect to the variable associated with the highest instability.
4. repeat the procedure in each of the children until stop conditions are met.

In our case, we will change the instability parameter with a goodness criterion of the approximation in the current partition (equivalent to R^2). The criterion will be described in the next sections.

2.2.1 Model Segmentation Approach

We start describing the ratio behind the MOB algorithm, then we will report how it fits to our purposes. Consider a parametric linear model $M(Y, \beta)$ with a

d-dimensional vector of parameters $\beta \in \mathcal{B}$ and observations[2] $Y \in \mathcal{Y}$. If we define a certain objective function, $\Psi(Y, \beta)$ and we take n observations Y_i, $M(Y, \beta)$ can be fitted trying to minimize Ψ.

$$\hat{\beta} = \arg\min_{\beta \in B} \sum_{i=1}^{n} \Psi(Y_i, \beta) \tag{2.2.1}$$

$\hat{\beta}$ can be found using estimation techniques e.g. maximum likelihood. The aim is to find a global model which fits all n observations. This task is performed recursively partitioning the initial domain D in respect to the d variables, until we find good sub-partitions which locally approximate well f. Roughly speaking, f represent the selected ML model and M its approximation in a locality created through the a subset of the n observations and the computation of the $\hat{\beta}$.

Given B the number of partitions $\{B_b\}$ in which each domain will be divided recursively, the best $\hat{\beta}$s can be easily computed locally optimizing Ψ, then minimizing $\sum_{b=1}^{B} \sum_i \Psi(Y_i, \beta_b)$. Practically, successive greedy optimization (local) are performed to obtain the best possible global score function (the minimum of it). In this case, the number of partitions in which we will split a domain will be fixed $B = 2$, in a similar way to what is done in a classic decision tree (Figure 2.2.1).

2.2.2 Algorithm

The idea is to fit in each node a regression. To assess if the node needs to be split, the R^2 of the node and its number of observations Y_i are checked. If $R^2 < \tilde{R^2}$, where $\tilde{R^2}$ is the minimum R^2 to reach in each leaf, and there are sufficient points, the split is performed. The split is performed in the point and the variable which in a sense maximizes the difference between the 2 future partitions, then creating the best regressions in each partition. The best β for a node are computed putting the derivative of the objective function (score

[2]For an observation $Y \in \mathcal{Y}$, we mean a matrix X of independent variables and a column y of the corresponding ouput values, then $(X; y)$

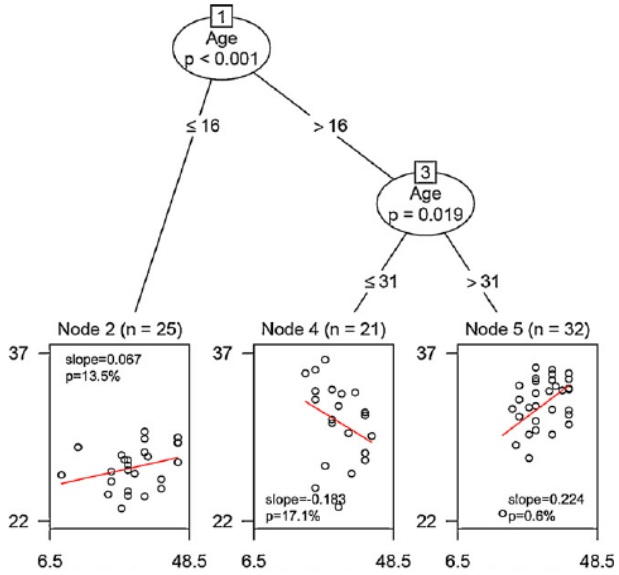


Figure 2.2.1: MOB tree example.

function) to zero, then finding its minimum:

$$\frac{\partial \Psi(Y, \beta)}{\partial \beta} = 0 \tag{2.2.2}$$

Summing up the steps are:

1. Fit the model on all the observations of the current node.
2. Fit a regression on the node x and compute the R^2 for D_x. If a split is necessary and possible ($R^2 < \tilde{R^2}$ and we have a sufficient number of points), search for $z \in z_1, ..., z_d$ and $z_i = \hat{z}$ which locally optimize Ψ.
3. split the node on $z_i = \hat{z}$.
4. repeat until stopping conditions for all nodes are met.

2.2.3 Objective Function and Score Function

We can consider as the global objective function Ψ the Mean Squared Error:

$$MSE = \Psi(X, t, \beta) = \frac{1}{2}(y - X\beta)^T(y - X\beta) \tag{2.2.3}$$

where X is a $n \times d$ matrix of the observations, containing the independent variables, and y is the outcome column $y = f(X)$ of dimension n, β is the vector of dimension d of the coefficients of a linear regression fit on the observations of a node. Moreover, X is simply a group of observations without the output column, which is y. Working with the MSE is very similar to working with the R^2 [3], so minimizing the MSE means optimizing the R^2 (pushing it towards $R^2 = 1$).

The score function ψ related to a subset of the data X will be the derivative of the MSE with respect to β:

$$\psi(X, y, \beta) = X^T X\beta - X^T y \tag{2.2.4}$$

We order the dataset D_x of the node x in an increasing way with respect to a variable z_i, then we compute $\psi(X, y, \beta)$ cumulatively, so increasing the X matrix with a new observation in increasing order ($X_{i,p}$ partition p for variable i). For example if X is composed of 3 rows x_1, x_2, x_3, we compute $\psi(X, y, \beta)$ with $X = \begin{pmatrix} x1 \end{pmatrix}$, then $X = \begin{pmatrix} x1 \\ x2 \end{pmatrix}$ and $X = \begin{pmatrix} x1 \\ x2 \\ x3 \end{pmatrix}$. The βs used in $\psi(X, y, \beta)$ are the ones of the regression fitted on the domain D_x of the whole node x. At the beginning and the end of the dataset, the statistic will be 0, essentially because we are keeping untouched the whole partition. The point inside D_x that yield the highest value of ψ represents the best split because it tells us that the range of the variable until the point considered has the best Least Squares estimates, the most different from the ones we had for the entire range. Hence, practically we are minimizing the global MSE through local maximum problems, in which we find the best points for the split, so the points where the initial regression with coefficients β on the node x perform the worst. Again, it is important to underline the βs used in $\psi(X, y, \beta)$ are the ones of the regression fitted on the domain D_x of the whole node x.

The dimension of $\psi(X, y, \beta)$ is $d \times 1$. To obtain a scalar which represents our

[3] R^2 is defined as $R^2 = 1 - \frac{RSS}{TSS}$ with TSS total sum of squares and RSS Residual Sum of Squares. After some derivations, the numerator of the MSE is equal to the numerator of R^2 apart for 1/2.

score function, we sum the absolute value of each c_k in $\psi(X, y, \beta) = (c_1, ..., c_d)$, than $\sum_{k=0}^{d} |c_k|$.

If we repeat it for all the variables x_i, we choose the split point with the highest $\sum_{k=0}^{d} |c_k|$ (highest difference from 0).

$$best_{x_i} = \arg\max_p \psi(X_{i,p}, y_{i,p}, \beta) = \arg\max_p \sum_{k=0}^{d} |c_{k,i,p}| \tag{2.2.5}$$

$$best_x = \arg\max(best_{x_1}, ..., best_{x_d}) \tag{2.2.6}$$

Following this approach, the global R^2 decreases for each split performed. A toy example is shown in figure 2.2.2.

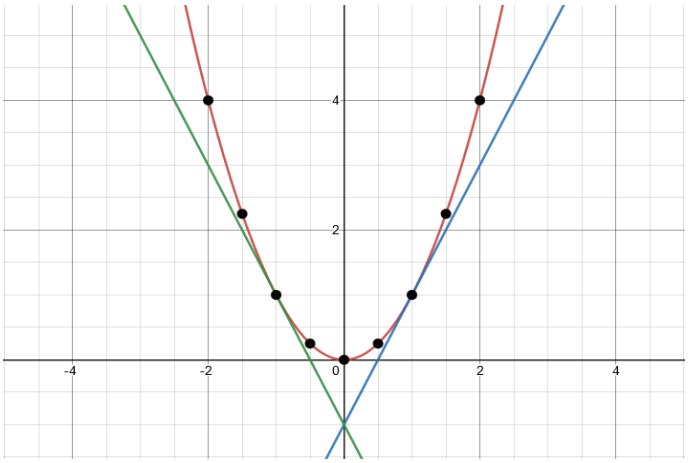


Figure 2.2.2: MOB expected Behaviour on a toy example. Here the ML function is $y = x^2$, if we set the maximum depth of the tree to 2 (so only two leaves), ideally we will obtain the blue and green regressions, which minimize the R^2, with split point $x = 0$.

Summing up, we are not explicitly calculating each "right" and "left" regression to check in which condition the average children R^2 is the highest. This is computationally advantageous because to obtain the β coefficients needed to compute the R^2, we have to do matrix inversions which are computationally

expensive. In the same node x, we always use the computed β_x and then we perform only matrix multiplications via $\psi(X, y, \beta) = X^T X\beta - X^T y$.

Algorithm 1: Global-Lime iterative algorithm

```
Given D the initial domain, root the root of the tree with domain D
R̃² maximum R², Ñ minimum points in a partition
N_j the points in a node j
stack = root
foreach node in stack do
    if R²(node) < R̃² AND nSamples(node) > Ñ then
        β = LinearModel(N_j).coefficients
        variable, split-value = max(ψ(N_j, β))
        node1, node2 = splitNode(node, variable, split-value)
        node.leftChild = node1
        node.rightChild = node2
        stack.append(node1, node2)
    else
        node.regression = LinearModel(N_j)
end foreach
return root
```

2.3 Computational complexity

The difference between using our custom criterion and the leftside/rightside regression approach is a better computational complexity. To assess this, we inspect a single operation that is done in a node at a particular step. Suppose in the node we have n points of dimension d ($n \times d$ matrix). Recalling the computational complexity of some basic matrix operations for a matrix M $a \times b$ [28]:

- **multiplication**: $2ab^2$
- **addition/subtraction**: ab

Algorithm 2: Global-Lime recursive algorithm

Given D the initial domain, root the root of the tree with domain D

$\tilde{R}^2$ maximum R^2, $\tilde{N}$ minimum points in a partition

N_j the points in a node j

if $R^2(node) < \tilde{R}^2$ *AND* $nSamples(node) > \tilde{N}$ **then**

    β = LinearModel(N_j).coefficients

    variable, split-value = $max(\psi(N_j, \beta))$

    node1, node2 = splitNode(node, variable, split-value)

    node.leftChild = recursiveGlobalLime(node1)

    node.rightChild = recursiveGlobalLime(node2)

else

    node.regression = LinearModel(N_j)

    return node

- **transpose**: ab
- **inverse**: a^3 (square matrix)
- **multiplication** ($a \times b$ and $b \times c$): $2abc$

The complexity for computing 2 regressions for $(n-k)$ and k points of dimension d (then we focus on n and not k which vary from 1 to n, so anyway a quantity expressible as dependent from n):

$$C(regressions) = 2(n-k)^2 d + 2k^2 d + 2d^3 = \mathcal{O}(n^2 d + d^3) \tag{2.3.1}$$

The complexity of $X^T X\beta - X^T y$, supposing β is given (transpose, I multiplication, II multiplication, subtraction):

$$\begin{aligned} C(\tfrac{\partial MSE}{\partial \beta}) = 2d(n-k) + (2d(n-k)^2 + d^2) + 2d(n-k) + d = \\ 4(n-k)d + 2d(n-k)^2 + d^2 + d = \\ \mathcal{O}(n^2 d + d^2) \end{aligned} \tag{2.3.2}$$

2.4 Sampling

Another important factor in the whole process is the number of points on which the model is built.

First of all, we want to assure that we fairly sample the ML function, we do not want to lose any type of irregularity and changes of behavior in the original ML function. This means that if we use a not sufficient number of points, we could lose important characteristics of the ML model. Making a naive example, if we use only two points (symmetric in respect to the origin) to sample a parabolic function $y = x^2$, we obtain a constant approximation $y' = K$ which completely erases the original function[4].

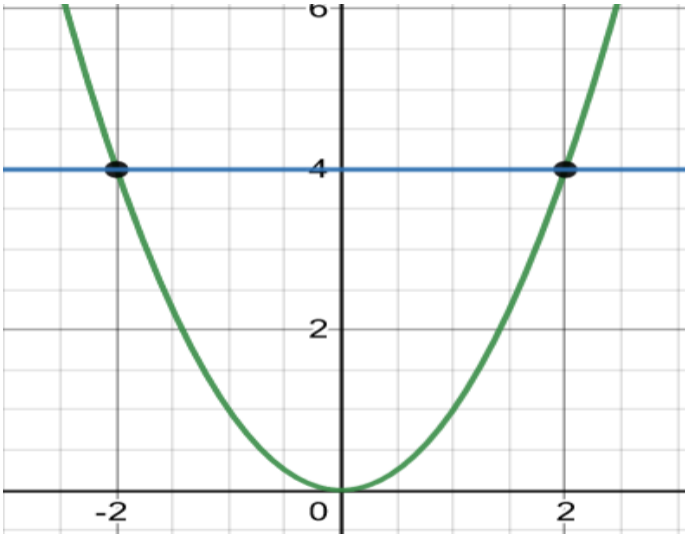


Figure 2.4.1: Example of misleading sampling.

Then we understand that a minimum number of generated points is required

[4]It is very interesting how the concept we are seeking is quite similar to the Nyquist-Shannon sampling theorem. The Nyquist-Shannon sampling theorem is used with signal processing and establishes a sufficient condition for a sample rate that permits a discrete sequence of samples to capture all the information from a continuous signal of finite bandwidth. The problem with that is that requires knowing the frequency and the explicit form of the ML function, which is not available, expressible, or transformable most of the time.

to obtain a minimum adherence of the approximated model to the ML function.

On the other hand, we cannot search the correct number of points using an approach such as a grid search [29], performing every time the complete algorithm due to the great expense of computational power and time. The idea is to create a criterion to check a priori, without running the whole algorithm, the minimum number of points which guarantees that we do not lose information, using only the ML function, the number of points, and criteria that summarizes the concept of not losing information. To do this, we can exploit the incremental nature of the Sobol sequence. Indeed, given 2^i points computed yet, if we are interested in computing 2^j points with $i < j$ positive integers, we can simply "restart" from the 2^i points and then iteratively add the remaining $2^j - 2^i$ points continuing from the last of the previously generated points.

Summing up, the core concept is to obtain the correct number of points which reduces as much as possible, given a certain threshold, the loss of information as described in the previous paragraph. Furthermore, we need a measure of the variation of the "output" value of the sampled points.

2.4.1 Total Variation Approaches

To simplify the exposition, in this section we will start to discuss the monodimensional case. Considering as ML function $y = f(x)$, we will sample the function using Sobol generated points. For a fair coverage of the domain, the number of Sobol points is always a power of 2 (2^m points) [30] [31]. Intuitively, we can order the N points in respect to a variable (in our case the only one x), and then compute the absolute value of the difference between $\delta_i = |f(x_{i+1}) - f(x_i)|$. Gathered all the δ, we can sum them in $V_N = \sum_{k=0}^{N-1} \delta_k$. V_N represents the mono-dimensional case of an approximated version of the total variation for a discrete ordered dataset such as ours Sobol generated.

The idea behind is that the approximated version of the total variation is a non-decreasing function that will converge when reached a certain number of points and gives us an idea of how good we sampled the curve. This is done

by looking at the convergence of V: reached $\tilde{N}$ points, V becomes stable with a confidence interval. Summing up, the total variation for a single variable function f and a partition of ordered points $P = \{x_1, ..., x_N\}$ is defined as:

$$V_P(f) = \sum_{i=0}^{N-1} |f(x_{i+1}) - f(x_i)| = \sum_{i=0}^{N-1} \delta_i \tag{2.4.1}$$

So in a mono-dimensional case, we order the points and we compute the total variation until we reach a satisfying result in terms of V_P stability.

2.4.2 Decision Tree Regressor and Grid Structure

To extend this concept in multiple dimensions, we tried to apply a novel approach to compute the total variation in a n-dimensional domain (also valid to calculate the approximated Riemann integral of a function). The idea is to run a `DecisionTreeRegressor` [32] to create locally valid and constant approximation of the function. Then, we need to extend the boundaries of each "hyper-volume" in a sort of grid as shown in Figure 2.4.2. This is performed

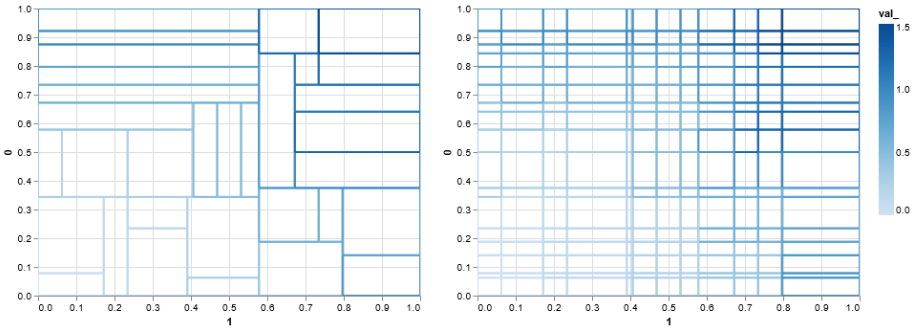


Figure 2.4.2: 2D grid structure obtained from a `DecisionTreeRegressor`

because in multiple dimensions we would not have a concept of the ordering of the points. So we create a lattice of rectangles that can be ordered variable by variable, to compute the total variation for various groups of points. Having applied this for an increasing number of points, we would have seen the total variation converge at a certain step. Hence, the nearer we are to the real

total variation value, the higher the probability that we have not missed any particular variations in the function.

However, the approach theoretically worked, but practically the computational power required was extreme. We will make some examples to be clearer. With only one independent variable, we have only to subtract elements, after having sorted the domain. With two, the situation is as in 2.4.2 where you obtain a matrix, then for each row and column you have to compute the total variation. With three, we have a tensor to elaborate. Each "rectangle" contains inside a matrix for which we have to compute the total variation for each row and column. With four, each "rectangle" has a tensor inside. It is evident that adding simply a dimension brings the problems to another level of complexity. The experiment we performed proved this method to be unfeasible for our aim, so we decided to move on different topics.

Chapter 3

Model Architecture

In this section, the model architecture and the implementation created is described[1]. We will start inspecting the main class for the explanation which is `Glime`, then we will focus on the real model inside, a `MOB` class instance with our custom score function plus an optimization on the calculation of the score function.

3.1 `Glime` class

The `Glime` class is the core of all we described before. Given[2]:

- the predict function of a ML model.
- the range of values for each feature (or the training dataset from which they will be extracted).

[1] All the code which will be described is available on github. In the repository the python MOB implementation, the `Glime` class, and test jupyter notebooks containing useful examples are present. Note that we will describe only the central concepts of the library, without lingering too much on utility modules.

[2] More parameters are available as input elements, but in this phase are not fundamentals and could lead to confusion, so only the most important are listed. If you are interested, you can check the `__init()__`.

- the number of Sobol points to be sampled.
- a dictionary of categorical features with the respective encoder/decoder.

the instance creates an approximation of the ML function and it is capable of giving, for each predicted element, the relative weight for the single features plus the intercept. In the specific, the core methods are:

- `fit()`: here we generate the Sobol points, we rescale them for each variable, given the range of values for each feature. After the `fit()` is concluded, the inside `MOB` instance is fitted and we are ready to explain instances through the other methods.
- `local_coeff(x)`: is the method which explain an instance x given as input. It returns the coefficient for each variable (both numerical and categorical[3].). If it is set through `mode`, it is possible to visualize graphically in a way similar to LIME a horizontal bar chart of the coefficients. An example is shown in 3.1.1.
- `global_coeff()`: shows the overall situation throughout the whole domain for the coefficients of all the variables. This is done through a weighted average of the coefficients in all the leaves, depending on their size[4].
- `predict(x)`: uses the underneath `MOB` instance to perform a predict on a point x, returning the output value of the model.

Practically, `Glime` is a wrapper class that exposes utility functions and hides the complexity of the real underlying model that is `MOB`.

3.2 `MOB` class

`MOB` represents the Python implementation of MOB algorithm described previously, in which our custom score function $\psi(X, y, \beta) = X^T X\beta - X^T y$ has been

[3] See section 4.5.2

[4] Currently the size of a leaf is represented by the numbers of points contained in it

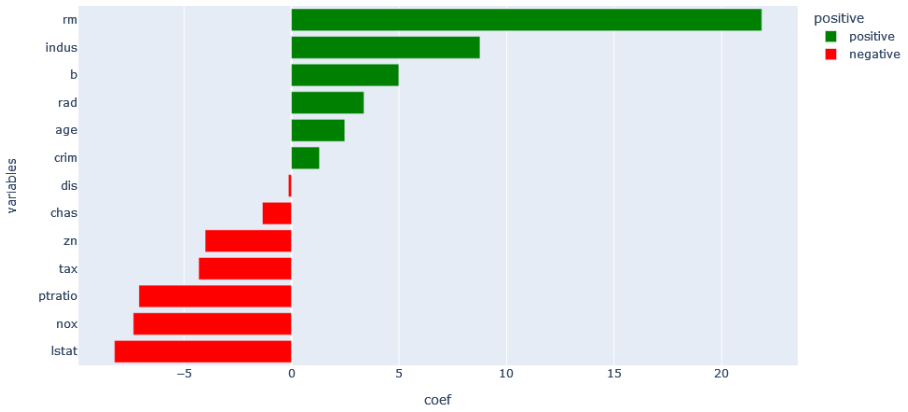


Figure 3.1.1: Feature importance example bar-chart on Boston dataset.

optimized. Then `MOB` builds the tree structure recursively splitting a node starting from the root that contains the starting domain, using the score function, until the stopping criteria are met (we reached a sufficient R^2 or the partition is too small yet). In each leaf is present a scikit-learn `LinearModel` which approximates the slice of the domain of its responsibility.

Furthermore, the nature of `MOB` is compliant with `sklearn.base.BaseEstimator`, exposing the methods `fit()`, `predict()` and the attribute `is_fitted_`. In particular:

- once initialized a `MOB` instance, it is possible to call the method `fit`. The signature of fit is `fit(X, y, weights=None)`, such as a scikit-learn classic estimator. Moreover, X represents the dataset on which we want to train the model. The dataset can be potentially anything with the caveat to contain continuous variables. In our case, `MOB.fit()` is used with the Sobol points generated inside `Glime`. y represent the output or target values in respect to X. This means that $y = f_{ML}(X)$, where $f_{ML}(\cdot)$ is the ML function, practically the `model.predict` of the related ML model. It is also possible to set the initial `weights` of the function. They are a mask

of (True, False), that tells us which point we have to use and which not. If `None`, the whole X is considered as the initial dataset.

- once the model has been fitted, it is possible to use `predict(X)`, which gives the prediction of a `LinearModel` related to each row of X. Through the module `mob_utils`, you can retrieve with `get_pred_node` the node related to a single point x and then simply doing `node.regression` the regression with the coefficients and the intercept, that represent the explanation[5].

The nodes that constitute the tree of `MOB` are instances of the class `Node` in `classes.py`. A node contains several attributes, the most important are:

- `split_var`: the variable on which the node has been splitted.
- `terminal`: `True` if the node is a leaf.
- `left_child` and `right_child` are the child nodes.
- `domain` is an instance of the class `RealDomain` and represents the domain of interest of the node.
- `regression` contains the `sklearn.linear.LinearModel`.

`RealDomain` implements the abstract class `AbstractDomain` and represents the support for multi-variables continuous real domains. It exposes methods such as `contains(x)`, `insert(x)` and most importantly `split(var, value)` that splits the domain in respect to a variable and a value, returning two not overlapped and complementary (in respect to the parent domain) domains. To do that, `RealDomain` is composed by a dictionary of `RealIntervals`, which represents a continuous interval (with the possibility to set the bounds open or closed) for a single variable, with "lower-level" utility functions for checking

[5] A `sklearn.linear_model.LinearRegression` has been used, so to retrieve the coefficients you can access the attribute `coef_`. However, it is not the focus in `MOB`. In fact, it is something we want to do in `Glime` and it is possible without using external modules directly via `Glime.local_coef(x)`.

if it contains an element, splitting the interval, and so on[6]. The split that we perform is binary and retains the nature of the boundary of the original interval. For example, if the original interval $I = [0, 1]$, the generated sub domain with split on 0.5 will be $I_l = [0, 0.5)$ and $I_l = [0.5, 1]$

3.2.1 Optimization of the Score Function

To select the best variable for the split we:

1. order the subset of the dataset regarding a node.
2. compute in an incremental way the score function ψ adding row by row until the dataset is exhausted.
3. select the value for that variable i that maximizes the score function ψ.
4. select the variable j with the best $\psi_{max} = \psi_j$.

Looking at point 2, we can think to compute the value of the score function only for the unique values in a column of the dataset. In fact, if we have multiple identical values, it is useless and potentially harmful to compute the coefficient in "not existing" splitting points. Since the split is virtually a hyperplane that cuts the space into two parts, e.g. if the split is performed in $x_0 = 0$, we cannot bring some 0s in the left partition and others on the right one. Practically, we are checking the score value for a split that is not achievable. Hence, we check the score function only on the first occurrence of a value[7]. Therefore, each time we add k rows, where k is the number of occurrences of the value we are considering (the first row for the first iteration or the previous split point checked).

Another small optimization concerns the parameter `minsplit` passed in the `__init()__` of `MOB`: it controls indirectly the minimum required size of a leaf,

[6] All the classes described are contained in the file `classes.py`.

[7] An important consideration that can be done is that for the nature of the Sobol points it is not possible to have the same value in a certain column. The optimization has been performed if one would use `MOB` as a classic ML model.

setting a minimum number of points contained in a leaf. Whereas it is not possible to create a node with a number of contained points less than `minsplit`, then the split points which will create these situations are excluded a priori from the computation.

Third, recalling $\psi(X, y, \beta) = X^T X\beta - X^T y$, we said that we compute it by adding from 1 to k row to the X matrix. Ideally, we have X_i that represents the subset of the dataset for the value of a split i and X_j, which is the "next" matrix obtained from X_i adding k rows. X_i is a submatrix of X_j, for which we have already computed $\psi_i(\cdot)$: the idea is to reuse the components of $\psi_i(\cdot)$ and add only the part of the sum related to the recently added rows. If we call χ_k the matrix made from the k rows we want to add to X_i, the resulting formula is:

$$X_j^T X_j = X_i^T X_i + \chi_k^T \chi_k \tag{3.2.1}$$

and the same is valid for $X^T y$ (the k new y are denoted as γ_k):

$$X_j^T y_j = X_i^T y_i + \chi_k^T \gamma_k \tag{3.2.2}$$

so putting everything together and adding the β:

$$\psi_j(\cdot) = X_i^T X_i \beta + \chi_k^T \chi_k \beta - (X_i^T y_i + \chi_k^T \gamma_k) = \psi_i(\cdot) + \chi_k^T \chi_k \beta - \chi_k^T \gamma_k \tag{3.2.3}$$

3.2.2 Advanced Optimization

The previous section described how to optimize matrixes multiplications when we extend them by adding iteratively rows at the bottom of the matrix. Hence, we are optimizing the single operation of computing the score, but the score continues to be computed in each node.

If we focus on how the score function is computed, we notice that essentially we compute all the information we need to perform the split in the root node. Moreover, in the root of the tree, we have the entire dataset of Sobol points, we order it and we compute the score cumulatively row by row. The idea is to reuse these pre-computed values to perform the expensive matrix multiplication only once in the root. In fact, the elements of the score function $\psi_i(X_i, y_i, \beta_i)$

in a certain node, so $X_i^T X_i$ and $X_i^T y_i$, can be derived from the computation in the root.

Take as an example the element $X_A^T y_A$ for a node A. We start computing in the root node $X_j^T y_j = (c_{1,j}, ..., c_{d,j})$ as we discussed before for each submatrix for $j \in 1, ..., n$, where n is the number of Sobol points. To simplify the exposition of the intuition, now suppose the score function is only composed by this term and we are computing the score for a fixed variable $\tilde{d}$, then X and y are ordered in respect to $\tilde{d}$. Then the score for X_j and $\tilde{d}$ is $v_j = \sum_{k=1}^{d} |c_{k,j}|$. Doing this for all the $j \in \{1, ..., n\}$, we obtain a vector $\mathbf{v} = (v_1, ..., v_n)$, with the precomputed scores for all the splitpoints cumulatively. If one want to compute the score from row 1 to row $k < d$, it is possible simply taking v_k. But if we are in a subpartition of X, e.g. from row α to row β with $\alpha < \beta \in \{1, ..., n\}$ we cannot compute the value in the same way, but such as a difference of values. In particular the simplified score for the subpartition $\{X\}_{(\alpha,\beta)}$ is equal to $v_\beta - v_\alpha$. Extending it for all the d variables, we will have vectors $\mathbf{v_j}$ with $j \in \{1, ..., d\}$ and then the score for variable j and for a sub-partition $\{X_j\}_{(\alpha,\beta)}$ is equal to $v_{j,\beta} - v_{j,\alpha}$.

The same approach can be replicated for the term $X^T X\hat{\beta}$, but with the difference that $\hat{\beta}$ changes from node to node. Thus, we can simply save $X_i^T X_i$ for all the $i \in \{1, ..., n\}$ in d vectors $\mathbf{w}$ (one for each ordering variable), then multiply the cached value for $\hat{\beta}$ later. Note that here we are saving a matrix of size $d \times d$ and not a scalar, but the functioning is the same $\{X\}_{(\alpha,\beta)}^T \{X\}_{(\alpha,\beta)} = w_\beta - w_\alpha$, given a fixed dimension $\tilde{d}$, where $w_\beta = \{X\}_{(1,\beta)}^T \{X\}_{(1,\beta)}$ and $w_\alpha = \{X\}_{(1,\alpha)}^T \{X\}_{(1,\alpha)}$.

Summing up, we reduce drastically computation time removing matrix multiplications from all nodes except for the root and substituting them with simple

substractions, but inevitably we increase memory consumption[8]. In fact, we perform only a big matrix multiplication in the root note[9] and we maintain two groups of d vectors respectively for X^TX and X^Ty. Furthermore, during the execution when a node becomes a leaf we can delete the score parts regarding its part of the domain, saving some memory. This becomes very simple if we think of the vectors such as dictionaries from which we remove elements when they are not useful.

Formalizing what we exposed in the paragraph, we represent a sub-partition of a matrix M of dimension $n \times d$ from row α to row β, with $\alpha < \beta \in \{1, ..., n\}$, as $\{M\}_{(\alpha,\beta)}$:

$$M = \begin{bmatrix} m_{1,1} & ... & m_{1,d} \\ ... & ... & ... \\ m_{\alpha,1} & ... & m_{\alpha,d} \\ ... & ... & ... \\ m_{\beta,1} & ... & m_{\beta,d} \\ ... & ... & ... \\ m_{n,1} & ... & m_{n,d} \end{bmatrix} \quad \{M\}_{(\alpha,\beta)} = \begin{bmatrix} m_{\alpha,1} & ... & m_{\alpha,d} \\ ... & ... & ... \\ m_{\beta,1} & ... & m_{\beta,d} \end{bmatrix} \tag{3.2.4}$$

We express that is sorted[10] in respect to a variable $\tilde{d} \in \{1, ..., d\}$ with $\{M_{\tilde{d}}\}_{(\alpha,\beta)}$. Given X the initial sobol matrix, $y = f_{ML}(X)$ the value of the ML function and $\hat{\beta}$ the coefficients of the regression in a node, we compute for

[8]Several adjustments are possible. While performing the optimized multiplication, we can remove rows from the loaded Sobol dataset when not useful, whilst adding the computed coefficient in the vectors. This can be done by finding first the ordering for each variable $\tilde{d}$, computing the first coefficient for each variable, then deleting the first row of X from the memory and proceeding this way until the matrix is all consumed. Practically, X is used only in the root node. Doing this, if one wanted to retrieve the values for the split point, it would access from disk only the single tuple containing that value. The same is valid for $\hat{\beta}_s$ of the regression. In this way, we use half of the memory needed, with the price of slower access to the actual value of the split. This will be something considered in future works.

[9]Actually we are performing multiple matrix multiplications, but optimizing the product is like doing one big matrix multiplication $d \times n$ $n \times d$ and $d \times n$ $n \times 1$.

[10]Sorted before partitioning.

each $\tilde{d} \in \{1, ..., d\}$, $j \in \{1, ..., n\}$:

$$v_{\tilde{d},j} = \{X_{\tilde{d}}\}^T_{(1,j)}\{y_{\tilde{d}}\}_{(1,j)} = (c^1_{\tilde{d},(1,j)}, ..., c^d_{\tilde{d},(1,j)}) \tag{3.2.5}$$

then we obtain the first d n-dimensionals vectors ($n \times d \times d$ tensor, $(n \times d) \times d \times 1$ matrixes):

$$\mathbf{v_{\tilde{d}}} = \begin{bmatrix} v_{\tilde{d},1} \\ ... \\ v_{\tilde{d},n} \end{bmatrix} = \begin{bmatrix} \begin{pmatrix} c^1_{\tilde{d},(1,1)} & ... & c^d_{\tilde{d},(1,1)} \end{pmatrix} \\ ... \\ \begin{pmatrix} c^1_{\tilde{d},(1,n)} & ... & c^d_{\tilde{d},(1,n))} \end{pmatrix} \end{bmatrix} \tag{3.2.6}$$

$$\mathbf{v} = \begin{bmatrix} \mathbf{v_1} & ... & \mathbf{v_{\tilde{d}}} & ... & \mathbf{v_d} \end{bmatrix}$$

similarly we do the same for the other term:

$$w_{\tilde{d},j} = \{X_{\tilde{d}}\}^T_{(1,j)}\{X_{\tilde{d}}\}_{(1,j)} = \begin{bmatrix} \omega^{1,1}_{\tilde{d},(1,j)} & ... & \omega^{1,d}_{\tilde{d},(1,j)} \\ .. & ... & ... \\ \omega^{d,1}_{\tilde{d},(1,j)} & ... & \omega^{d,d}_{\tilde{d},(1,j)} \end{bmatrix} \tag{3.2.7}$$

where ω is the generic element of the $X^T X$ matrix. Then we obtain the second tensor (a tensor $n \times d \times d \times d$, $n \times d \times (d \times d)$ matrixes):

$$\mathbf{w_{\tilde{d}}} = \begin{bmatrix} w_{\tilde{d},1} \\ ... \\ w_{\tilde{d},n} \end{bmatrix} = \begin{bmatrix} \begin{pmatrix} \omega^{1,1}_{\tilde{d},(1,1)} & ... & \omega^{1,d}_{\tilde{d},(1,1)} \\ .. & ... & ... \\ \omega^{d,1}_{\tilde{d},(1,1)} & ... & \omega^{d,d}_{\tilde{d},(1,1)} \end{pmatrix} \\ ... \\ \begin{pmatrix} \omega^{1,1}_{\tilde{d},(1,n)} & ... & \omega^{1,d}_{\tilde{d},(1,n)} \\ .. & ... & ... \\ \omega^{d,1}_{\tilde{d},(1,n)} & ... & \omega^{d,d}_{\tilde{d},(1,n)} \end{pmatrix} \end{bmatrix} \tag{3.2.8}$$

$$\mathbf{w} = \begin{bmatrix} \mathbf{w_1} & ... & \mathbf{w_{\tilde{d}}} & ... & \mathbf{w_d} \end{bmatrix}$$

so that the score for a variable $\tilde{d}$, and the subset (a, b) of the sorted matrix $\{X_{\tilde{d}}\}_{(a,b)}$, $\{y_{\tilde{d}}\}_{(a,b)}$ and $\hat{\beta}$ of the node is:

$$\psi(\cdot) = (w_{\tilde{d},b} - w_{\tilde{d},a})\hat{\beta} - (v_{\tilde{d},b} - v_{\tilde{d},a}) \tag{3.2.9}$$

where only $\hat{\beta}$ needs to be computed. Hence recalling 2.3, the computational cost for a single operation in a node becomes (plus a subtraction of 2 $d \times d$ and 2 $d \times 1$ matrix instead of the multiplications):

$$\begin{aligned} C(\tfrac{\partial(MSE)}{\partial x}) = \cancel{2dn} + (\cancel{2dn^2} + d^2) + \cancel{2dn} + d + (d^2 + d) = \\ \cancel{4nd} + \cancel{2dn^2} + d^2 + d + (d^2 + d) = \\ 2d^2 + 2d = \\ O(d^2) \end{aligned} \tag{3.2.10}$$

However, the total memory cost is high, in particular for d very big (where f is the size in byte of a floating point number, $minsplit << n$ and $d << n$):

$$\begin{aligned} m = (n - 2minsplit)d[(d * d * f) + (d * 1 * f)] = \\ 8d^2 f(d+1)(n - 2minsplit) = \\ O(fd^2(d+1)n) = \\ O(fd^3 n) \end{aligned} \tag{3.2.11}$$

Also hybrid approaches are thinkable, in which we "cache" only parts of the products[11] or e.g. only $X^T y$ which is only $d \times 1$.

It is worth also mentioning that fixing d and n, $X^T X$ is invariant for all the applications of `Glime` except for the scaling (but it is a reversible operation), because Sobol sequences points without the scrambling option are fixed. Also, the ordering is fixed. Hence, some kind of pre-computing of matrixes and indices is thinkable in future applications.

Another optimization one could think of, and that is implicit in the whole reasoning above, is to optimize the sorting process in all the nodes. In fact, we perform it in every node, then we can simply save a parallel matrix with indices of sorting for each variable so that they are selected in a subset from a node, using the information of its domain of competence (in the implementation are the `weights`). This could lead to a small improvement in performance, because

[11] For example, we can think to save only k * γ $X^T X$ products, where k are the best/most promising k split-point in the root node, so the points in which $\psi(\cdot)$ is high, and γ is the "neighborhood" of that points. Obviously, after the first split, we have no guarantees that one of that points will be used.

if we use a fast algorithm such as quicksort, the computational time for a dataset long n and d variables is $O(d * nlogn)$ [33], so almost linear yet.

`Note`: all these optimizations still need to be implemented.

3.3 Other Modules

Inside the repository are present other modules that contain utilities and other useful methods. Inside `glime_utils.py` there are some methods used also by the `Glime` class. For example `get_sobol_x()` and `get_sobol_y()` to generate the Sobol points and `rescale_sobol_points()` to rescale them in the range of the variables passed in the training dataset or specified in a dictionary containing `variable:range`. Inside `mob_utils.py` are present methods to retrieve nodes inside `MOB`, to interact with the tree, and other methods used by `MOB` during the expansion (e.g. the workhorse function for computing the scores). A module specific for visualization, called `visualize.py` and based on Altair and Plotly has been developed to visualize the regressions, the domains created by `MOB`, the coefficients of `Glime`, and the surfaces of `Glime` in two dimensions when possible and useful.

Chapter 4

Testing and Results

4.1 Testing MOB Implementation Behaviour

Once described the implementation of Global-Lime, the next step is to conceive a series of tests to inspect if the model performs correctly. The initial test will be performed on ad-hoc functions and dataset, then on real data.

4.2 Test on `DomainTree` Custom Function

Summing up, the core of Global-Lime is to take an arbitrary ML function f and, through a fair sampling strategy, to build a series of not overlapped hyperplanes which approximate f. Hence, we obtain an ensemble of explainable models, each one valid in a certain partition of the initial domain.

The first test we thought about is to see if the model is capable of reconstructing with a certain margin a model which is very similar to how Global-Lime works for. Hence, we built a tree structure that recursively partition a domain, retaining for each variable a fixed percentage of their initial domains as a minimum split, with the possibility of a random stop in the creation process to allow generating more diverse tree models. This is done because we want that the dimension of each the partition will be large enough, in order to have

a sufficient number of points to reconstruct the hyperplane with Global-Lime. The tree fits in every obtained partition a random linear model[1]. An example of `DomainTree` behaviour is shown in Figure 4.2.1.

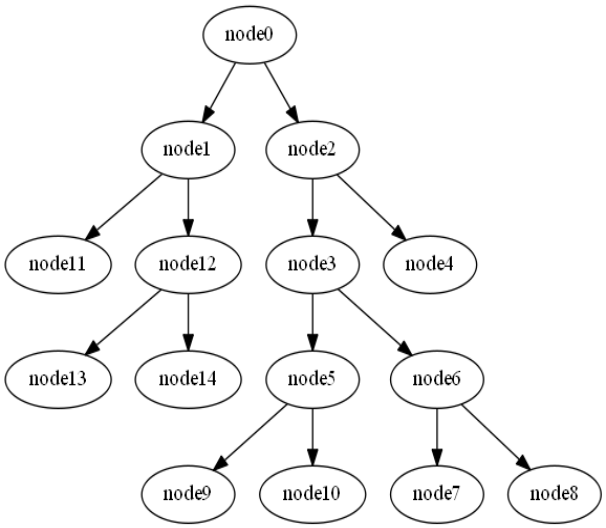


Figure 4.2.1: Example of a `DomainTree`.

Each leaf of the tree represents a partition spatially not overlapped with the others. `DomainTree` takes as input the variables and their initial domains, then recursively partition the domain with the rule: left leaf has the right bound not included and right leaf has the left bound included, the other bounds remain unchanged.

For clarity we make an example: if we have only one variable, the starting interval is $[0, 1]$ and the split point is $x = 0.5$, we will have $node_{left} = [0, 0.5)$ and $node_{right} = [0.5, 1]$. Acting this way garantees to obtain spatially separated domains with a simple and clear rule. The result for a bidimensional case (two variables $(x0, x1)$) is shown in Figure 4.2.2.

For sure, it could be helpful to visualize what happens in a single dimension, so with the initial domain of a single variable and only one output variable. The

[1]The linear model are LinearModel of sklearn in which the parameters $coef\_$ and $intercept$ are randomly generated by a uniform distribution between parametric $[a, b]$.

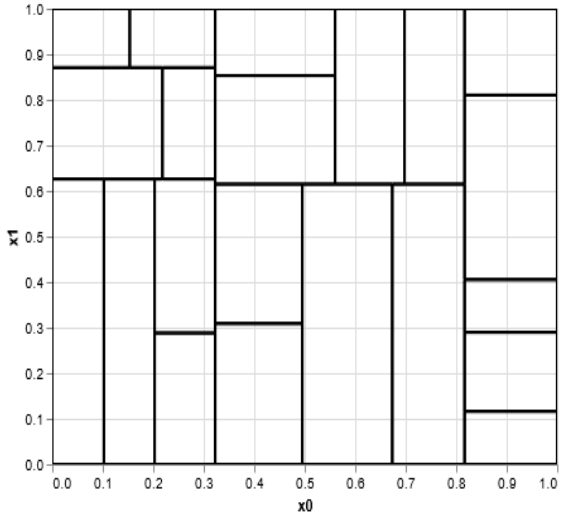


Figure 4.2.2: Grid structure of `DomainTree` domains. The initial domains for the variables are $x0 \in [0, 1)$ and $x1 \in [0, 1)$ and the tree maximum depth was fixed to 5. In each rectangle, a plane with random coefficients is inserted. Note that this figure represent the partitioning of the domains of $x0, x1$, y is not depicted.

result is shown in Figure 4.2.3

4.2.1 Key Performance Indicators (KPI) for `DomainTree`

To inspect if the model correctly reconstructed the DomainTree function, we followed `DomainTree` approaches:

- **qualitative**: we inspected the functions created from a graphical point of view to see whether the overall behaviour is the expected one, e.g. in Figure 4.2.4 of the idea behind this KPI. We compare only the reconstructed domains (independent variables X) and not the output (y).
- **quantitative**: we developed 3 KPIs which consider the number of partitions, the overall R^2 (as a weighted average in respect to the leaf size) and the number of matching partitions between the models.

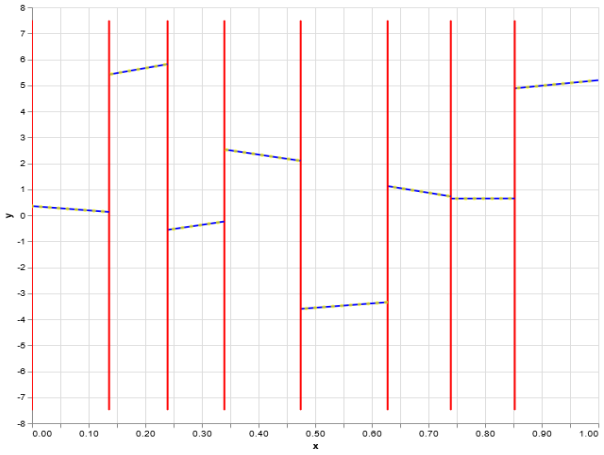


Figure 4.2.3: 1-dimensional case for tree domain. The initial domains for the variables is $x_0 \in [0, 1)$. The blue lines represent the regressions created by `DomainTree` with coefficients between $[-5, 5]$. The red lines represent the different leaves, while the yellow dots are the sampled points using Sobol sequence (2^6 points).

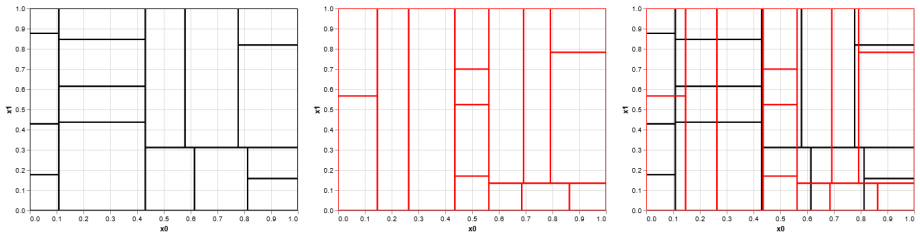


Figure 4.2.4: Comparison of two domains. Here to highlight the differences and show the idea behind the KPIs two random `DomainTree` are reported, then overlapped in the last figure. In this case, we would consider the result as poor, due to the fact that no partition matches another one.

The quantitative tests are namely:

- **partition number**: checks the number of partitions created by the `DomainTree` and `Glime` so that the difference can be computed.
- $\mathbf{R^2}$: checks the minimum, maximum and average R^2 weighted by the

partition size obtained by Global-Lime.

- **matching partitions**: checks the number of partitions that coincide between Global-Lime and `DomainTree` in respect to a certain confidence interval. The confidence interval is a multiple of the minimum distance between Sobol points $\delta(d,n) = \frac{1}{2}\frac{\sqrt{d}}{n}$, where d is the number of dimensions and n is the number of points [34]. Global-Lime can perform the split only on a value that coincides with a value present in the dataset, hence even in case of optimal performance, it could be possible a slight waterfall misalignment between original and reconstructed partitions. Thus, even only one slightly shifted partition can lead to a compromised reconstruction of the following ones. For this reason, a confidence interval is needed.

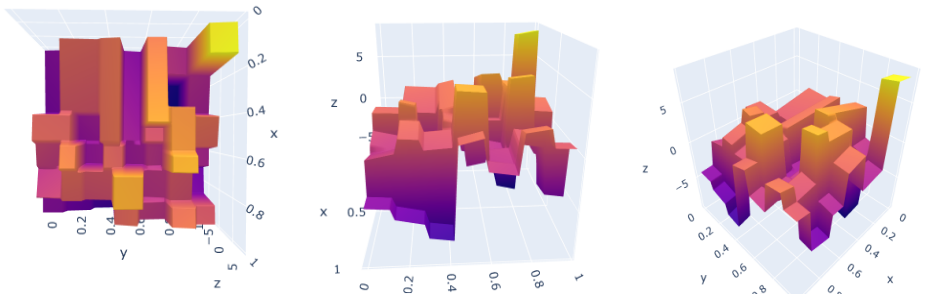
  A partition $P \in R^d$ is valid if all the intervals that compose the partition match an original one. Having defined the minimum distance between two points $\delta(d,n)$ and a constant γ positive integer, an interval $[a,b]$ matches with $[\alpha,\beta]$ if:

$$a \in [\alpha - \delta\gamma, \alpha + \delta\gamma] \wedge b \in [\beta - \delta\gamma, \beta + \delta\gamma] \tag{4.2.1}$$

  So with $\delta(d,n) = \frac{1}{2}\frac{\sqrt{d}}{n}$:

$$a \in [\alpha - \frac{1}{2}\frac{\sqrt{d}}{n}\gamma, \alpha + \frac{1}{2}\frac{\sqrt{d}}{n}\gamma] \wedge b \in [\beta - \frac{1}{2}\frac{\sqrt{d}}{n}\gamma, \beta + \frac{1}{2}\frac{\sqrt{d}}{n}\gamma] \tag{4.2.2}$$

As already shown in Figure 4.2.2, it is possible to extend the experiment in more than one dimension. However, an interesting case of visualization can be the 3D case, so when we have 2 independent variables and a dependent variable computed in this case as the outcome of the linear regression in a certain region of the initial domain. An example of a randomly generated 3D-tree function, to better visualize the partitioning image 4.2.2 shown before, is depicted in Figure 4.2.5.



(a) Upper view of a 3D `DomainTree`.

(b) Front view of a 3D `DomainTree`.

(c) Lateral view of a 3D `DomainTree`.

Figure 4.2.5: 3D view of a `DomainTree` with $z = f(x, y) = \beta_0 + \beta_1 x + \beta_2 y$ from -8 (violet) to 8 (yellow). x and y are the independent variables.

4.2.2 Results

From a qualitative point of view, `Glime` and then our `MOB` implementation performed very well on this type of task. If `MOB` is run with a sufficient number of Sobol points, then the original ensemble of linear models created by the `DomainTree` are perfectly reconstructed. Obviously, if the number of Sobol points is too small, we are not capable of capturing all the variation on the original model, then we miss some hyperplanes and the partitions are less in number. This affects the global R^2 of `MOB`[2]. In the table 4.1 are reported the outcomes of some experiments. The metrics inspected are the KPI mentioned above plus a delta on the number of created partitions (`delta-p` = δ_p = `textglime_partition_number - original_partition_number`). For all the experiments:

- $\gamma = 2$
- `minplit` which is the minimum percentage referred to the initial range of

[2]Note that the global R^2 is computed as the weighted average of the R^2 in the single leaves. The weight is given by the number of points "contained" in the leaf.

an interval to retain[3] is set to 0.05/0.1. Thus it is the minimum size in percentage for each variable.

- `tree-depth`=6, which means 6 levels of splitting starting from the root and proceeding to the children (5 levels of split for the 2 children generated from the root etc.).
- the seed for the experiment is `seed=42`.
- the coefficients of the regression are sampled from a uniform distribution in [-5, 5].
- the interval for each variable is [0, 1].

Regarding the internal `MOB` instance of `Glime`, the parameters are set to:

- `minsplit` = 10.
- `stopping_value` = 0.99.
- `stopping_crit` = "R2".

As we can see in Table 4.1, with the tree function is always possible to achieve a good R^2 score that is the core of the whole experiment (see if `MOB` is capable of retrieving multiple discontinuous hyperplanes). In particular, you have to pay attention to the correct number of Sobol points, weighting performance and score. A higher number of points means a better score (score=1 is always approachable with a sufficiently big number of points), but the space and time complexity rises. Consider also that for a good space covering a correct dimensioning is fundamental. Since you can use only a number of points which is a power of 2 (2^m), the decision of the correct m impacts heavily the overall performance: just changing m of 1 unit leads to a much more complex model and higher computing and memory consumption. Besides that, we can conclude the model guarantees good performance on our custom tree function

[3]For example if we have an interval [0, 1] and minsplit is set to 0.1, the smallest achievable sub-partition will be of length 0.1.

Table 4.1: Tests on `DomainTree` with various parameters.

*2 to the power of

d	**minsplit**	**sobol-points***	$\mathbf{R^2}$	**matching-p**	**p-number**	$\delta_\mathbf{p}$
1	0.1	6	0.643	1	7	-2
1	0.1	7	1.000	7	7	0
1	0.05	7	0.871	3	12	-3
1	0.05	9	1.000	11	12	2
3	0.1	9	0.798	1	32	2
3	0.1	11	0.968	1	32	27
5	0.1	11	0.956	0	32	33
5	0.1	12	0.977	0	32	54
10	0.1	12	0.978	0	32	96
10	0.1	14	0.991	0	32	178

and it behaves correctly also in presence of drastic discontinuities such as that shown in Figure 4.2.5.

Comparing the number of partitions of the ground truth model and of Global-Lime, in general the functioning seems to be very coherent in a small number of dimensions, whilst it degrades e.g. for d=10 when we have a delta of 178. At the end this is not a great problem, it is only an index of how fragmented the model is in overall. This behaviour is also caused by the increasing number of points in input, besides the increasing dimension number; for example for d=3 the original partition number is 32, with 2^9 points delta is 2 but $R^2 = 0.798$, with 2^{11} points delta is 27 but $R^2 = 0.968$ [4].

Summing up, Global-Lime proved to perform well in this scenario. Two qualitative examples for 1D and 2D are reported in Figure 4.2.6 and 4.2.7.

[4] A simple demo containing also a visualization section is available in `/notebooks /glime_discontinuous_test.ipynb`

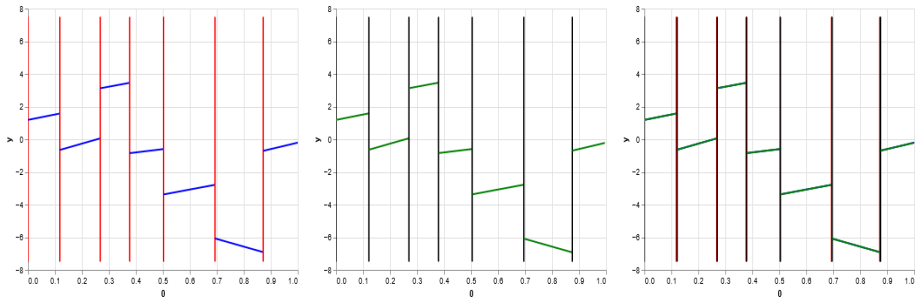


Figure 4.2.6: `DomainTree` and `Glime` 1D-example. The first image from left represents the function generated by `DomainTree`, where the red lines are the random splits and the blue ones are the regressions. The central image is the function reconstructed by `Glime`. As we can notice from the last image, which is the overlap of the first and the second, `Glime` perfectly reconstructs the original model.

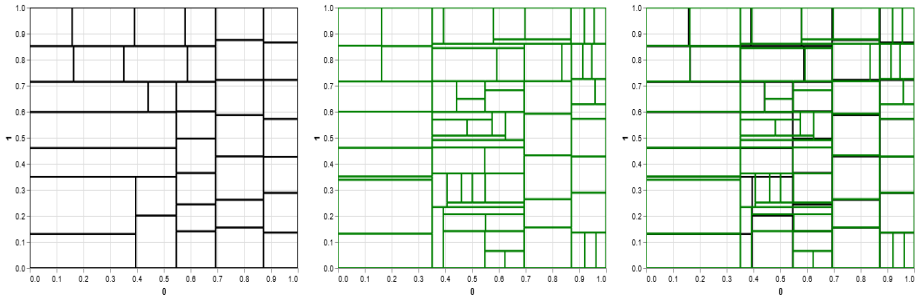


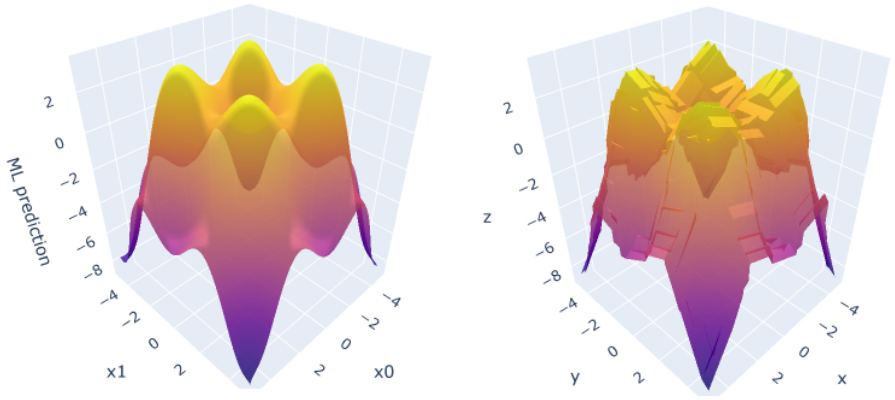
Figure 4.2.7: `DomainTree` and `Glime` 2D-example. The first image from left represents the domains generated by `DomainTree`. The central image is the domain reconstructed by `Glime`. As we can notice from the last image, which is the overlap of the first and the second, `Glime` approximates well the original lattice (the main splits are reproduced almost perfectly), even if the obtained domain is more fragmented.

4.3 Testing on Simple Math Functions

The next step is to test `Glime` in a continuous domain scenario with more complex functions. Even in this scenario, the model showed to perform well

with a collection of math functions. It is worth to discuss the case of $f(X) = f(x_1, ..., x_n) = \sum_{i=1}^{d}(x_i sin\ (x_i))$ depicted in Figure 4.3.1 and in the monodimensional case in Figure 4.3.2. As we can see, `Glime` correctly wraps the original ML function in an ensemble of hyperplanes oriented coherently with the original function. A slight drawback could be the instability shown near the 4 peaks, in which multiple planes with quite different coefficients are close to each other. Indeed, it is not trivial to solve, because it is inherent to the nature of the original function.

In the table 4.2, a group of examples showing the average R^2 for selected functions is reported . As we can see, in one dimension the approximation is always good, also with few points. It is more difficult if we have a particularly unstable function, with higher dimensionality and not sufficient points. In the table 4.2 the example of $cos(\sum_{i=1}^{d} x_i^2)$ has been proposed. With d=2 and 2^{12} Sobol points it has a score of 0.557, but with 2^{14} points it approaches a score of 0.784 that is sufficiently good. The others examples show how it is possible to reconstruct the original model with a good R^2, even if it becomes hard with very wavy functions such as $e^{-\prod_{i=1}^{d}|x_i|}cos(\sum_{i=1}^{d}\pi x_i)$.



(a) $\sum_{i=1}^{d}(x_i sinx_i)$ function.

(b) $\sum_{i=1}^{d}(x_i sinx_i)$ reconstructed by `Glime`.

Figure 4.3.1: **x**sin**x** example with 2 indipendent variables, $y = x_0 sinx_0 + x_1 sinx_1$

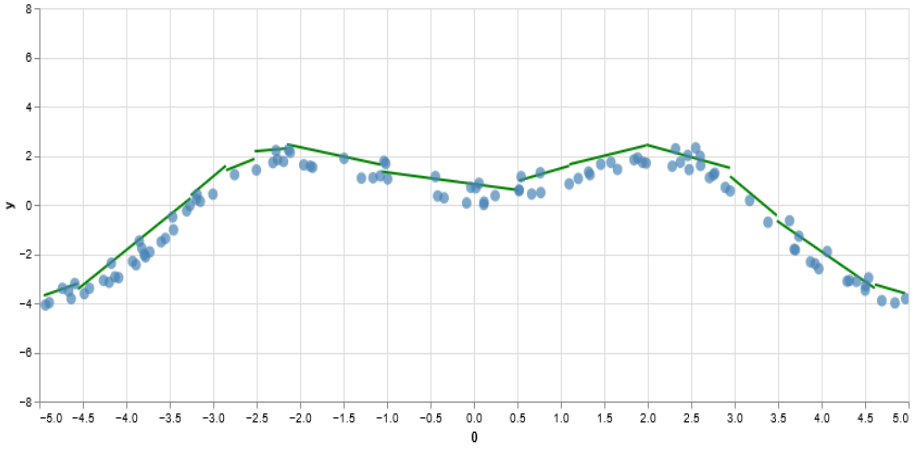


Figure 4.3.2: xsinx 1D-example. The blue dots represent the points from the original dataset created from xsinx, the green line is the approximation obtained from `Glime`.

4.4 Testing on a Real Numerical Dataset (Boston Data)

After some artificial experiments on ad hoc and selected functions, we decided to test `Glime` on a real numerical dataset to see how it performs in a plausible scenario. The dataset is "The Boston Housing Dataset" [35]. This dataset contains information collected by the US Census Service concerning housing in the area of Boston. Even if is quite small (506 cases), it represents one of the most used datasets for algorithm benchmarking. The variables are 14 and nominally represent:

1. **CRIM** - per capita crime rate by town.
2. **ZN** - proportion of residential land zoned for lots over 25,000 sq.ft.
3. **INDUS** - proportion of non-retail business acres per town.
4. **CHAS** - Charles River dummy variable (1 if tract bounds river; 0 otherwise).

Table 4.2: Testing `GLime` on math functions with various parameter. The domain for all the variables in the function is [-5, 5]. *2 to the power of

dimensions	**function**	**sobol-points***	$\mathbf{R^2}$
1	$\sum_{i=1}^{d}(x_i sin x_i)$	9	0.979
1	$\sum_{i=1}^{d}(x_i sin x_i)$	10	0.965
2	$\sum_{i=1}^{d}(x_i sin x_i)$	10	0.785
2	$\sum_{i=1}^{d}(x_i sin x_i)$	12	0.931
5	$\sum_{i=1}^{d}(x_i sin x_i)$	12	0.791
5	$\sum_{i=1}^{d}(x_i sin x_i)$	14	0.848
1	$cos(\sum_{i=1}^{d} x_i^2)$	9	0.876
1	$cos(\sum_{i=1}^{d} x_i^2)$	10	0.90
2	$cos(\sum_{i=1}^{d} x_i^2)$	12	0.557
2	$cos(\sum_{i=1}^{d} x_i^2)$	14	0.784
5	$cos(\sum_{i=1}^{d} x_i^2)$	14	0.643
5	$cos(\sum_{i=1}^{d} x_i^2)$	15	0.647
1	$e^{-\prod_{i=1}^{d}\lvert x_i\rvert} cos(\sum_{i=1}^{d} \pi x_i)$	9	0.931
1	$e^{-\prod_{i=1}^{d}\lvert x_i\rvert} cos(\sum_{i=1}^{d} \pi x_i)$	10	0.961
2	$e^{-\prod_{i=1}^{d}\lvert x_i\rvert} cos(\sum_{i=1}^{d} \pi x_i)$	14	0.859
2	$e^{-\prod_{i=1}^{d}\lvert x_i\rvert} cos(\sum_{i=1}^{d} \pi x_i)$	15	0.915
5	$e^{-\prod_{i=1}^{d}\lvert x_i\rvert} cos(\sum_{i=1}^{d} \pi x_i)$	14	0.557
5	$e^{-\prod_{i=1}^{d}\lvert x_i\rvert} cos(\sum_{i=1}^{d} \pi x_i)$	15	0.564

5. **NOX** - nitric oxides concentration (parts per 10 million).

6. **RM** - average number of rooms per dwelling.

7. **AGE** - proportion of owner-occupied units built prior to 1940.

8. **DIS** - weighted distances to five Boston employment centres.

9. **RAD** - index of accessibility to radial highways.

10. **TAX** - full-value property-tax rate per $10,000.
11. **PTRATIO** - pupil-teacher ratio by town.
12. **B** - $1000(Bk - 0.63)^2$ where Bk is the proportion of blacks by town.
13. **LSTAT** - % lower status of the population.
14. **MEDV** - Median value of owner-occupied homes in $1000s.

The tasks for the dataset are two:

1. **NOX** is to be predicted.
2. **MEDV** is to be predicted.

We decided to perform task 2 (**MEDV**).

4.4.1 Model Selection and Pre-processing

As the ML model on which run `Glime`, `XGBRegressor` from the `xgboost` library has been used. XGBoost is an optimized distributed gradient boosting library [36]. Gradient boosting models are a class of ensemble ML algorithms used for classification and regression tasks. These ensembles are obtained from groups of small decision trees, where trees are added and fitted one at the time to adjust the global prediction, as a result of the multiple contributions of the single models. Models are fit with an arbitrary loss function and optimized with a gradient descent optimization algorithm, similarly to the neural networks. In Table 4.3 are reported the hyperparameters used for the regressor.

The dataset is already well-formed, hence no particular preprocessing procedure is needed. Only a `MinMaxScaler` [37] has been used to scale all the features in a range from 0 to 1. Afterwards, the dataset has been divided into train and test with 85-15 proportions. The performance achieved by `XGBRegressor` on the dataset are:

- MSE (Mean Squared Error): 9.048.

Table 4.3: Hyperparameters used in `XGBRegressor` for Boston Housing dataset. If not specified, the default has been used.
*y_train.mean()

parameter	**value**
booster	gbtree
random_state	42
base_score	22.649*
n_estimators	40000
learning_rate	0.01
max_depth	2
early_stopping_rounds	300
seed	100

- MAE (Mean Absolute Error): 2.328.

`Glime` has been run, achieving $R^2 = 0.90$ with 99 partitions created. The hyperparameters are reported in table 4.4.

Table 4.4: Hyperparameters used in `Glime` for Boston Housing dataset. If not specified, the default has been used

parameter	**value**
n_sobol_points	11
predict_function	`regressor.predict(x)`
minsplit	15
stopping_value	0.99

We may ask to perform explanations on new samples or samples deriving from the dataset. For example, we can take the third row and ask for an explanation of the output value of the model (36.31905). The output is shown in Figure 4.4.1.

Discussing the most prominent coefficients, we understand that: **LSTAT**, **NOX**, **PRATIO** negatively influence **MEDV** which makes sense. If **NOX** is high, the area is polluted and less valuable, as well as **LSTAT** and **PRATIO** are important indicators of the wellness of the zone. Moreover, **RM** seems to be the most important parameter for **MEDV** and again it is legit to say that a flat is more valuable if there is a higher number of rooms per dwelling. Again, also all the other coefficients are coherent from what we could imagine before running the explanation.

4.5 Testing on Categorical Data and Classification

Since this moment, we have been discussing implicitly the behaviour of `Glime` only in presence of pure numerical dataset and in regression tasks, but this does not mean that `Glime` cannot work in presence of categorical features or classification task. In this section we will see that with the correct preprocessing

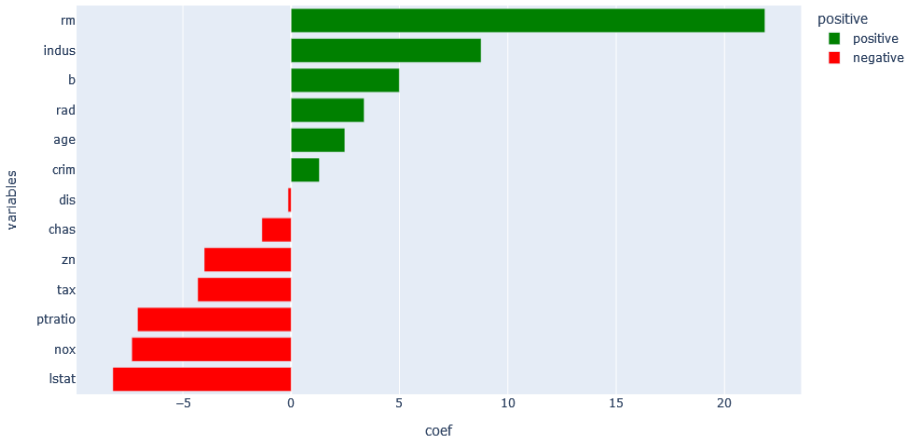


Figure 4.4.1: Explanation on a row of Boston Dataset.

and leveraging a `TargetEncoder` [38] in combination with the coefficients of an explanation, is possible to interpret also categorical (and then also ordinal) data and adjust `Glime` for classification[5].

4.5.1 Credit Risk Modeling Dataset of Example

The dataset of example is a dataset of Credit Risk Modeling [39] in which we have to predict how capable each applicant is of repaying a loan. In particular, we want just a flag 0/1 to understand if a creditor is good or bad (0 good and 1 bad). In particular, the dataset is composed by 1225 rows and 14 + 1 of output (BAD) columns[6]. We have chosen to take only 9 of them, which are the most significant:

1. **YOB** - Year of birth.
2. **NKID** - Number of children.
3. **DEP** - Number of other dependents.
4. **PHON** - Is there a home phone (1=yes, 0 = no).
5. **SINC** - Spouse's income.
6. **AES** - Applicant's employment status (categorical column).
7. **DAINC** - Applicant's income.
8. **RES** - Residential status.
9. **DHVAL** - Value of Home.

The dataset is quite well-formed, so the preprocessing was minimal. All the numerical features[7] have been processed through `MinMaxScaler`, then the dataset

[5]Since now, we have been testing the model for tasks of binary classification e.g. predict YES/NO, leveraging the `predict_proba(x)` of the classification models, but extensions for general classification problems are conceivable.

[6]The dataset is downloadable from GitHub.

[7]For the categorical ones refer to the next section.

has been divided in train and test with 85-15 proportion. As ML model we used `XGBClassifier` again from the `xgb` library. In table 4.5 are reported the hyper-parameters for the classifier. The model reached a ROC=0.7 and AUC=0.68, showing difficulties mostly in classifying bad creditors.

Table 4.5: Hyperparameters used in `XGBClassifier` for CRM dataset. If not specified, the default has been used. *(y_train == 0).sum()/(y_train == 1).sum(). **y_train.mean()

parameter	**value**
objective	"binary:logistic"
scale_pos_weight	2.7992*
booster	"gbtree"
random_stae	42
base_score	0.2632 **
n_estimators	40000
learning_rate	0.001
max_depth	2
early_stopping_rounds	3000

Then `Glime` has been called over the classifier with the hyperparameters shown in table 4.6. The achieved R^2 is 0.84 and 213 partitions have been

Table 4.6: Hyperparameters used in `Glime` for CRM dataset. If not specified, the default has been used

parameter	**value**
n_sobol_points	12
predict_function	`classifier.predict_proba(x)[:, 1]`
minsplit	15
stopping_value	0.95

created. As shown in the tables, the predict_function is the second column of predict_proba [40] of the classifier. So the probability of a sample to be 1 (BAD).

4.5.2 Target Encoding for Explainable Categorical Features

The problem with categorical features is that they do not have an intrinsic ordering, so we are not able to understand clearly how a feature varies the output. To perform a linear regression, we need numerical values. Thus, we must encode the categorical column into numbers and each category is bounded to a certain real number. Assigning these numbers, we have to decide the category ordering, even if it is not explicit. If we assign equally spaced numbers (e.g. 0 1 2 3 ...), we imply that the "conceptual" distance between categories is the same, but several times this is not true. If we try to solve this problem with a one-hot encoding, we fall into the curse of dimensionality.

An example is **AES**, that can have the following values:

- V = Government.
- W = housewife.
- M = military.
- P = private sector.
- B = public sector.
- R = retired.
- E = self-employed.
- T = student.
- U = unemployed.
- N = others.

- Z = no response.

We can think of using a `LabelEncoder` [41] or an `OrdinalEncoder` [42] to put an artificial ordering on the variables, but it is not useful since the ordering do not respect the real contribution which each different value of the feature gives to the output. A solution could be to use a `OneHotEncoder` [43], but the problem with this type of encoding is that it greatly increases the dimensionality of the training data (by adding a new feature for each unique category in the original dataset). This often leads to poorer model performance due to the curse of dimensionality [44]. Furthermore, none of these encoders is capable of giving a correct proportion to the values of the single labels and to discover coincident labels in the features in respect to the output.

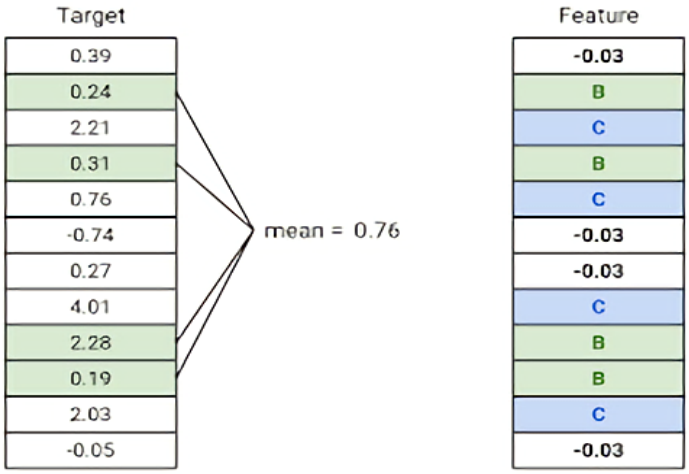


Figure 4.5.1: Target encoding.

A `TargetEncoder` allows to retain information about different categories, without creating other columns and ordering them proportionally to the contribution they give to the target value (output feature). For each feature, we replace each category with the mean target value for samples that have that category. An example is shown in Figure 4.5.1.

The interesting aspect of this type of encoding is that if we multiply the

obtained values for the coefficient of the feature in a certain locality, we obtain the concrete variation in the target value. In our case, because the target value is between 0 and 1, the product represents the increase in the percentage of the probability to be a **BAD** creditor. To show this concept and to dispose of a useful explanation also for categorical features, a small module inside `Glime` has been implemented. The module retrieves the mapping inside the `TargetEncoder`, identifies coincident labels and then is capable of visualizing graphically the increments.

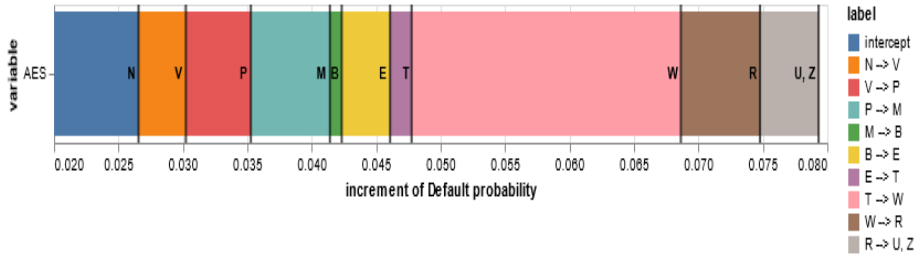


Figure 4.5.2: Target Encoding plus `Glime` coefficients on CRM dataset for AES feature.

The case of **AES** is shown in Figure 4.5.2. As we can see the labels U (unemployed) and Z (no response) coincident. Furthermore, again the results we obtain make sense:

- the categories with the highest probability of default and which mostly influence the output are respectively W, R and (U, Z).
- the difference between the right-side features and left-side is extremely underlined in the jump T->W.
- best creditors are labels V, P, N, usually people with high money availability.

We can do the same for **RES** as shown in Figure 4.5.3.

In this way, adding the bar chart for the single feature importance (shown in Figure 4.5.4) we are capable of explaining all the features on the dataset and

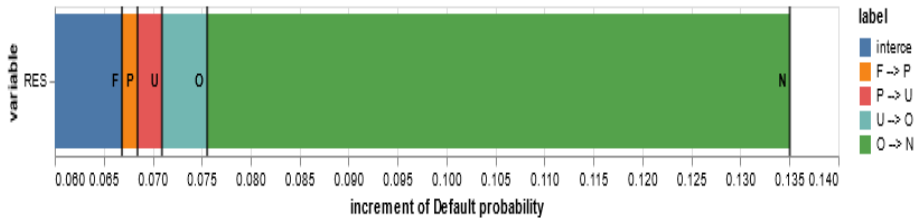


Figure 4.5.3: Target Encoding plus `Glime` coefficients on CRM dataset for RES feature.

which weight they have on the probability of default. Referring again to Figure 4.5.4, we understand that higher **SINC** and **DAINC** (the spouse's income and the applicant income) lower the probability to be a bad creditor, as we could think. Similarly, **RES** and **AES** are important to determine the probability of default; and we understand which is the magnitude of the single labels thanks to Figure 4.5.2 and 4.5.3. Also, you can look that higher **NKID** and **YOB** mean higher probability to be a bad creditor.

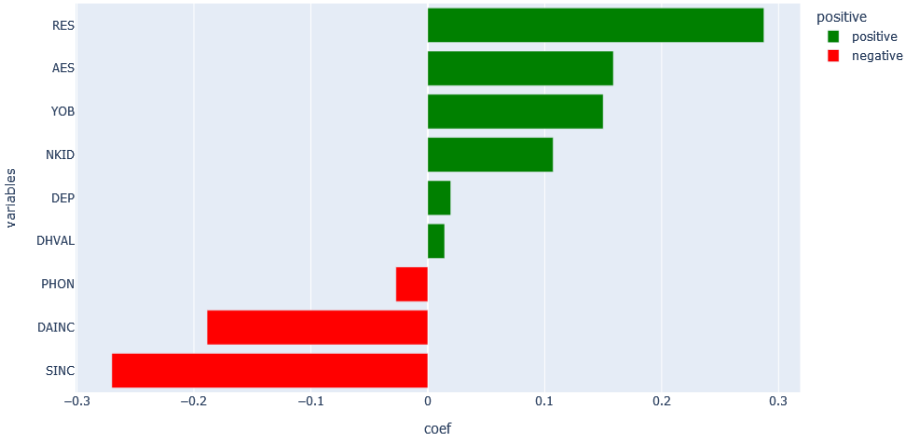


Figure 4.5.4: Target Encoding plus **Glime** coefficients on CRM dataset.

4.6 Dashboard for Visualization

Once built the model and trained on a dataset, it is important to visualize simply what we have produced via the approximated model. A simple use case could be: given a pre-trained model obtained on a certain dataset, we are interested in searching for an explanation on a single individual, also projecting a value for a certain variable. Practically, we would be interested in making also "what-if" analysis: adding in input an existing entry but with slightly modified values (creating plausible future scenarios) or even never seen before individuals, then trying to understand which parameters will be more important than the others, inspecting the coefficients of the linear approximation of the node bounded to that element. For this purpose, a simple dashboard has been built. The dashboard takes as input a pre-trained model and a dataset, then it visualizes the "current slice" of the model (and of the dataset) which we are interested in. The control panel selects variables, values and make a prediction on a single variable in the sidebar. There is also a "full-what-if" part in which you can insert values for the x-variables and see the coefficients in a barplot form for that element.

For the visualization part, the frameworks used are Streamlit for the dashboard front-end in pure python and Altair plus Plotly for image visualization.

4.6.1 Streamlit Front-end and Altair

Streamlit is an open-source Python library that facilitates to create and share custom dashboards as web apps for machine learning and data science. Even if it is quite new[8], it is a valid alternative to more complex frameworks like Dash. The main characteristic of Streamlit is that all the app is a single .py file in which the layout and the dynamics of the app are described and it is always executed from top to bottom. This means that every time an event occurs, the entire script is executed: a behaviour that leads to potential great inefficiency.

[8] A stable 1.0 version was released only on October 2021.

Because of that, another core concept is caching through `st.cache`[9] decorator. A function is marked through `st.cache` and then the Streamlit engine checks:

- The input parameters that you called the function with.
- The value of any external variable used in the function.
- The body of the function.
- The body of any function used inside the cached function.

If this is the first time Streamlit has seen these four components with these exact values and in this exact combination and order, it runs the function and stores the result in a local cache. Then, next time the cached function is called, if none of these components changed, Streamlit will just skip executing the function altogether and, instead, return the output previously stored in the cache [45].

It is also possible to build stateful applications with the concept of session. In Streamlit each browser tab starts with a blank state with no variables shared between sessions. Session State is a way to share variables between reruns, for each different user session, and callbacks can be used to manipulate state and changing on the state [46].

Altair is a declarative statistical visualization library for Python, based on Vega and Vega-Lite. Vega-Lite is a high-level grammar of interactive graphics. It provides a concise, declarative JSON syntax to create an expressive range of visualizations for data analysis and presentation.

4.7 Dashboard Layout

The dashboard is structured as (Figure 4.7.1):

- **sidebar**: in the sidebar there is the possibility to upload the selected dataset and a pre-trained model to visualize. In a demo scenario, a mock

[9] with `streamlit as st`.

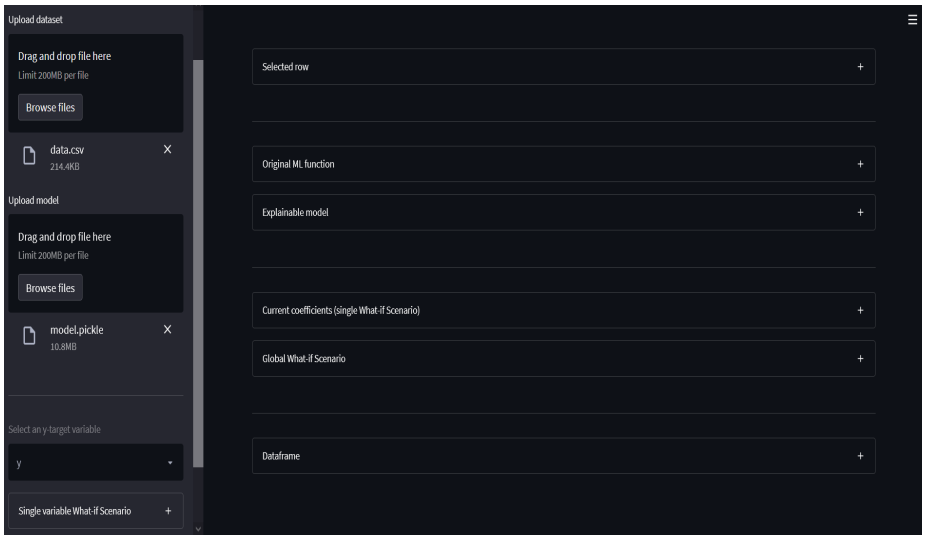


Figure 4.7.1: Dashboard Layout.

dataset containing the Sobol points of the ML function is visualized. Thus, it is possible to select the target variable and perform through an expander a "Single variable What-if scenario". Practically, you can select a row of the dataset you uploaded (visualized in the first container of the application), select an independent variable and see what happens if you change the value of only that variable. This is visualized in the body of the app, if you look at the graph of the two functions. The moving dot represents the "Single variable What-if scenario".

- **function visualization**: the first container contains the graphs of the ML function, the explainable model and the selected row. It is important to say that only the points of the ML graph that belong to the nodes displayed in the explainable model are visualized, and not the entire dataset.

- **coefficients visualization**: the first expander visualizes the coefficients of the current node (decided throughout the single variable projection and the selected row) for each variable and highlights the value of the current variable, the local coefficient and the explanation for the target variable.

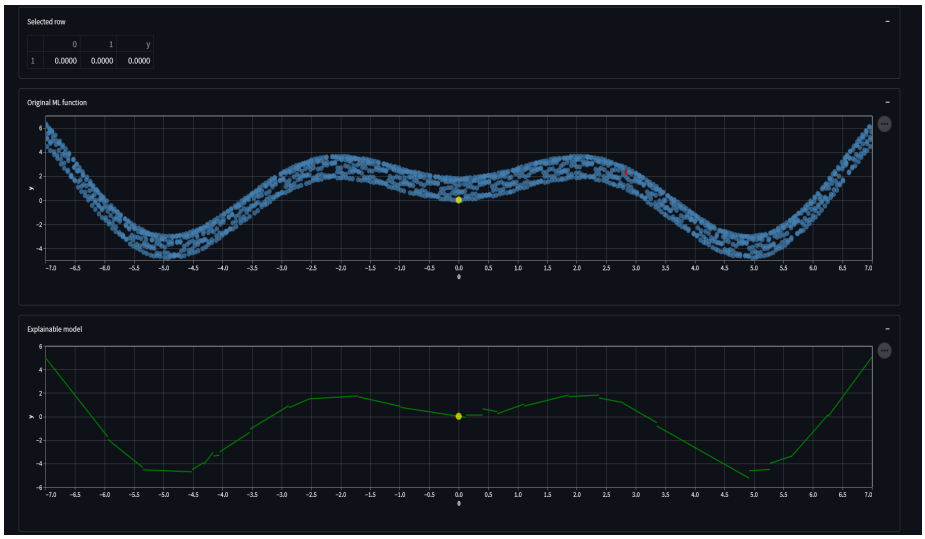


Figure 4.7.2: Dashboard Functioning. In this example the dataset is composed of 2^{12} points in $D = [-7, 7] \times [-7, 7]$, two variables (x0, x1) and the $y = x_0 sinx_0 + x_1 sinx_1$. The blue dots are the points of the dataset relative to a fixed region of the space, depending on what row of the dataset was selected and which parameters were varied. The yellow dot represents the point selected and changed by the "what-if" analysis. The green line is the `Glime` reconstructed model. In this example, on the x axis as indipendent variable is placed variable 0, while variable 1 is fixed since we are performing a 2D representation of a 3D dataset.

The second expander permits to perform a full what-if scenario for all the variables, without being bound to a certain existing individual in the dataset. Then the coefficients and the prediction of the target value are displayed. In case of a categorical dataset, after inserting the name of the categorical features, a list of target encoding representations multiplied by the coefficient of the current node is displayed such as in Figure 4.5.3.

Finally, a chart containing the trend of all the coefficients varying a certain feature is displayed (Figure 4.7.4).

Figure 4.7.3: Dashboard Coefficients and What-If Analysis. The dataset is the same showed in Figure 4.7.2. The first box contains the coefficients of the current node. The delta on the explanation and the y-value are shown on the right-side. No target encoder bars are shown because the dataset of example is purely numerical. The second box enables the user to perform a "full" what-if analysis, without being bound to a particular row of the dataset.

4.8 Future Developments

As shown in the previous sections, Global-Lime proved to work well in various different scenarios, giving important insights about the ML model functioning and the contributes of each feature (locally and globally). However, there are several open themes and problems which it is valuable to mention:

- correct **dimensioning of the number of Sobol points**: to have a fair sampling of the space the number of Sobol points is a power of 2. Furthermore, for a good adherence of Global-Lime to the ML function, it is necessary to have a sufficient number of points. In section 2.4 we

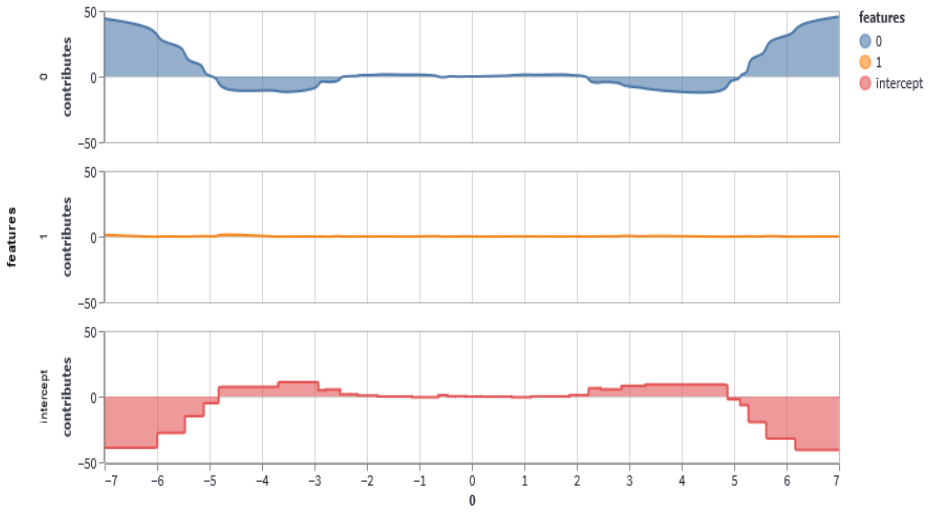


Figure 4.7.4: Dashboard Sliding Coefficients.

tried to develop a method to assess the minimum number of points, given a certain adherence to the function. These efforts did not bring valuable results, since the Total Variation approach was computationally extremely expensive[10] and theoretically harsh. Because of that, now there is no a priori criteria to decide which is the correct number of Sobol points. Clearly more points lead to better precision, but the more points we add, the more memory and computational power we consume.

- **implementation of MOB**: the current implementation of MOB is a recursive depth-first expansion of a tree, so difficult to parallelize. A possible improvement could be to substitute this implementation with an iterative one, in which we maintain a first-in-first-out stack of open nodes, starting from the root and proceeding with the expansion. The advantage of the iterative implementation is the easier parallelization of multiple CPUs and advanced threading.

[10] A library for computing the total variation of a function using a decision tree regressor was built, but due to the poor performance of the method was set aside, waiting for further ideas.

- **implementation of the score function**: now the score function is computed using the optimization for matrix multiplication described in 3.2.1. However, more advanced approaches can be done to speed up the algorithm, such as that exposed in 3.2.2.
- **GPU support**: the model is not designed to work with GPUs. Further developments could turn Global-Lime compatible with GPUs using frameworks such as CuPy, JAX or Numba.
- **extention to general classification**: the model works only in case of regression ad binary classification. With some efforts, it is possible to extend the method for general classification.

Conclusion

In this work, we have designed and implemented a new global model-agnostic technique for Machine Learning explainability, starting from the LIME approach and trying to generalize it. We implemented a Python version of MOB with a custom split criterion, which can be used as a classic Machine Learning model for regression and binary classification tasks.

Unifying MOB and quasi-random sampling through Sobol sequences, we developed a new explainability technique: Global-Lime. Global-Lime works as a wrapper for the original model and supplies an interpretable version of it, from both a local and a global point of view. After a series of tests on ad-hoc functions, mathematical functions of increasing difficulty and two real datasets (one purely numeric and one containing also categorical features), we proved the correct functioning of Global-Lime. Moreover, we leveraged target encoding in combination with our technique to provide a useful explanation also for categorical features, depicting precisely how they impact the output value. Furthermore, we discussed the possible drawbacks of the technique, the possible mathematical and computational optimizations for our novel score function and the future developments for the library.

To display more clearly the functioning of Global-Lime, a small visualization module for displaying partitions, regressions, Sobol points, coefficients, and categorical features contribute has been built. Furthermore, a simple dashboard for model analysis that encloses all the previous efforts have been developed.

Concluding, we hope to have contributed enriching the developing field of

Machine Learning explainability, providing a novel technique for a more transparent AI.

Bibliography

[1] C. Molnar, *Interpretable Machine Learning, A Guide for Making Black Box Models Explainable*. 2019.

[2] I. Sample, *Computer says no: Why making ais fair, accountable and transparent is crucial*, Nov. 2017. [Online]. Available: https://www.theguardian.com/science/2017/nov/05/computer-says-no-why-making-ais-fair-accountable-and-transparent-is-crucial.

[3] "Europe plans to strictly regulate high-risk ai technology," *AAAS Articles DO Group*, 2021. DOI: 10.1126/science.abb3741.

[4] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of Research and Development*, vol. 3, no. 3, pp. 210–229, 1959. DOI: 10.1147/rd.33.0210.

[5] T. M. Mitchell, *Machine learning*. McGraw Hill, 2017.

[6] C. Bishop, *Pattern recognition and machine learning*. Springer, 2007.

[7] K. Soni, *Supervised vs. unsupervised learning*, Jul. 2020. [Online]. Available: https://towardsdatascience.com/supervised-vs-unsupervised-learning-14f68e32ea8d.

[8] O. Chapelle, B. Scholkopf, and A. Zien, *Semi-supervised learning*. MIT Press, 2006.

[9] G. Hilton and T. J. Sejnowski, *Unsupervised learning: Foundations of Neural Computation*. Massachusetts Institute of Technology, 1999.

[10] A. Beck and M. Kurz, *A Perspective on Machine Learning Methods in Turbulence Modelling*. Oct. 2020. DOI: 10.13140/RG.2.2.17469.69608.

[11] D. Silver, A. Huang, C. J. Maddison, *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016. DOI: 10.1038/nature16961.

[12] M. T. Ribeiro, S. Singh, and C. Guestrin, *"why should i trust you?": Explaining the predictions of any classifier*, 2016. arXiv: 1602.04938 [cs.LG].

[13] B. Goodman and S. Flaxman, "European union regulations on algorithmic decision-making and a "right to explanation"," *AI Magazine*, vol. 38, no. 3, pp. 50–57, 2017. DOI: 10.1609/aimag.v38i3.2741.

[14] J. Kingston, "Using artificial intelligence to support compliance with the general data protection regulation," *Artificial Intelligence and Law*, vol. 25, no. 4, pp. 429–443, 2017. DOI: 10.1007/s10506-017-9206-9.

[15] G. Stiglic, P. Kocbek, N. Fijacko, M. Zitnik, K. Verbert, and L. Cilar, "Interpretability of machine learning-based prediction models in healthcare," *WIREs Data Mining and Knowledge Discovery*, vol. 10, no. 5, Jun. 2020, ISSN: 1942-4795. DOI: 10.1002/widm.1379. [Online]. Available: http://dx.doi.org/10.1002/widm.1379.

[16] M. Du, N. Liu, and X. Hu, *Techniques for interpretable machine learning*, 2019. arXiv: 1808.00033 [cs.LG].

[17] G. Visani, F. Chesani, E. Bagli, D. Capuzzo, and A. Poluzzi, *Explanations of machine learning predictions: A mandatory step for its application to operational processes*, 2020. arXiv: 2012.15103 [cs.LG].

[18] *Complete or quasi-complete separation in logistic/probit regression*. [Online]. Available: https://stats.idre.ucla.edu/other/mult-pkg/faq/general/faqwhat-is-complete-or-quasi-complete-separation-in-logisticprobit-regression-and-how-do-we-deal-with-them/.

[19] P. McCullagh and J. A. Nelder, *Generalized linear models*. Chapman and Amp; Hall, 1999.

[20] *1.10. decision trees*. [Online]. Available: `https://scikit-learn.org/stable/modules/tree.html`.

[21] T. Laugel, X. Renard, M.-J. Lesot, C. Marsala, and M. Detyniecki, *Defining locality for surrogates in post-hoc interpretablity*, 2018. arXiv: `1806.07498 [cs.LG]`.

[22] R. Guidotti, A. Monreale, S. Ruggieri, D. Pedreschi, F. Turini, and F. Giannotti, *Local rule-based explanations of black box decision systems*, 2018. arXiv: `1805.10820 [cs.AI]`.

[23] D. Alvarez-Melis and T. S. Jaakkola, *On the robustness of interpretability methods*, 2018. arXiv: `1806.08049 [cs.LG]`.

[24] G. Visani, E. Bagli, F. Chesani, A. Poluzzi, and D. Capuzzo, *Statistical stability indices for lime: Obtaining reliable explanations for machine learning models*, Nov. 2020. [Online]. Available: `https://arxiv.org/abs/2001.11757`.

[25] P. Bratley and B. L. Fox, "Algorithm 659: Implementing sobol's quasirandom sequence generator," *ACM Transactions on Mathematical Software*, vol. 14, no. 1, pp. 88–100, 1988. DOI: `10.1145/42288.214372`.

[26] F. Kuo and S. Joe, *Sobol sequence generator: Primitive polynomials and direction numbers*. [Online]. Available: `https://web.maths.unsw.edu.au/~fkuo/sobol/`.

[27] A. Zeileis, T. Hothorn, and K. Hornik, "Model-based recursive partitioning," *Journal of Computational and Graphical Statistics*, vol. 17, no. 2, pp. 492–514, 2008. DOI: `10.1198/106186008x319331`.

[28] Y. Li, S.-L. Hu, J. Wang, and Z.-H. Huang, "An introduction to the computational complexity of matrix multiplication," *Journal of the Operations Research Society of China*, vol. 8, no. 1, pp. 29–43, 2019. DOI: `10.1007/s40305-019-00280-x`.

[29] *Sklearn.model_ selection.gridsearchcv*. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html.

[30] *Scipy.stats.qmc.sobol*. [Online]. Available: https://scipy.github.io/devdocs/reference/generated/scipy.stats.qmc.Sobol.html.

[31] A. B. Owen, *On dropping the first sobol' point*, Dec. 2021. [Online]. Available: https://arxiv.org/abs/2008.08051.

[32] *Sklearn.tree.decisiontreeregressor*. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html.

[33] W. Xiang, "Analysis of the time complexity of quick sort algorithm," in *2011 International Conference on Information Management, Innovation Management and Industrial Engineering*, vol. 1, 2011, pp. 408–410. DOI: 10.1109/ICIII.2011.104.

[34] I. Sobol and B. Shukhman, "Quasi-random points keep their distance," *Mathematics and Computers in Simulation*, vol. 75, no. 3-4, pp. 80–86, 2007. DOI: 10.1016/j.matcom.2006.09.004.

[35] *The boston housing dataset*. [Online]. Available: https://www.cs.toronto.edu/~delve/data/boston/bostonDetail.html.

[36] *Xgboost documentation*. [Online]. Available: https://xgboost.readthedocs.io/en/stable/.

[37] *Sklearn.preprocessing.minmaxscaler*. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html.

[38] *Target encoder*. [Online]. Available: https://contrib.scikit-learn.org/category_encoders/targetencoder.html.

[39] L. C. Thomas, D. B. Edelman, and J. N. Crook, *Credit scoring and its applications*. Society for Industrial and Applied Mathematics, 2002.

[40] *Python api reference*. [Online]. Available: https://xgboost.readthedocs.io/en/stable/python/python_api.html#xgboost.dask.DaskXGBClassifier.predict_proba.

[41] *Sklearn.preprocessing.labelencoder*. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html.

[42] *Sklearn.preprocessing.ordinalencoder*. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OrdinalEncoder.html.

[43] *Sklearn.preprocessing.onehotencoder*. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html.

[44] E. Keogh and A. Mueen, "Curse of dimensionality," in *Encyclopedia of Machine Learning and Data Mining*, C. Sammut and G. I. Webb, Eds. Boston, MA: Springer US, 2017, pp. 314–315, ISBN: 978-1-4899-7687-1. DOI: 10.1007/978-1-4899-7687-1_192. [Online]. Available: https://doi.org/10.1007/978-1-4899-7687-1_192.

[45] *Optimize performance with st.cache*. [Online]. Available: https://docs.streamlit.io/library/advanced-features/caching.

[46] *Streamlit state*. [Online]. Available: https://docs.streamlit.io/library/advanced-features/session-state.