Associazione Italiana Financial Industry Risk Managers

Notes on Explainability and Optimization

Damiano Verda

—

First Edition

Notes on XAI and Optimization

AIFIRM Edizioni - Educational Book Series

ISBN Online: 979-12-80245-39-7

DOI: 10.47473/2016eda0009

Dedication

Albert Einstein once said: «I never think of the future. It comes soon enough». This goes to everyone who is not afraid of the future, and eager to welcome it. Soon enough.

Contents

Dedication **iii**

Introduction **xi**

1 XAI On Structured Data **1**
1.1 Pre-Modeling . . . 2
1.2 In-Modeling . . . 6
1.3 Post-Modeling . . . 21
1.4 Financial Use Case . . . 29

2 XAI On Unstructured Data **33**
2.1 Images . . . 33
2.2 Text . . . 48
2.3 Graphs . . . 57
2.4 Financial Use Case . . . 64

3 Optimization **75**
3.1 Taxonomy . . . 75
3.2 Linear and Non-Linear Optimization . . . 77
3.3 Solving Optimization Problems . . . 85
3.4 Financial Use Case . . . 93

Damiano Verda **95**

Bibliography **97**

List of Figures

1.1 Boolean Lattice diagram. Source: (Muselli et al., 2011). 15
1.2 Sample visualization of LIME output, generated via Python code 23
1.3 Waterfall SHAP plot 25
1.4 Summary SHAP plot 26
1.5 Example of SAFE-AI curves on the HMDA dataset. The figure shows the mean RGA, RGR, and RGE curves computed across the 5-fold evaluation for all models 31

2.1 LIME Image workflow. 34
2.2 How LIME is supposed to work (A), and how it actually works (B) using Monte Carlo sampling for a large enough k. Source: (Rashid et al., 2024) 37
2.3 Binomial (A) and Shapley weight (B) distributions for $k = 10, 20$ and 50. Source: (Rashid et al., 2024) 40
2.4 Dependent variable undersampling (low $RC(Y)$) results in confused explanations (low $CV(\beta)$). Source: (Rashid et al., 2024) 41
2.5 Graph and its adjacency matrix. Source: Noble, J. (2026) *What is a graph neural network.* IBM. Available at: https://www.ibm.com/think/topics/graph-neural-network 57
2.6 GNN Architecture: From input to output. Source: GeeksforGeeks (2025) *What are graph neural networks?* Available at: https://www.geeksforgeeks.org/deep-learning/what-are-graph-neural-networks/ . . 58
2.7 Adjacency matrices constructed according to the proposed approach and Gini importance indices obtained from decision trees trained on synthetic datasets comprising a number of relevant features ranging from 2 to 6, which are predictive of the target class either independently or combined. Source: (Sirocchi et al., 2025). 65
2.8 Local explanation subgraph for a selected high-degree transaction in the Elliptic dataset 74

3.1 Feasible Region 76
3.2 Infeasible Problem 77

3.3 Linear vs Nonlinear Optimization 78
3.4 Convex vs Nonconvex Optimization 79

List of Tables

1.1 Sample dataset, as of September 2024 2
1.2 Toy example for describing the LLM working principle. 11
1.3 Toy example after binarization. Source: (Muselli et al., 2011). 12
1.4 Performance comparison across the HMDA dataset using standard and SAFE-AI metrics across 5 folds. 30

3.1 Solution evolving over different iterations 82
3.2 Hierarchical Tree Clustering — working principle (step 1). Source: Vyas (2020). 91
3.3 Hierarchical Tree Clustering — working principle (step 2). Source: Vyas (2020). 91
3.4 Hierarchical Tree Clustering — working principle (step 3). Source: Vyas (2020). 92

Introduction

This book does not aim to be a comprehensive overview of all possible techniques in the realm of explainability and optimization. It is, conversely, a set of notes oriented towards providing the reader with a compact, yet reasonably flexible set of tools to face a crucial problem: making more effective and better-informed decisions. Together with a subset of standard techniques, these notes also include a few ideas that stemmed from my research in the field.

The first section of the book looks at one aspect of this challenge: making the most of the available data, which are already in tabular form. This means being able to treat them and understand them (*pre-modeling*), derive a model on top of those data understanding its behavior (*in-modeling*) and ask more specific questions, to a model that may have higher performance but a less interpretable behavior (*post-modeling*).

The second section provides some ideas about how the concepts described in the first section can be adapted and applied even if the input data has a more complex, unstructured form. More specifically, three examples are considered: *images*, *text* and *graphs*. As regards the *graphs*, i.e., structures with entities (*nodes*) involved in relations (through connections called *edges*), they are not only considered as possible *input data*, but also as one way to represent correlations among input features (by means of a *feature graph*).

The third section is dedicated to a different aspect of the decision-making process: *optimization*. In fact, there are circumstances in which the aim could be not only *predicting* and *understanding* an unknown variable/phenomenon, but making the most out of our *decision variables*, the variables that we can control. This setting corresponds to an *optimization* problem. We analyze its *taxomony*, the *main algorithms* to deal with (different kinds of) optimization problems and a few *applications*. Among these applications, a wealth management scenario that we use to close the circle: in fact, firstly we *optimize* the allocation of a given capital, and then we apply one of the techniques described in previous sections (*Logic Learning Machine*) to make this decision *explainable*.

1 XAI On Structured Data

In critical contexts such as credit risk, decisions are increasingly supported by artificial intelligence tools. It is not sufficient to know *what* to do, it is also essential to understand *why*. We refer to **explainable** (Explainable AI, XAI) and **interpretable** (Interpretable AI) techniques as those methodologies that provide some form of explanation for the decisions suggested by models. The terms are sometimes used interchangeably, but they actually refer to different concepts.

Explainable AI (XAI)

In the XAI paradigm, a predictive model (possibly a "black box") is first built and then an additional analysis layer is applied to explain its decisions *a posteriori*. The explanation therefore does not coincide with the model producing the output and is, in principle, **model-agnostic**.

Interpretable AI

In the interpretable paradigm, the understanding of the input mapping to output mapping is intrinsic to the model itself (e.g., feature weights or sets of If–Then rules). This makes answering the question "Why was this decision made?" more direct, but restricts the range of usable models.

Generally speaking, the notion of "interpretability" or "explainability" is associated with considerations that involve indeed the choice of the model to extract (we may refer to this as the *in-modeling* phase) or that follow this choice and aim to add an explainability layer on top (let us refer to this phase as *post-modeling*). For the sake of simplicity, we use the acronym XAI whenever we propose considerations that are valid for both approaches (interpretable and explainable AI).

Also, it must be considered that any choice concerning the model or its explanation relies on a good level of understanding of the process that we aim to predict and/or control, as well as of the data representing it. It is indeed important to be able to "explain", at least to some extent, what happens *before* the model is extracted to avoid confusion and misinterpretations of its results. To this end, it is worth starting our

journey in XAI proposing some considerations to be taken into account in what we may call a *pre-modeling* phase.

1.1 Pre-Modeling

We may represent the pre-modeling phase as composed of:

1. **Data understanding**: Verification of what each field represents, the possible presence of a primary key, the operational meaning of the variables, and their relevance to decisions.
2. **Data quality**: checks for completeness and consistency
3. **Feature selection**: removal of non-essential variables to reduce complexity without degrading performance.
4. **Feature engineering**: creation of new informative variables

Name	Surname	Sex	Birth date	Age	Phone	Email	City	Height	Weight	Target
Mario	Rossi	M	5/12/1997	26	3346535632		Genoa	183	84	No
Giovanni	Verdi	M	8/4/2000	24		giovaverdi@mail.it	Savona	178	70	Yes
Marco	Neri	M	12/11/1995	28		nerimarco95@fmail.com	Genoa	189	79	No
Simona	Azzurri	F	4/6/1989	35	3386999789	Sim.azz@industries.com	Genoa	167	58	No
Michele	Arancioni	M	7/1/1999	25	87124	michelearancio@mail.com	Savona	172	72	Yes
Beatrice	Viola	F	30/9/2000	24	3313311786		Savona	175	62	No
Roberta	Gialli	F	3/2/2001	23	3286578908	robi001coldmail.it	Genoa	158	52	Yes
Simone	Blu	M	8/12/1997	26		Simo.blu97@coldmail.it	Genoa	178	76	No
Giulia	Neri	F	24/12/1997	36	3421924568	Giula.neri@webmail.it	Genoa	168	60	No

Table 1.1: Sample dataset, as of September 2024

Data understanding

Let us consider, as an example, the sample dataset represented in Table 1.1, which describes some features of hypothetical customers of a gym, plus the target variable that we would like to predict for future customers, representing whether they subscribed to a given plan or not. Considering that our target variable can assume a finite number of values, we are dealing with a *classification problem*. As this finite set of values is only composed by two possibilities (”Yes” and ”No”) we are dealing with a *binary classification problem*.

Firstly, we observe that some of the features are not only *useless to our goal* but also *dangerous in terms of sensitivity*. Knowing the name and surname of the customers does not ensure any predictive power: any alphanumerical unique ID enabling us to distinguish rows from each other (that is, any *primary key*) would serve the same

purpose as name and surname, with less ambiguity and removing the need of handling personal data.

The same could be said for features like ”Phone” or ”Email”: also in this case, there is no reason to handle these sensitive data, without any perspective of it being predictive of the target that we aim to characterize.

Even with these very straightforward considerations, we reach a first result: we realize that our informative set in input is smaller than it seems at first glance, because at least four columns are not expected to bring any contribution to the solution of the considered problem.

Having a closer look, we can also observe that the age of the customer could be relevant, but the information that the ”Birth date” column adds on top of that is probably not (a few months less or more in the age are not likely to be the deciding factor in subscribing the plan of our interest). In other words, we can reasonably assume that the *Birth date* column is redundant, *for our need*, with respect to the *Age* column.

After this brief schematic analysis, which exemplifies what we mean with **data understanding**, and why it is useful to start from that, let us take a step further and assess **data quality**.

Data quality

The first thing that we can notice, from this point of view, is that some of the data are *missing*, that is the corresponding cells are *blank*. Yet, we can also observe that missing data affect the *phone* and *Email* columns, that is, two columns that we are going to ignore, due to the previous considerations. In this specific example then, we do not need to define a strategy for *missing handling*.

If we needed to, we could consider the following options:

- Ignoring all rows with at least one of the relevant columns hosting a missing value. This is a very solid strategy, but it requires having a significant set of rows with all relevant information to be applied.

- Filling the missing value with the average/median/mode value for the column or with the average/median/mode value of the column when considering only the rows with something in common with the current one. For example: fill in missing heights with the mean male height for men and with the mean female height for women. This solution allows to cope with partially filled rows discarding less information, but this comes at the risk of introducing some noise, as the estimate criterion may be over-simplistic.

- Applying specific techniques and algorithms that end up estimating, for each blank cell, a likely value given the values of the other input variables, especially the most correlated ones. An example of such technique is Multiple Imputation by Chained Equations (MICE) (Azur et al., 2011). Code packages featuring an implementation of the MICE algorithm are available on Github [1].

Other interesting data quality checks may be applied to *comply with the expected data format* (if any). For example, we notice that one of the phone numbers (row 5) is composed by fewer digits than it should, and that one of the e-mail addresses (row 7) does not contain an "@" character. These data are clearly wrong, considering what they should represent and their *coherence with respect to other values in the same column*.

Extending this logic, it is also important to consider *coherence with respect to values in the same rows, in other columns*, as long as these other columns host information that is clearly correlated. For example, considering *birth date* and *age* we can infer that the data collection is likely to have happened in 2024...and that either the age or the birthday value of the last row is wrong! In fact, if the birth date happened in 1997 age cannot be 36. In this case, we may even propose an educated guess and assume that the "true" value for *age* for the last row is 26, and not 36 (but double-checking is, of course, more than recommended). Similarly, we could validate the coherence among the *age/birth date*, *height* , and *weight* columns.

Feature selection

Even if obviously not irrelevant from our predictive perspective as *name* or *surname*, some other fields may also not be essential for the predictive task at hand: if so, removing them reduces computational complexity and could reduce the risk of *overfitting*, that is, pushing the model to learn not only the main patterns in data, but also noise.

A detailed analysis of feature selection techniques would be out of the scope of this book, but let us mention the three main approaches to which feature selection techniques belong.

- **filter methods** are based on a uni-variate correlation metric between each input variable and the output, defining minimum correlation thresholds for an input variable to be included in the model. They are usually lightweight and relatively easy to handle, but are not equipped to consider correlations between *groups of input variables* and the output.

[1] https://github.com/amices/mice

- **wrapper methods** address the limitation of **filter methods** by introducing an auxiliary model dedicated to feature selection. In extreme synthesis, the criterion for feature selection corresponds in fact to optimizing the performance of this auxiliary model. Even when picking a light model, such as a *linear/logistic regression*, picking a **wrapper method** implies a higher computational cost with respect to a **filter method**.
- **embedded methods** represent the fact that some models *incorporate feature selection* as one of the steps / side-products of their modeling process. Among the (interpretable) models that are addressed in the following sections, this is the case both for *classification and regression trees* and for the *Logic Learning Machine*.

Feature engineering: BMI example

Another layer of what we may call *pre-modeling* explainability consists is blending and engineering the information which is contained in separate columns into a single, additional, and more informative feature.

For example, even among the small set of data available in Table 1.1, it is possible, given weight (kg) and height (cm), to define the Body Mass Index (*BMI*), which allows us to determine whether the individual is underweight/overweight (while height and weight alone are less informative) as:

$$\text{BMI} = \frac{\text{weight (kg)}}{\left(\frac{\text{height (cm)}}{100}\right)^2}. \tag{1.1}$$

In this example, such a transformation enriches anthropometric information beyond raw measurements, potentially improving separability, and hence predictive power, with respect to our target variable.

1.2 In-Modeling

Moving to the in-modeling phase, let us describe in this section three different modeling techniques that ensure a good level of *interpretability*. Firstly, *linear/logistic regression*, which models the underlying process by identifying an optimal "weight" to assign to each of the input variables, thus highlighting their importances.

Secondly, *classification and regression trees*, which ensure an even stronger interpretability, justifying each prediction through the application of an IF-THEN rule extracted from the input dataset, could exhibit some quantitative performance problem, especially when dealing with an imbalanced problem i.e, a problem for which one (or more) of the output classes is represented by significantly fewer samples with respect to the other ones.

Finally, also due to the above considerations, we propose the *Logic Learning Machine (LLM)*, which is still a rule-based model, ensuring a level of interpretability comparable to *classification and regression trees*, but aims to be more robust in terms of quantiative performance through the extraction of (potentially) overlapping rules, handling their conflicts through an algorithm named *Standard Applied Procedure*. *Linear/logistic regression* and *classification and regression trees* are standard and well-established models, so we are simply summarizing their main characteristics, referring to scientific literature for further details.

The *Logic Learning Machine* is a more innovative and less widespread model, so we describe its modeling and model application procedure in greater detail.

Linear/Logistic Regression

Given a data set $\{y_i, x_{i1}, ..., x_{ip}\}$ with $i = 1, ..., N$ of statistical units N, a linear regression model assumes that the relationship between the dependent variable y and the vector of regressors $\mathbf{x}$ is linear. This relationship is modeled through a disturbance term ϵ - an unobserved random variable that adds noise to the linear relationship between the dependent variable and regressors (Stock et al., 2011). Thus, the model takes the following form:

$$y_i = \beta_0 + \beta_1 x_{i1} + ... + \beta_p x_{ip} + \epsilon_i = \boldsymbol{x}_i^\intercal \boldsymbol{\beta} + \epsilon_i \tag{1.2}$$

where $\intercal$ denotes the transpose operator, so that $\boldsymbol{x}_i^\intercal \boldsymbol{\beta}$ is the inner product between the vectors $\boldsymbol{x}_i$ and $\boldsymbol{\beta}$. The linear relationship can be rewritten in matrix form: $\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$ where:

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} \mathbf{x}_1^\top \\ \mathbf{x}_2^\top \\ \vdots \\ \mathbf{x}_n^\top \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & \cdots & x_{1p} \\ 1 & x_{21} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \cdots & x_{np} \end{bmatrix}, \quad \boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_p \end{bmatrix}, \quad \text{and} \quad \boldsymbol{\varepsilon} = \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{bmatrix}. \tag{1.3}$$

In order to estimate $\boldsymbol{\beta}$ we can use the traditional econometric formula for an ordinary least squares (OLS) problem: $\hat{\boldsymbol{\beta}} = (\boldsymbol{X}^\intercal \boldsymbol{X})^{-1} \boldsymbol{X}^\intercal \boldsymbol{y}$

In case of a classification problem, the above formulation is adapted towards estimating not a real number, but a likelihood of a given output to be the correct one. Through a binarization step, this estimated likelihood is turned into a categorical output value: this is called a *logistic regression model*.

After fitting a **linear/logistic regression model**, we often wonder whether any of the input variables are *statistically significant* with respect to the output we aim to predict. The *overall F-test* answers this question. The *overall F statistic* is computed as the variance explained by the model, divided by the variance not explained by the model. This ratio is expected to follow a *Fisher–Snedecor distribution* (Yoo et al., 2017). The probability that none of the input variables is statistically significant is the p-value associated with the overall F statistic, which can be obtained from Fisher–Snedecor distribution tables or via libraries such as *scipy* in Python. Once we have verified that *at least one* input variable is statistically significant in predicting the output, we may want to determine whether *each variable* is significant when considered individually. To achieve this, we perform a *partial F* test.

Specifically, the difference between the *variance explained by the full model* (including the input variable under consideration) and the *variance explained by the reduced model* (excluding that variable) is divided by the variance not explained by the full model. This ratio is also expected to follow a *Fisher–Snedecor distribution*. The corresponding probability can again be gathered from well-known tables or code packages[2].

Another more advanced (and more computationally intensive) diagnostic technique based on a multivariate linear model is the *DFBeta analysis*. Here, alongside the "base" regression model, we compute N **additional regression models** (where N is the number of training samples), each time excluding a single sample from the training set. For each additional model, we compute the difference between the coefficients $\beta_0, \beta_1, \ldots, \beta_k$ and those of the base model. If the absolute value of such a difference,

[2]See, for example: https://docs.scipy.org/doc/scipy/reference/generated/scipy.special.fdtr.html

for one or more coefficients, **exceeds** $\frac{2}{\sqrt{N}}$ (common rule of thumb, with N being the number of available samples), this indicates an anomalous impact of the corresponding sample during training. This information can guide subsequent **outlier detection** analyses.

Classification and Regression Trees

Classification and Regression Trees (CART) are popular estimation methods in Machine Learning, suitable for addressing both classification and regression tasks. These models approximate data using a stepwise function, built by repeatedly splitting the observation space based on threshold values of the input variables. At each split, the algorithm identifies regions of data where the target variable values are relatively homogeneous. In regression tasks, the prediction is typically the mean value within the region, while in classification, it corresponds to the most frequent class. This technique is often referred to as a "tree" due to the visual representation of its decision-making process.

The partitioning of the data space by the model can be depicted as a binary tree, where each element, called a node, contains a threshold that guides the division of observations. The initial node is known as the *root*, and the final partitions, which represent specific data subsets, are called *leaves*.

To build such a model, a training dataset with known target values is required. The CART algorithm constructs the tree through a sequential process of binary splits. Initially, for every input variable, potential threshold values are evaluated to find the one that best separates the data. The goal is to minimize within-group variance while maximizing the difference between the resulting subsets. These constraints are incorporated into the objective function formula:

$$D = \sum_{i=1}^{K} \left[\sum_{j \in S_i} \left(y_i - \hat{\beta}_i \right)^2 \right] = \sum_{i=1}^{K} D_i \tag{1.4}$$

where y_i and $\hat{\beta}_i$ are, respectively, the values of the target variable and the associated parameter found in one of the data subspaces K_i, $i = 1, ..., K$ S_i. Consequently, the minimization problem can be expressed as $\min \sum_{i=1}^{K} D_i \left(\hat{\beta}_i \right)$.

From this point onward, the same procedure is applied to each of the newly formed subsets: the algorithm scans the data within each subset to identify the variable and threshold that produce the best possible split. This process is repeated iteratively until

specific stopping criteria are met. These criteria may include, for instance, the formation of a region where all data points belong to the same class (in Classification Trees) or share the same value (in Regression Trees); the point at which further splits fail to significantly improve the objective function; or when a resulting subset contains only a single observation.

By the end of this process, the algorithm generates a highly detailed and intricate tree structure, characterized by numerous *branches* and *leaves*. However, such a tree is often difficult to interpret due to its complexity and the large number of splits. In addition, it could overfit the training data, an issue that compromises the model's ability to generalize well to new, unseen data.

To address this problem, a simplification step known as pruning is employed. This automated technique systematically reduces the fully grown tree by eliminating branches that contribute little to the accuracy of the prediction or carry limited information value, improving the clarity and robustness of the model. Defining the loss function as follows:

$$C_\alpha(K) = \sum_{i=1}^{K} D_i + \alpha K \tag{1.5}$$

where K is the size of the tree considered at each step and α is the parameter associated with the computational cost of the model, at each step, the leaf whose removal results in the smallest increase in the objective function is removed. The procedure continues until the value of $C_\alpha(K)$ converges to an optimal value. A pseudocode of the CART algorithm is provided by (Loh, 2011).

Logic Learning Machine

The Logic Learning Machine is a rule-based method alternative to decision trees ((J. Ross Quinlan, 1986) (J Ross Quinlan, 2014)). In short, the LLM transforms the data into a Boolean domain where some Boolean functions (one for each output value) are reconstructed starting from a portion of their truth table with a method described in the paper (Muselli et al., 2011).

When applied to a regression problem, the algorithm iteratively selects a pattern from the training set and defines a subgroup by including other instances that are sufficiently similar, based on a user-defined threshold known as the *Maximum Desired Dispersion Coefficient*. This coefficient controls the allowable variability within the subgroup: a smaller value results in tighter, more homogeneous clusters, while a larger value permits the inclusion of more diverse patterns. Once a subgroup is defined, the

task is reformulated as a (set of) binary classification problem(s) distinguishing between the selected subgroup, and the remaining data and classification rules are extracted accordingly. The output associated with each rule is computed as the median of the target values of all covered points, including both the original subgroup and any additional instances within the tolerated classification error.

Each of the classification problems derived from the original regression is solved by the algorithm creating a set of intelligible rules through Boolean function synthesis. Then, the LLM algorithm evaluates their quality, applies them, and, through the rules, derives a feature ranking ((Ferrari et al., 2023) also show how the same rules can be used to suggest control actions, to drive the output towards a desired value). This happens following five steps:

1. Discretization
2. Latticization or Binarization
3. Positive Boolean function
4. Rule generation
5. Rule Quality and Standard Applied Procedure

In a classification problem d-dimensional examples $x \in X \subset \Re^d$ are to be assigned to one of q possible classes, labeled by the values of a categorical output y. Starting from a training set S including N pairs (x_i, y_i), with $i = 1, \ldots, N$, deriving from previous observation, techniques for solving classification problems have the aim of generating a model $g(x)$, called classifier, that provides the correct answer $y = g(x)$, for most input patterns x. To analyse the process, a bi-class toy problem is used, whose training set is demonstrated in Table 1.2.

Discretization

In this step, a mapping converts each continuous variable domain into a discrete domain. $\psi_j \; X : X_j \longrightarrow I_M$, where X_j is the domain of the j-th variable and $I_M = 1, \ldots, M$ is the set of positive integers up to M. The mapping must preserve the ordering of the data. If $x_{ij} \leq x_{kj}$, then $\psi_j(x_i) \leq \psi_j(x_k)$, $\forall\, j = 1, \ldots, d$. One way

X_1	X_2	Y
0.07	Red	O_0
0.11	Red	O_0
0.22	Black	O_0
0.14	Red	O_0
0.23	Green	O_0
0.08	Black	O_1
0.12	Green	O_1
0.21	Red	O_1
0.26	Red	O_1
v 0.24	Blue	O_1

Table 1.2: Toy example for describing the LLM working principle.

to describe ψ_j is that it consists of a vector $\gamma_j = (\gamma_{j1}, \ldots, \gamma_{jm}, \ldots, \gamma_{M_j-1})$ such that:

$$\psi_j(x_i) = \begin{cases} 1 & x_{ij} \leq \gamma_{j1} \\ m & \gamma_{jm-1} < x_{ij} \leq \gamma_{jm} \\ M_j & x_{ij} > \gamma_{jM_j-1} \end{cases} \tag{1.6}$$

One of the supported strategies for discretization consists in creating M_j intervals having the same length. Let ρ_j be the vector of all the α_j values for input variable j in ascending order ($p_{jl} < p_{jl+1},\ \forall\, l = 1, \ldots, \alpha_j$), then the cutoff γ_{jm} is given by:

$$\gamma_{jm} = p_{j1} + \frac{p_{j\alpha_j} - p_{j1}}{M_j} m \tag{1.7}$$

This method is referred to as *Equal Width discretization.*

Binarization

In this step, each discretized domain is transformed into a binary domain through a mapping $\varphi_j : I_{M_j} \longrightarrow \{0,1\}^{M_j}$, where I_{M_j} is the domain of the j-th variable and $\{0,1\}^{M_j}$ is a string having a bit for each possible value in I_{M_j}. The mapping must maintain the ordering of data: $u < v$ if and only if $\varphi_j(u) < \varphi_j(v)$, where the standard

z	Y
01 0111	O_0
01 0111	O_0
10 1101	O_0
01 0111	O_0
10 1110	O_0
01 1101	O_1
01 1110	O_1
10 0111	O_1
10 0111	O_1
10 1011	O_1

Table 1.3: Toy example after binarization. Source: (Muselli et al., 2011).

ordering between z and $w \in \{0,1\}^{M_j}$ is defined as follows:

$$z < w \quad \text{if and only if} \quad \begin{cases} \exists i \quad \text{such that} \quad z_i < w_i \\ \forall\, l < i \qquad\qquad\quad z_l \leq w_l \end{cases} \qquad (1.8)$$
$$z \leq w \quad \text{if and only if} \quad z_i \leq w_i \quad \forall\, i = 1, \ldots, M_j$$

If the relation in the equation above holds, then it is said that z covers w. A suitable choice for φ_j is the inverse only-one coding, that for each $k \in I_{M_j}$ creates a string $h \in \{0,1\}^{M_j}$ having all bits equal to 1 except the k-th bit which is set to 0. For example, let $x_{ij} = 3$ with domain I_5, then $\varphi_j(x_i) = 11011$. In this way $\varphi_j(x_i) = z_i$, where z_i is obtained by concatenating $\varphi_j(x_i)$ for $j = 1, \ldots, d$. As a result, the new training set is $S^{'} = \{(z_i, y_i)\}_{i=1}^{N}$, with $z_i \in \{0,1\}^B$, where $B = \sum_{j=1}^{d} M_j$. The training set obtained by applying discretization with the single cutoff 0.15 for the variable X_1 and subsequent binarization for the toy problem is shown in Table 1.3.

Synthesis of the Boolean function

The training set $S^{'}$, obtained after binarization, can be divided into two different subsets according to the output class: T is the set containing (z_i, y_i) with $y_i = O_1$, whereas F is the set containing the example for which $y_i = O_0$. T and F can be viewed as a portion of the truth table of a Boolean function f that must be reconstructed. Before proceeding with the method description, it is useful to give some

definitions and notations.

- Each Boolean function can be written with operators *AND*, *OR*, and *NOT* that constitute the Boolean algebra; if *NOT* is not considered then a simpler structure, called Boolean lattice, is obtained. From now on, only the Boolean lattice is considered. It can be drawn by positioning z over w if $z > w$ and by linking all the pairs z, w for which an a does not exist such that $w < a < z$. An example for $\{0,1\}^3$ is shown in Figure 1.1.

- The sum (OR) and product (AND) of η terms can be denoted as follows:

$$\begin{aligned} \vee_{j=1}^{\eta} z_j &= z_1 + z_2 + \ldots + z_\eta \\ \wedge_{j=1}^{\eta} z_j &= z_1 \cdot z_2 \cdot \ldots \cdot z_\eta = z_1 z_2 \ldots z_\eta \end{aligned} \tag{1.9}$$

- A logical product is called an implicant of a function f if the following relation holds: $\wedge_{j=1}^{\eta} z_j \leq f$, where each element z_j is called literal. The product is called prime implicant if the relation no longer holds when a literal is removed from the implicant.

- The ordering in a Boolean lattice is defined by the equations above. According to this ordering, a Boolean function $f : \{0,1\}^B \longrightarrow \{0,1\}$ is called positive if $z \leq w$ implies $f(z) \leq f(w)$ for each $z, w \in \{0,1\}^B$.

- A subset $A \subset I_B$ such that for each element $z, w \in A$, an ordering cannot be established (neither $z < w$, nor $w < z$), is called antichain.

- Given $a \in \{0,1\}^B$, then the set $L(a) = \{z \in \{0,1\}^B \mid z \leq a\}$ is called a lower shadow of a, whereas the set $U(a) = \{z \in \{0,1\}^B \mid z \geq a\}$ is called an upper shadow of a. The lower and upper shadows for $101 \in \{0,1\}^3$ are shown in Figure 1.1.

- Given the subset $T, F \in \{0,1\}^B$, then T is lower separated from F, if there is no element $z \in T$ belonging to the lower shadow of some element of F.

- Given the binary string a, if there is a $z \in T$ such that $a \leq z$, there is not a $w \in F$ such that $a \leq w$, and for each $b < a$, there is $w \in F$ such that $b \leq w$, then a is called bottom point for the pair (T, F).

- Every positive Boolean function can be written in its unique, not redundant Positive Disjunctive Normal Form (PDNF) (Aizenstein et al., 1995) as the sum of its

prime implicants: $f(z) = \vee_{a \in A} \wedge_{j \in P(a)} z_j$, where $P(a)$ is the subset I_B containing each i such that $a_i = 1$; A is an antichain of $\{0,1\}^B$ and each a is called the minimum true point. For example, the not redundant PDNF $f(z) = z_1 z_3 + z_4$ is obtained from antichain $A = \{1010, 0001\}$.

From these definitions, it follows that a method for finding f must retrieve the set of minimum true points to be used from T and F, to represent f in its irredundant PDNF. It follows that the set of all bottom points for (T, F) is an antichain, which elements are candidate minimum true points.

The LLM firstly identifies implicants, and then turns them into rules. The algorithm adopted by the LLM to produce implicants is called *Shadow Clustering* (Muselli et al., 2011). It generates implicants for f by analysing the Boolean lattice $\{0,1\}^B$. The algorithm selects a node in the diagram and generates bottom points (T, F) by descending the diagram: moving down from a node to another node is equivalent to changing a component from 1 to 0 and a bottom point is added to A when any further move down leads to a node belonging to the lower shadow of some $w \in F$. In particular, the starting node is chosen between the $z \in T \subset \{0,1\}^B$ that do not cover any point $a \in A$ such that $a \leq z$ (in other words the algorithm ends when each element in T covers at least one element in A). Once A has been found, it may contain redundant elements and consequently, it must be simplified to find A^*, from which the PDNF of the positive Boolean function f can be derived.

Different versions of Shadow Clustering exist depending on the choice of the element to be switched from 1 to 0 at each step of the diagram descent. For example, *Maximum-covering Shadow Clustering* (MSC) at each step changes the i-th element that maximises the associated potential covering, defined as the number of elements $z \in T$ for which $z_i = 0$. As concerns the selection of $A^* \subset A$, a possible choice is to subsequently add to A^* the element of A that covers the highest number of points in T that are not covered by any other element of A^*. The application of the Shadow Clustering algorithm [Algorithm 1] to the dataset after binarization shown in Table [1.3] produces the implicants:

- 100011 which corresponds to $z_1 \wedge z_5 \wedge z_6$
- 011100 which corresponds to $z_2 \wedge z_3 \wedge z_4$

Then, the PDNF of the resulting Boolean function is the following: $f(z) = (z_1 \wedge z_5 \wedge z_6) \vee (z_2 \wedge z_3 \wedge z_4)$

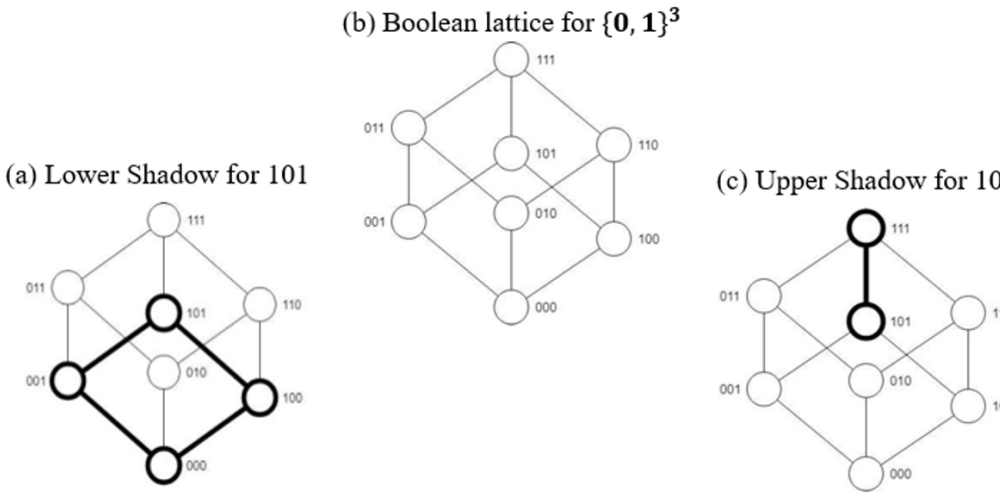


Figure 1.1: Boolean Lattice diagram. Source: (Muselli et al., 2011).

Rule generation

In the last step, each implicant of the positive Boolean function f is transformed into an intelligible rule, where, as said before, a function is generated for each output value. Then the consequence of the rules only depends on f. The transformation takes into account the coding applied during binarisation. In particular, z was obtained by concatenating the results of the mapping $\varphi_j(x)$ for each $j = 1, \ldots, d$ and consequently it can be split into substring h_j for each attribute, whose bit $z_i \in h_j$ corresponds to a nominal value if X_j is nominal. In contrast, it corresponds to an interval if X_j is ordered. For each implicant, a rule in *IF-THEN* form is generated by adding a condition for each attribute X_j as follows:

- If $z_i = 0$ for each $z_i \in h_j$, then no condition relative to X_j is added to the rule;
- If X_j is nominal, then a condition $X_j \in V$ is added to the rule, where V is the set of values associated with each $z_i \in h_j$ such that $z_i = 0$;
- If X_j is ordered, then a condition $X_j \in V$ is added to the rule, where V is the union of the intervals associated with each $z_i \in h_j$ such that $z_i = 0$.

Algorithm 1: Shadow Clustering algorithm (bottom-up)

Data: $P(x)$
$I = P(x)$
$A = \emptyset$
while $I \neq \emptyset$ **do**
  choose $i \in I$ and remove it from I
  if *there is* $y \in F$ *such that* $p(I \cup A) \leq y$ **then**
    add i to A
return $p(A)$

Each rule r_i (representing the i-th rule) can be generally formalised as:

$$r_i : \text{IF} \bigwedge_{j=1}^{J_i} X_j \in V_i \text{ THEN } O_i \tag{1.10}$$

where $\bigwedge$ denotes logical AND, $X_j \in V_i$ represents the condition of the feature X_j that corresponds to a value (or range of values) of rule r_i, O_i denotes the output class associated with rule r_i, and J_i represents the number of premises (features) in rule r_i.

For the implicant 100011 obtained in the previous step, $h_1 = 10$ leads to the condition $X_1 \in (0.17, \infty)$ or $X_1 > 0.17$ and $h_2 = 0011$ leads to the condition $X_2 \in \{Red, Blue\}$. Then the rule relative to 100011 is:

$$\text{IF } X_1 > 0.17 \text{ AND } X_2 \in \{Red, Blue\} \text{ THEN } Y = O_1$$

For the implicant 011100 obtained in the step described previously, $h_1 = 01$ leads to the condition $X_1 \in (-\infty, 0.17]$ or $X_1 \leq 0.17$ and $h_2 = 1100$ leads to the condition $X_2 \in \{Black, Green\}$. Then the rule relative to 011100 is:

$$\text{IF } X_1 \leq 0.17 \text{ AND } X_2 \in \{Black, Green\} \text{ THEN } Y = O_1$$

Please note that if X_j is ordered, conventionally the upper bound of the interval, if finite, is always included in the condition, whereas the lower bound is excluded. To generate the rule for the other class, it is sufficient to label O_0 with 1 and O_1 with 0. In the case of the multiclass problem, it is enough to decompose the problem into several bi-class issues: for each sub-problem, the target class is labelled with 1, and all the remaining classes with 0.

Rule quality and class prediction

The process described in the previous subsection explains that each element x_i of the training set only satisfies rules associated with the output class of x_i, but since data are affected by noise, usually it is preferable to admit some errors in order to enable the model to generalize. In order to permit a fraction of error, the descent of the diagram does not stop when a further move down leads to the lower shadow of some $w \in F$, but it continues until a further move leads to a node belonging to the lower shadow of a percentage element $w \in F$ greater than a regularization parameter ε_{max}. Then, it is usual that an element of the training set covers the rule of different classes. When it happens, the output class is established according to the relevance of the rules satisfied by it. In order to present relevance, the following quantities relative to a rule r in the *IF <premise> THEN <consequence>* form are introduced:

- $TP(r)$ is the number of training set examples that satisfy both the premise and the consequence of the rule r,
- $FP(r)$ is the number of training set examples that satisfy the premise but do not satisfy the consequence of the rule r,
- $TN(r)$ is the number of training set examples that do not satisfy either the premise or the consequence of the rule r,
- $FN(r)$ is the number of training set examples that do not satisfy the premise and satisfy the consequence of the rule r.

Please note that an example x_i satisfies the premise of the rule r if it satisfies all its premise conditions, whereas x_i does not satisfy the premise of the rule r if it does not satisfy at least one among its premise conditions. Combining these quantities, quality measures for a rule r can be computed:

$$\text{Covering:} \quad C(r) = \frac{TP(r)}{TP(r) + FN(r)} \tag{1.11}$$

$$\text{Error:} \quad E(r) = \frac{FP(r)}{TN(r) + FP(r)} \tag{1.12}$$

It is evident that the greater the covering, the more relevant the rule is; on the other hand, the smaller the error, the less relevant the rule is. Then, the relevance of a rule r is obtained by combining $C(r)$ and $E(r)$:

$$R(r) = C(r)(1 - E(r))$$

Once the relevance of the rule is defined, it is possible to use it in order to compute a score $S(x_i, o)$ for each class o that measures how likely it is that $y_i = o$:

$$S(x_i, o) = \sum_{r \in \mathfrak{R}_o^i} R(r) \tag{1.13}$$

with $\mathfrak{R}_o^i = \{r \mid r \in \mathfrak{R},\, r \leq x_i,\, O(r) = 0\}$, where $\mathfrak{R}$ is the complete ruleset, $r \leq x_i$ denotes that x_i satisfies the premise of the rule.

On the other hand, to obtain a measure of relevance $R(c)$ for a condition c included in the premise part of a rule r, the rule $r^{'}$ can be considered obtained by removing that condition from r. Since the premise part of $r^{'}$ is less stringent, we obtain that $E(r^{'}) \geq E(r)$ so that the quantity $R(c) = (E(r^{'}) - E(r))C(r)$ can be used as a measure of relevance for the condition c of interest. $O(r) = o$ denotes the consequence of r predict class o. Then $\mathfrak{R}_o^i$ is the set of rules satisfied by x_i that predict class o. From the scores of each output class, it is possible to define the probability that $y_i = o$:

$$P(o \mid x_i) = \frac{S(x_i, o)}{\sum_{k \in O} S(x_i, k)} \tag{1.14}$$

Then the selected output is the one that maximizes the output probability:

$$\tilde{y}_i = \underset{o}{max} P(o \mid x_i)$$

Feature ranking

For every ordered variable $x_j \in X$ let us denote with $M_j - 1$ the collection of all the thresholds γ_{jl} involved in the conditions of rules r_k; through these thresholds the domain of the component x_j is subdivided into M_j adjacent intervals $[-\infty, \gamma_{j1}]$, $(\gamma_{j1}, \gamma_{j2}], \ldots, (\gamma_{j,l-1}, \gamma_{ji}], \ldots, (\gamma_{jM_{j-1}}, +\infty]$. Let us denote with $J_{j1}, J_{j2}, \ldots, J_{M_j}$ these intervals, so that $J_{j1} = [-\infty, \gamma_{j1}], J_{j2} = (\gamma_{j1}, \gamma_{j2}]$, etc.

Now, if a rule $r_k \in \mathfrak{R}_o = \{r \mid r \in \mathfrak{R}, O(r) = 0\}$ for the output class o (i.e. whose consequence part is $y = o$) includes a condition c_{kl}, with relevance $R(c_{kl})$, involving the ordered component x_j, the points of m_{kl} of the M_j adjacent intervals verify that condition. For instance, if the condition c_{kl} is $x_j \leq \gamma_{j3}$, the points of the $m_{kl} = 3$ intervals J_{j1}, J_{j2} and J_{j3} satisfy c_{kl}. It is then possible to retrieve a measure of relevance $R_k^o(J_{ji})$ for each interval J_{ji}, with respect to the output class o, by looking at the quantities $R(c_{kl})$ of the conditions c_{kl}, that are included in rules r_k, involve the component x_j, and are verified by points of J_{ji}. In particular, if a condition c_{kl}

involving x_j is satisfied by m_{ki} of the M_j adjacent intervals, the relevance quantity that can be attributed to each of these intervals is:

$$R_k^o(J_{ji}) = \frac{R(c_{kl})}{m_{kl}}$$

By collecting all the relevancies derived from all the rules $r_k \in \mathfrak{R}_o$ including a condition c_{kl} on the component x_j, we can obtain the measure of relevance $R^h(J_{ji})$ of the interval J_{ji} with respect to the output class o:

$$R^o(J_{ji}) = 1 - \prod_{r_{k \in \mathfrak{R}_o}} (1 - R_k^o(J_{ji})) \tag{1.15}$$

Starting from 1.15 a measure of relevance $R^o(x_j)$ for the component x_j (with respect to o) can be derived by considering the variation of $R^o(J_{ji})$ over the M_j adjacent intervals $J_{j1}, J_{j2}, \ldots, J_{M_j}$. In fact, if $R^o(J_{ji})$ does not change so much in these intervals, then different thresholds are essentially used to determine parts of the input domain where the behavior of the model $g(x)$ is similar. This means that the variable x_j has little discriminant power among different classes, but simply characterizes the input domain with respect to $g(x)$ for the output class o.

A possible way of measuring the variation of a quantity is to consider its standard deviation σ; with this choice we have:

$$R^o(x_j) = M_j \sigma_j \left(R^o(J_{ji})\right) \tag{1.16}$$

where σ_j stands for the standard deviation over the M_j intervals $J_{j1}, J_{j2}, \ldots, J_{M_j}$.

A sign for $R^o(x_j)$, which indicates if the variable x_j is directly (if the sign is positive) or inversely (if the sign is negative) correlated with the output class o, can also be retrieved by looking where higher values of $R^o(J_{ji})$ are located. In particular, if higher values of $R^o(J_{ji})$ occur at higher (resp. lower) i then the variable x_j is directly (resp. inversely) correlated with the output class o.

Hence, a procedure for deriving the sign of $R^o(x_j)$ consists in subdividing the product of 1.2 in two parts: the first one, denoted with $R^{o-}(J_{ji})$, contains terms $R_k(J_{ji})$ originated by conditions c_{kl} of the form $x_j \leq \gamma_{ji}$, whereas $R^{o+}(J_{ji})$ includes terms $R_k(J_{ji})$ derived by conditions c_{kl} of the kind $x_j > \gamma_{ji}$. As for conditions c_{kl} of the form $\gamma_{ji_1} < x_j \leq \gamma_{ji_2}$ terms $R_k(J_{ji})$ for $i \leq (i_1 + i_2)2$ (resp. $i > (i_1 + i_2)2$) are inserted into $R^{o-}(J_{ji})$ (resp. $R^{o+}(J_{ji})$). With these definitions the sign of $R^o(x_j)$ becomes negative if $R^{o-}(J_{ji}) < R^{o+}(J_{ji})$ and positive in the opposite case.

If the variable x_j is nominal, then equation 1.2 can still be used to determine mea-

sures of relevance $R^{o-}(J_{ji})$ if $G_j = \{v_{j1}, v_{j2}, \dots\}$ is the collection of the possible values assumed by x_j and $J_{ji} = \{v_{ji}\}$, for $i = 1, 2, \dots, |G_j|$. In this case equation 1.15 becomes:

$$R^o(x_j) = |G_j| \sigma_j \left(R^o(J_{ji})\right) \tag{1.17}$$

A sign for $R^o(x_j)$ cannot be determined and it is therefore always considered as positive.

If the (absolute) maximum on the q output classes of the quantities $R^o(x_j)$ is greater than 1, then all relevances $R^o(x_j)$ are normalized to this maximum so that their values lie in the range [0,1]. By averaging the quantities $R^o(J_{ji})$ and $R^o(x_j)$, for $o = 1, \dots, q$, we can obtain absolute measures of relevance $R(J_{ji})$ and $R(x_j)$ (independent of the output class o) for J_{ji} and for the variable x_j:

$$R(J_{ji}) = \frac{1}{q} \sum_{o=1}^{q} R^o(J_{ji}) \quad , \quad R(x_j) = \frac{1}{q} \sum_{o=1}^{q} R^o(x_j)$$

1.3 Post-Modeling

Explainable Artificial Intelligence (XAI) aims to make Machine Learning models—especially black-box models—more transparent by providing human-understandable justifications for their output. XAI techniques address this by approximating the model's decision boundaries, highlighting influential regions, or assigning attribution scores to input features.

It is worth considering that, once derived, the quality of different post-hoc explanations can also be quantitatevely compared, by means of metrics such as the ones introduced in (Calzarossa et al., 2025b) (Calzarossa et al., 2025a). More specifically, the approach proposed to measure, in a transparent way, the degree of robustness of the explanations, is applied to ensemble tree models. More specifically, the Gini impurity is used to derive feature importance plots, to explain the output of tree models and of ensembles of tree models: a feature importance plot is built using the mean decrease in impurity, computed as the normalized reduction of the Gini impurity. Optimizing with respect to this metric follows, in the proposal, the *parsimony principle*, that is it assumes that the lower the number of relevant predictors, the better. In other words, it is suggested to tune the parameters of a model so that the Gini value is as small as possible, to maximize concentration of the explanations.

Two of the most well-known techniques of this kind are LIME and SHAP, whose characteristics are described in the following part of this section, in which we also propose a small application in Python on a loan approval sample dataset.

Perturbation-based Methods - LIME

Perturbation-based methods explain predictions by analyzing the change in model output in response to changes in the input. One of the most influential methods is LIME (Local Interpretable Model-agnostic Explanations), introduced by Ribeiro et al. (Ribeiro et al., 2016a). LIME generates a local interpretable approximation around an input ξ by perturbing it and fitting a sparse linear surrogate model:

$$\hat{f}(x) = \arg\min_{g \in G} \mathcal{L}(f, g, \pi_x) + \Omega(g) \tag{1.18}$$

Here, f is the black-box model, G is the class of interpretable models (e.g. linear models), π_x is a proximity-based kernel defining the locality around x, and $\Omega(g)$ penalizes complexity.

Let us apply this logic to one example: a dataset consisting of 500 loan applications, each described by features such as Gender, Married, Dependents, Education, Self-

Employed, ApplicantIncome, CoapplicantIncome, LoanAmount, LoanAmountTerm, CreditHistory, and PropertyArea. The target variable is LoanStatus, which indicates whether or not a loan was approved.

A pre-trained classifier (more specifically, a model composed by a set of classification trees, named *random forest*: each tree alone would be interpretable, but the interaction among them makes the prediction harder to interpret) is loaded to predict the likelihood of loan approval. The primary goal is to provide interpretability for the predictions through LIME, as the multiplication of decision trees reduces the interpretability of the base model. This is particularly important in financial applications, where understanding why a loan was approved or denied for a specific applicant can inform a more auditable decision-making.

```
# Basic libraries
import pandas as pd
import numpy as np

# Library to load models
import joblib

# Library to visualize results
from IPython.display import HTML

# LIME library
import lime
from lime import lime_tabular

# Load Random Forest model and its dataset
model = joblib.load(r"data/loan_prediction_model.pkl")
data = pd.read_csv("data/df1_loan.csv")
```

LimeTabularExplainer initializes the LIME explainer for tabular data, creating a local model to approximate the behavior of the classifier. The explainer needs the training data to understand the feature space and generate perturbed samples and the list of features to make the explanation humanly readable.

```
# Initialize the LIME explainer
explainer = lime_tabular.LimeTabularExplainer(
    training_data=np.array(X_train_lime),
    feature_names=X_train_lime.columns.tolist(),
    class_names=['Not Approved', 'Approved'],
    categorical_features=categorical_features,
    verbose=True,
    mode='classification'
)
```

After initializing the explainer, a specific instance from the test set is selected. The method $explain\_instance()$ works as follows:

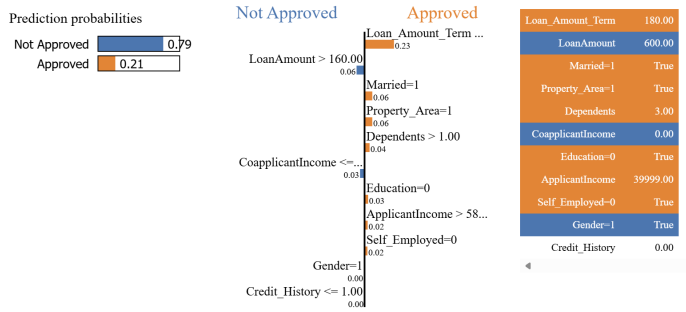


Figure 1.2: Sample visualization of LIME output, generated via Python code

1. It creates multiple perturbed samples around the selected instance.
2. It obtains prediction probabilities from the trained Random Forest model for these samples.
3. It fits a simple, interpretable model.
4. It extracts the most influential features contributing to the prediction.

The Python method requires both an input instance (as an array) and a prediction function that outputs the predicted probabilities, in this case $model.predict\_proba$, as shown in the following code:

```
exp = explainer.explain_instance(
    np.array(X_test_lime.iloc[3]), # Selceted instance
    lambda x: model.predict_proba(pd.DataFrame(x, columns=X_train_lime.columns)
    ),
    num_features=15
)
HTML(exp.as_html())
```

The output explanation shows:

- **Prediction Probabilities**: On the left, the predicted class for the selected instance with its probabilities
- **Feature Contribution Plot**: In the center, a ranked list with weights of features that contribute positively or negatively to the prediction
- **Explanation Contribution**: On the right, the contribution of each feature in the interpretable model

The example in Figure 1.2 shows that the selected instance ($X\_test\_lime.iloc[1]$) is predicted to be approved by the model, with a confidence of 95%. The explanation plot shows that the feature that contributes most to the prediction is $Loan\_Amount\_Term$. Meanwhile the features that pushed the prediction to "Not Approved" are: $Gender$, $Dependents$, $Married$, $Property\_Area$.

Shapley-Based Attribution Methods

SHAP (SHapley Additive exPlanations) (S. M. Lundberg et al., 2017) is grounded in cooperative game theory and attributes the output of a model to its input features by calculating their average marginal contribution across all possible feature subsets. The exact Shapley value ϕ_i for the input feature i is given by:

$$\phi_i = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(|N| - |S| - 1)!}{|N|!} [f(S \cup \{i\}) - f(S)] \tag{1.19}$$

Here, N is the set of all features and $f(S)$ is the prediction of the model when only the features in the subset S are present. Since computing all 2^n subsets is infeasible for high-dimensional data, SHAP relies on kernel-based sampling (e.g., KernelSHAP) or tree-specific optimizations (TreeSHAP).

Let us now apply SHAP to the same financial dataset and to the same pre-trained Random Forest classifier as the LIME, to predict the likelihood of loan approval. The primary goal is to provide explainability for the predictions through SHAP.

```
# Basic libraries
import pandas as pd
import numpy as np

# Library to load models
import joblib

# SHAP library
import shap
import matplotlib.pyplot as plt

# Preprocessing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

# Load Random Forest model and its dataset
model = joblib.load(r"data/loan_prediction_model.pkl")
data = pd.read_csv("data/df1_loan.csv")
```

```
# Initialize the SHAP explainer
explainer = shap.Explainer(model)
```

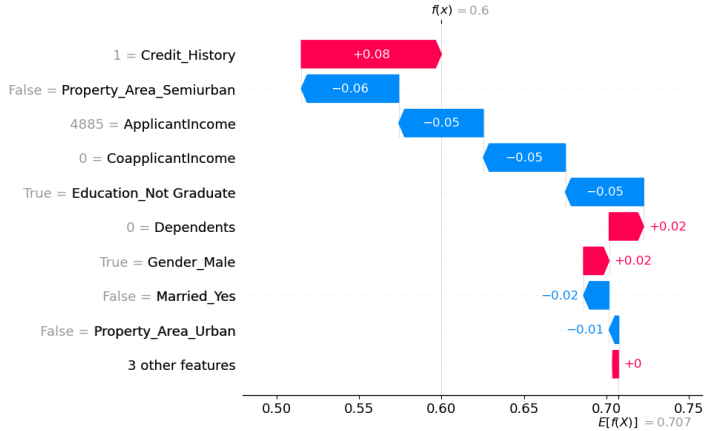


Figure 1.3: Waterfall SHAP plot

```
shap_values = explainer(X_test)
```

SHAP creates a model-agnostic SHAP explainer, and it automatically selects the most appropriate algorithm. The method $explainer(X\_test)$ computes SHAP values for every instance in the test set. The output contains the contribution of each feature to the model's prediction.

The method $waterfall\_plot()$ generates a waterfall plot for a selected instance and for a certain class. In the example the instance being the first test instance 0 and the the class 1, Approved. The Waterfall plot shows:

- The base value, meaning the average approval probability $E[f(x)]$
- How each feature contributes to the prediction: higher, for positive SHAP values, or lower, in case of negative ones. For example for the feature $Credit\_History$ the SHAP value is 0.08
- The final predicted probability for the selected instance $f(x)$

In the Waterfall Plot, in Figure 1.3, where x is the selected instance, $f(x)$ is the predicted value of the model and $E[f(x)]$ is the expected value of the target variable. For each feature, the length of the bar represents its SHAP value.

The method $summary\_plot$ shows SHAP values for all test instances for a certain class, in the case shown in Figure 1.4 for class 1, that is the one associated to the approval of the loan request. The Summary plot shows:

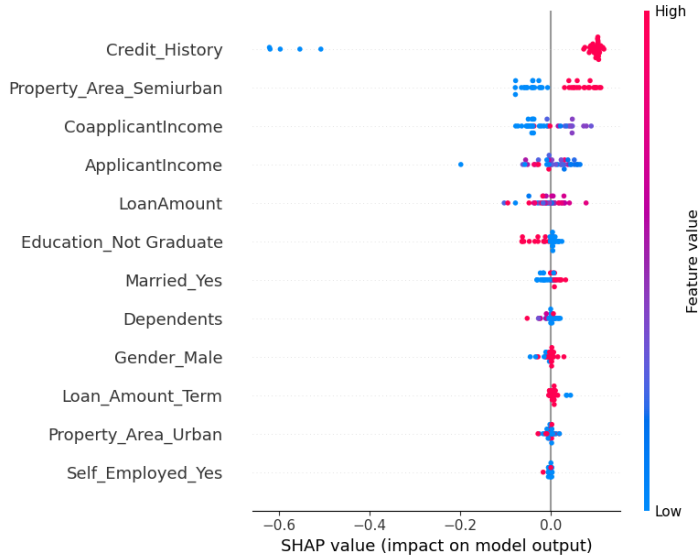


Figure 1.4: Summary SHAP plot

- A ranking of features by overall importance, in which $Credit\_History$ is shown to be the most important one
- The distribution of SHAP values for each feature, represented by the color spectrum from red to blue
- The relationship between feature values and their impact on predictions, represented by dots

```
shap.summary_plot(shap_values[..., 1], X_test)
```

```
shap.waterfall_plot(shap_values[0, :, 1])
```

Although SHAP ensures fairness and consistency, its quality of explanation depends heavily on how feature subsets are defined. In certain applications, data are not a static snapshot, but a sequence of observations over time, known as a time series. Applying standard SHAP to long time series can be problematic due to High computational cost for large numbers of time steps and the presence of missing or non-informative segments (background data).

tSHAP is an extension of SHAP specifically designed for time series data. It was introduced in 2025 to provide fast and exact SHAP values for time series classification and regression.

The method is based on two key ideas:

- A sliding window mechanism that groups time steps to reduce computational complexity while preserving exact SHAP values.
- Explicit handling of background (non-informative) data to reduce its negative impact on explanations.

An open-source Python implementation of tSHAP is available on Github[3].

Still speaking about a *post-modeling* focus on explainability, it is important to mention that recently, a new paradigm, referred to as *SAFE-AI* (Babaei, Giudici, and Raffinetti, 2025) , has been introduced to quantitatively measure an innovative set of metrics aimed at comparing models, both in terms of their *robustness* and in terms of their *explainability*.

SAFE-AI

Let Y and Y' be two random variables with cumulative distribution functions $F_Y, F_{Y'} : \mathbb{R} \to [0, 1]$. The p-order Cramér–von Mises divergence between their distributions is defined as:

$$CvM_p(F_Y, F'_Y) = \int_{-\infty}^{\infty} |F_Y(u) - F'_Y(u)|^p dF_Y(u) \tag{1.20}$$

In practice, F_Y and $F_{Y'}$ may represent the empirical cumulative distributions of the true outcomes and the corresponding model predictions on a test sample. More generally, the divergence can also compare two predictive distributions obtained from distinct datasets or model architectures.

Building on the first-order CvM divergence, we introduce the Rank Graduation (RG) index as:

$$\mathrm{RG}_1(Y, Y') = 1 - \frac{CvM_1(F_Y, F'_Y)}{G(Y)}, \tag{1.21}$$

where $G(Y)$ denotes the Gini coefficient computed from the empirical distribution of the true outcome (Auricchio et al., 2026).

An equivalent formulation of the CvM divergence can be expressed as:

[3] https://github.com/mlgig/tshap

$$\mathrm{CvM}_p(\mu, \mu^*) = \int_{\mathbb{R}} |F_\mu(t) - F_{\mu^*}(t)|^p \, dF_\mu(t) = \int_0^1 |L_{Y,Z}(t) - L_Y(t)|^p \, d\mu(t), \tag{1.22}$$

where $L_Y(t)$ denotes the Lorenz curve associated with Y, and $L_{Y,Z}(t)$ denotes the concordance curve between Y and Z.

In the binary setting, when Y represents the true labels and $Y' = \hat{Y}$ the predicted scores, the RG measure coincides with the classical Area Under the ROC Curve (AUC). Unlike AUC, RG naturally extends to ordinal and multiclass outcomes, providing a consistent ranking-based performance measure.

For multiclass problems, RG is computed using a one-vs-rest strategy: a score is evaluated for each class by comparing the class indicator with the predicted class probability. The overall multiclass RG is obtained as a convex combination of the per-class scores weighted by the empirical class frequencies.

The metric can also quantify different SAFE-AI dimensions depending on the variables (Y, Y') under comparison. When Y is the ground truth and $Y' = \hat{Y}$ the model prediction, it measures accuracy resilience (RGA) (Giudici and Raffinetti, 2025). When predictions on original data $Y = \hat{Y}$ are compared with predictions on perturbed inputs $Y' = \hat{Y}^{(p)}$, it evaluates robustness (RGR). Finally, when predictions are compared before $Y = \hat{Y}$ and after $Y' = \hat{Y}^{(p)}$ controlled feature modifications, it measures the explainability (RGE).

A key strength of the SAFE-AI framework is that RGA, RGR, and RGE are not separate metrics but specific realizations of the same RG family. This common foundation ensures conceptual consistency and enables their aggregation into a unified evaluation framework. Furthermore, it naturally extends to the AUC–based approach, where the integral of each RG curve provides a global summary measure (Giudici and Kolesnikov, 2026):

- **AURGA (Area Under the RGA Curve)** evaluates ranking stability when progressively removing the most confident predictions. For each removal fraction $x \in [0, 1]$, samples are sorted by prediction confidence, the top x fraction is removed, and RGA is recomputed. The resulting curve $\mathrm{RGA}(x)$ is integrated as:

$$\mathrm{AURGA} = \int_0^1 \mathrm{RGA}(x)\, dx$$

- **AURGR (Area Under the Robustness Curve)** measures prediction stability under increasing perturbation levels σ obtained through Gaussian noise injec-

tion. The predictions are recomputed at each perturbation level and compared with the original output using RGR, producing a curve $\mathrm{RGR}(\sigma)$, which is integrated as:

$$\mathrm{AURGR} = \int_0^1 \mathrm{RGR}(\sigma)\, d\sigma$$

- **AURGE (Area Under the Explainability Curve)** evaluates structural dependence on input components. Features are progressively removed according to a predefined strategy (greedy importance-based feature removal for tabular data, spatial or token masking for images and text), predictions are recomputed, and the resulting $\mathrm{RGE}(x)$ curve is integrated as:

$$\mathrm{AURGE} = \int_0^1 \mathrm{RGE}(x)\, dx$$

1.4 Financial Use Case

Classical models, as the *Logic Learning Machine* natively interpretable model have been tested on a financial dataset, measuring not only classical performance indicators (such as *accuracy*), but also the innovative metrics proposed by the *SAFE-AI* paradigm (Kolesnikov et al., 2026).

The considered dataset is derived from the 2017 Home Mortgage Disclosure Act (HMDA) records for the state of New York (Consumer Financial Protection Bureau, 2017). The data contain detailed information on mortgage loan applications and is used in studies of credit risk and fairness (Babaei, Giudici, and Wu, 2026). The target variable is *action_taken*, representing loan denial or approval. During preprocessing, irrelevant variables are removed while categorical attributes (e.g. applicant sex, race, ethnicity, loan type, lien status) are preserved as nominal features, which can be directly handled by the *Logic Learning Machine* model.

Across all datasets, we benchmark some interpretable models, such as *Logic Learning Machine* and *Logistic Regression*, against a set of widely used classifiers representing black-box approaches (the details of these techniques are outside the scope of this book): *Random Forest (RF)*, *Support Vector Machine (SVM)*, *Extreme Gradient Boosting (XGB)*, and a *Multilayer Perceptron (MLP)*. In addition, two *ensemble strategies* (that is, techniques in which multiple models contribute to the final prediction) are considered: a soft *Voting Ensemble Model (VEM)* combining RF and XGB, and a *Stacking Ensemble Model (SEM)* using RF and XGB as base learners with a Logistic Regression controlling their interaction, as a meta-learner.

Hyperparameter optimization for LR, RF, SVM, and XGB is performed using Optuna (Akiba et al., 2019), maximizing the mean macro-F1 score under an internal stratified cross-validation. The resulting optimal parameters are reused in the outer folds to prevent data leakage.

Model evaluation follows a *stratified 5-fold cross-validation* protocol. Models are trained on a subset of data (the so-called *training split*) and evaluated 5 times on the rest of the data. The performance corresponds to the average result on this portion of data, not used for training. Standard predictive performance is reported using Accuracy, macro-F1 and Mean Squared Error (MSE) computed from the predicted class probabilities.

Model	Accuracy	F1-score	MSE	RGA	AURGA	AURGR	AURGE
LR	0.637	0.568	0.225	0.683	0.249	0.650	0.661
RF	0.756	0.630	0.169	0.722	0.281	0.521	0.681
SVM	0.816	0.460	0.146	0.614	0.196	0.475	0.455
XGB	0.794	0.583	0.156	0.675	0.238	0.430	0.590
VEM	0.795	0.607	0.147	0.712	0.273	0.491	0.653
SEM	0.776	0.595	0.197	0.665	0.233	0.417	0.576
MLP	0.820	0.514	0.136	0.713	0.274	0.551	0.673
LLM (Specific)	0.800	0.583	0.152	0.668	0.235	0.658	0.609
LLM (General)	0.806	0.527	0.235	0.756	0.157	0.739	0.715

Table 1.4: Performance comparison across the HMDA dataset using standard and SAFE-AI metrics across 5 folds.

The results in Table 1.4 reveal the differences between predictive performance and SAFE-AI metrics in the HMDA dataset. Two configurations of the Logic Learning Machine are reported, corresponding to different rule generalization levels. More specific rules improve predictive fitting, while more general rules reduce predictive performance but increase robustness and explainability. This trade-off is particularly evident for this dataset and appears less pronounced in the image and text datasets considered below.

Considering the standard predictive metrics, MLP achieves the highest accuracy (0.820) and the lowest MSE (0.136), while RF obtains the best F1-macro (0.630), indicating a stronger balance under class imbalance. Other models such as SVM, VEM and XGBoost also reach high accuracy, although SVM shows the lowest F1-macro (0.460).

SAFE-AI metrics provide complementary insights into model behavior. RF achieves the highest RGA and AURGA values, while MLP and VEM obtain comparable results

in terms of these metrics, indicating stability of predictive rankings under feature removal. In terms of robustness to input perturbations, Logistic Regression shows strong AURGR despite lower predictive performance, highlighting that simpler models can remain stable under noise. Similarly, explainability robustness measured by AURGE is highest for Random Forest and MLP, while SVM appears more sensitive to the removal of informative features.

Focusing on the Logic Learning Machine, the configuration with more specific rules achieves a competitive predictive performance, while the more general configuration improves robustness and explainability, reaching the highest AURGR and AURGE values. Overall, these results highlight how SAFE-AI metrics reveal strong robustness and explainability properties depending on the rule generalization level.

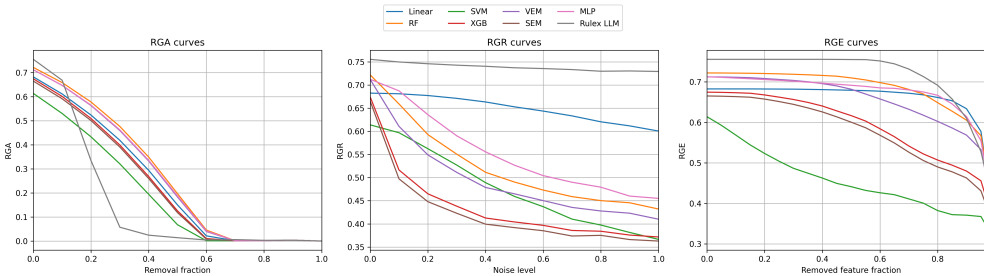


Figure 1.5: Example of SAFE-AI curves on the HMDA dataset. The figure shows the mean RGA, RGR, and RGE curves computed across the 5-fold evaluation for all models

Figure 1.5 illustrates an example of the SAFE-AI curves, which highlights differences in model stability under the SAFE-AI evaluation framework. Due to space limitations, SAFE-AI curves are reported only for the HMDA dataset as a representative example.

Consistent with the aggregated metrics reported in Table 1.4, the Logic Learning Machine configuration with more general rules exhibits stronger robustness patterns, reflected in the slower decrease of the RGR and RGE curves.

2 XAI On Unstructured Data

In the previous chapter, we have analyzed how it is possible to provide some motivations for decisions proposed by an algorithm or, more generally, how to make its behavior more auditable, assuming that our input information is stored in a table. Yet, it could be that our source of information has a more complex structure: for example, a (set of) photo(s) and/or a (set of) document(s).

In that case, a pure replica of what we have learned so far would be ineffective to deal with this new challenge because, for example, techniques like LIME or SHAP, in their native form (the one that we have investigated up to now) are not well-equipped to scale to *images* or *text*. In this chapter, we aim to give an overview about how XAI can be applied to this kind of non-tabular data, to which we refer to as *unstructured data*.

2.1 Images

A fundamental challenge in Machine Learning (ML) for Computer Vision is explaining how a black-box model classifies images, providing insights into the representations the model has learned from data. A key approach to this problem involves attributing importance scores to individual pixels, identifying their contribution to the decision-making process of the model. This task, commonly referred to as *explaining model predictions*, plays a crucial role in enhancing interpretability and trust in AI-driven image classification.

One of the most widely used methods for this purpose is an adapted version of SHAP (SHapley Additive exPlanations), which applies game-theoretic principles to the explainability of ML. SHAP combines feature removal (masking) (S. M. Lundberg et al., 2017) with hierarchical image partitioning (S. Lundberg, 2020), computing feature attributions on a refinable axis-aligned (AA) grid of pixels to approximate the regions most relevant to an image classifier.

Another influential method is an adaptation of LIME (Local Interpretable Model-agnostic Explanations) (Ribeiro et al., 2016a), which, despite lacking theoretical guaranties, remains popular for its ability to pre-identify relevant image regions through

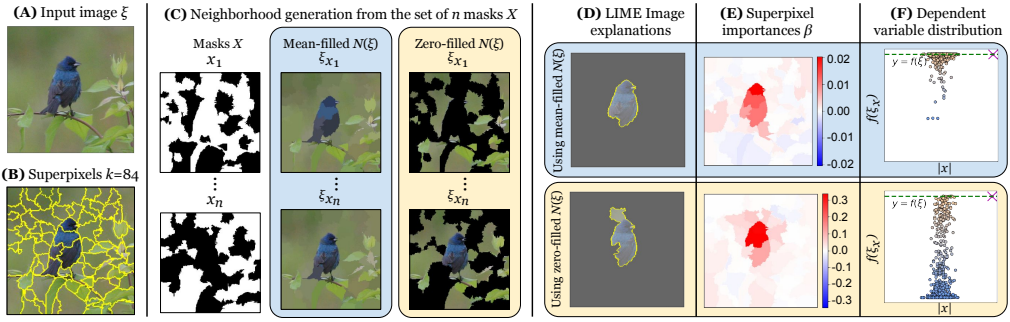


Figure 2.1: LIME Image workflow.

segmentation. However, LIME and similar approaches rely on predefined segmentation that matches the relevant image regions, and cannot adaptively refine these regions if the initial segmentation is suboptimal, limiting their effectiveness for complex image data.

LIME for Images

Let us briefly review how the LIME approach, described in the previous chapter for structured data, can be adapted to images.

Fig. 2.1 depicts the LIME Image workflow steps, and is used throughout this section to provide examples. Let us consider the domain of RGB images of size $h \times w$, denoted as $\mathcal{I} \in [0-255]^{h \times w \times 3}$.

Let $f : \mathcal{I} \rightarrow \mathbb{R}$ be a black-box model function that provides a prediction score given an input image [1], and let $\xi \in \mathcal{I}$ be the sample image being explained.

The main purpose of LIME is to generate a *linear* model g that locally approximates the explained black-box model f in the vicinity of an input sample ξ.

LIME explanations are not built directly on the image $\mathcal{I}$, but on a smaller domain denoted as the *interpretable representation*. This domain is obtained by dividing the input image into k *superpixels* (also called *segments*, *regions* , or *patches*) using a tool such as the *quick shift* (Vedaldi et al., 2008) algorithm. A superpixel is a contiguous region of pixels of ξ that share some kind of similarity, and such that the k superpixels form a partition of the pixels of ξ. Fig. 2.1A shows an example of an image taken from the Imagenet Object Localization Challenge. Fig. 2.1B shows its segmentation obtained

[1] We consider only the case of a binary class prediction, as the multi-class prediction is usually treated as several one-vs-rest binary class prediction problems.

from the *quick shift* algorithm[2], resulting in $k = 84$ superpixels. The model used for the classification is ResNet50 (He et al., 2016), pretrained for ImageNet task. The image in Fig. 2.1A is correctly classified as *indigo_bunting* with probability 99.49%.

The approach of LIME Image is based on the concept of *superpixel masking*. Let $x \in \{0,1\}^k$ be a binary vector (mask) representing the presence (value 1) or the absence (value 0) of each of the k superpixels. Using a mask x, a *perturbed input image* ξ_x is obtained by preserving the pixels of each superpixel i having $x[i] = 1$, and replacing every other pixel whose superpixel i has $x[i] = 0$. Replacement can be done in several ways. By default, pixels of a masked superpixel i are replaced by the mean color of that superpixel (*mean-filled*). Alternatively, they can be replaced with a fixed color value, like black (*zero-filled*). We use notation $x' = x[i \leftarrow v]$ to denote a new mask x' obtained from a mask x by replacing the value for superpixel i with v. Moreover, let $|x|$ be the number of preserved superpixels, i.e. those having $x[i] = 1$.

In LIME Image, the individual values of a mask vector x are sampled using an unbiased Monte Carlo strategy, i.e.

$$x[i] \sim B(0.5), \qquad 1 \leq i \leq k \tag{2.1}$$

where $B(p)$ is a Bernoulli-distributed random variable that has probability p=0.5. A *set of masks* X with n samples is made by randomly sampling n instances of (2.1) for the same input image ξ having k superpixels. A *synthetic neighborhood* $N(\xi) = \{\xi_x \mid x \in X\}$ with n samples is created by perturbing the input image ξ using n randomly sampled masks. A depiction of the set of masks n is shown in Fig. 2.1C: randomly sampled masks x_i are used to generate perturbed input images ξ_{x_i}, using two replacement strategies.

All perturbed samples $N(\xi)$ can be classified by the black-box model f, resulting in *dependent variables*:

$$Y = \{f(\xi_x) \mid \xi_x \in N(\xi)\} \tag{2.2}$$

A *distance function* is adopted, in order to weight the perturbed samples differently. The intuition followed by LIME is that samples closer to ξ should weigh more. Given a mask x, the weight w_x is:

$$w_x = \exp\left(\frac{-D(x)^2}{\sigma^2}\right) \tag{2.3}$$

where D is the cosine similarity score between x and $\vec{\mathbf{1}}$ (the vector of ones, i.e., the

[2]Using: $kernel\_size = 4$, $max\_dist = 7$, $ratio = 0.2$.

mask where everything is preserved), while $\sigma = 0.25$ (by default) is the *width of the kernel*. See (Garreau et al., 2020) for an analysis on the role of (2.3) and σ.

Let us use the default value, as the focus is on the sampling methodology; let $W = \{w_x \mid x \in X\}$. Having the mask set matrices $X \in \{0,1\}^{n\times k}$, the weights $W \in \mathbb{R}^{n\times 1}$, and the dependent variables $Y \in \mathbb{R}^{n\times 1}$ for all the observed samples in the synthetic neighborhood $N(\xi)$, then Y can be written as the response variable of the *linear regression model*. LIME adopts a *simple linear homoschedastic model* (DuMouchel et al., 1983) for its regression coefficients, which is:

$$Y = X \cdot \beta + \epsilon \tag{2.4}$$

where the vector β is the weighted least squares estimator of the regression coefficients of Y on X weighted by W.

To simplify our analysis, we consider no regularization factors (default for LIME Image is ridge regression with L^2 regularization), similarly to (Garreau et al., 2020). This simplification does not significantly alter our main considerations, which are focused on the sampling strategy.

The coefficients β are results from:

$$\beta = (X^\mathsf{T} W X)^{-1} X^\mathsf{T} W Y \tag{2.5}$$

which solves Eq. (2.4). A linear function $g(x)$ with coefficients β is a linear regressor that locally approximates the initial black-box model f.

The k coefficients of β can be interpreted as *the importance* of the characteristics (or *the attribution* of the characteristics) of each of the k superpixels of the input image ξ. In that sense, the k superpixels form the set of *interpretable features* of the input image, over which the explanation is built.

There are two levels of interpretation of β. By default, LIME Image suggests to select only the superpixels with the highest value (Fig. 2.1D), resulting in a sub-region in the image (the get_image_and_mask method). The number of selected superpixels is decided by the user: LIME does not provide a heuristic for this task. Alternatively, the coefficients can be visualized as a *heatmap*, identifying the contribution of each superpixel to the classification (Fig. 2.1E). The intensity of the color represents the value, with white representing the zero. Coefficients with higher absolute values mean that the corresponding superpixel is more important in the classification result $f(\xi)$.

The scale of the coefficients can vary (in Fig. 2.1E the same scale is used for both heatmaps) and it is known as not particularly relevant, as shown in (Garreau et al., 2020) (only the ratios among the coefficients are).

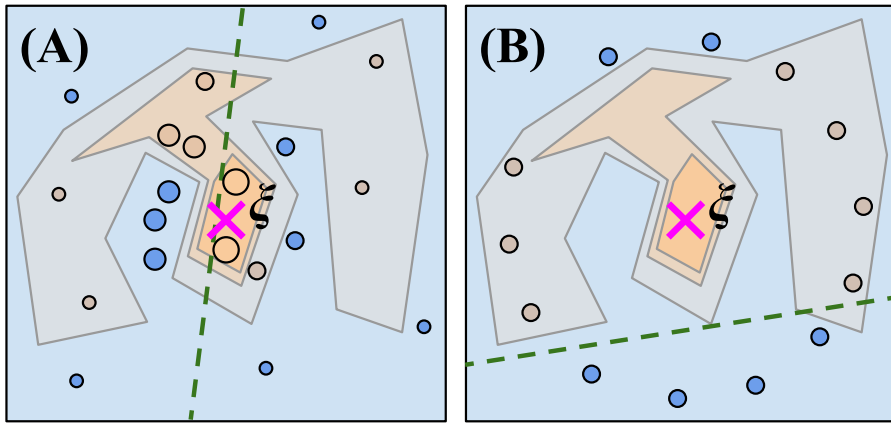


Figure 2.2: How LIME is supposed to work (A), and how it actually works (B) using Monte Carlo sampling for a large enough k. Source: (Rashid et al., 2024)

Finally, it is important to examine the distribution of the Y values in the neighborhood (i.e. the values of $f(\xi_x)$) with respect to the count $|x|$ of masked superpixels (Fig. 2.1F). This plot shows if the Y values are sampled throughout the distribution (top to bottom), or if there are clear unbalances. In Fig. 2.1F, the distribution for the zero-filled case has a good balance, since there are values obtained from the black box model f covering the entire spectrum of values, while in the plot for the mean-filled case the balance is problematic, having most Y values concentrated in the top.

Shapley for images and Hierarchical Coalition Structures (HCS)

Let us consider the setup of a n-coalition game $(\mathcal{N}, \nu)$, which can be considered analogous to an importance scores attribution task in XAI (Rozemberczki et al., 2022a). The finite set $\mathcal{N} = \{1, \ldots, n\}$ is the set of players (*features*). Each nonempty subset $S \subseteq \mathcal{N}$ is a *coalition*, and $\mathcal{N}$ is itself the *grand coalition*.

A *characteristic function* $\nu : 2^n \to \mathbb{R}$ assigns to each coalition S a (real) *value of value* $\nu(S)$, and it is assumed that $\nu(\varnothing) = 0$ (it is always possible to ensure $\nu(\varnothing) = 0$ by translation of the equation system).

A *marginal contribution* of a player i to a coalition S (assuming $i \notin S$) is given by:

$$\Delta_i(S) = \nu(S \cup \{i\}) - \nu(S) \tag{2.6}$$

Semivalues (Dubey et al., 1981), weighted sums of marginal contributions (2.6), were

proposed to address the issue of fair distribution of the total value $\nu(\mathcal{N})$ of the grand coalition $\mathcal{N}$ among its members. The Shapley value, a well-known semivalue introduced in (Shapley, 1953), demonstrates favorable axiomatic properties and has been used effectively to explain ML models (Rozemberczki et al., 2022a).

A fixed a-priori *coalition structure* (López et al., 2009; Guillermo Owen, 2013; Guilliermo Owen, 1977) for the $\mathcal{N}$ players is a finite set $\{T_1 \dots T_m\}$ of m partitions of $\mathcal{N}$ (i.e. $\cup_{k=1}^{m} T_k = \mathcal{N}$, and $T_i \cap T_j \neq \varnothing \Leftrightarrow i = j$). Elements T_i are usually called *partitions*, *coalitions*, *teams* or *unions*.

Let us consider a recursive definition of a hierarchical coalition structure, where each partition T can be either an *indivisible partition* or a *sub-coalition structure* itself $T = T_1 \cup \ldots \cup T_m$. Let $T{\downarrow}$ be the (downward) recursive partitioning of T, defined as:

$$T{\downarrow} = \begin{cases} \{T_1 \dots T_m\} & \text{if } T \text{ admits a sub-coalition structure} \\ \perp & \text{if } T \text{ is indivisible} \end{cases} \tag{2.7}$$

We denote with $\mathcal{T}$ the HCS root, and assume w.l.o.g. that $\mathcal{T}$ contains all the elements of $\mathcal{N}$.

A special case of HCS happens when each sub-coalition structure is made by two partitions, i.e. the hierarchy forms a binary tree. We refer to these structures as *binary hierarchical coalition structures* (BHCS). In that case the recursive downward partitioning of T can be simplified as:

$$T{\downarrow} = \begin{cases} \{T_1,\ T_2\} & \text{if } T \text{ admits a binary sub-coalition structure} \\ \perp & \text{if } T \text{ is indivisible} \end{cases} \tag{2.8}$$

Computing Shapley values has exponential time complexity, which is unfeasible for image data with hundreds or thousands of features (pixels). An approximate approach, introduced by (Guilliermo Owen, 1977), can be used to drastically reduce complexity by grouping features into hierarchical coalitions.

This concept has been pioneered for image data by the SHAP Partition Explainer (S. Lundberg, 2020; Shrikumar et al., 2017; S. M. Lundberg et al., 2017).

A *coalition value* $\Omega_i(\mathcal{T})$ represents the worth of the player i in a game with coalition structure $\mathcal{T}$, and is known as the Owen coalition value (Guilliermo Owen, 1977). Computing coalition values on a binary HCS T as defined in (2.8) can be done with a

recursive formula:

$$\Omega_i^{\mathrm{B}}(Q,T) = \begin{cases} \frac{1}{2}\Omega_i^{\mathrm{B}}(Q \cup T_2,\, T_1) \,+\, \frac{1}{2}\Omega_i^{\mathrm{B}}(Q,\, T_1) & \text{if } T{\downarrow} = \{T_1,\, T_2\} \\ \frac{1}{|T|}\Delta_T(Q) & \text{if } T \text{ is indivisible} \end{cases} \tag{2.9}$$

s.t. $\Omega_i(\mathcal{T}) = \Omega_i^{\mathrm{B}}(\varnothing, \mathcal{T})$. The former case of Eq. (2.9) deals with coalitions T that admit a sub-coalition structure $T{\downarrow} \neq \perp$. We assume, for notational simplicity and without loss of generality, that $i \in T_1$. The latter case of Eq. (2.9) deals with indivisible coalitions. In that case, the formula assigns a single coalition value to all players in T, divided uniformly among all members in T.

Stratified LIME

Let us now consider a variation of LIME which has been recently introduced to improve its effectiveness in the image domain: *Stratified LIME* (Rashid et al., 2024).

Our initial consideration is that, although there have been a number of successful applications of LIME (Bodria et al., 2023), the effectiveness of the explanation process largely depends on several factors.

One of such factors is the sampling process, which is stochastic and inherently uncertain. The use of a Monte Carlo strategy in Eq. (2.1) to sample the interpretable feature space when it is made up of more than a few dozen superpixels has important consequences.

Under-Representation of the Neighborhood. The intuition behind LIME is depicted in Fig. 2.2A, which is inspired by the one found in Ribeiro et al., 2016b, Fig. 3. The explained sample ξ (represented as a cross) is surrounded by its synthetic neighborhood $N(\xi)$ (represented as dots), whose classifications are obtained by the black box model f and weighted by their proximity to ξ (size of dots). A linear regressor (the green dashed line) is fit on these points weighted by their distance to ξ, and in principle it should be locally faithful to $f(\xi)$. LIME Image however works like that only when the number of superpixels is very small. Since masks are obtained from Eq. (2.1) having a fixed Bernoulli coefficient of 0.5, the probability of selecting a mask x having a given number of preserved superpixels $|x|$ follows the binomial distribution $\mathcal{B}(k, |x|)$ with probability mass function $\binom{k}{|x|}p^{|x|}(1-p)^{k-|x|}$.

Fig. 2.3A shows the probability mass function for a few k values, k being the number of superpixels. This PMF is of course not uniform, and the probability of randomly

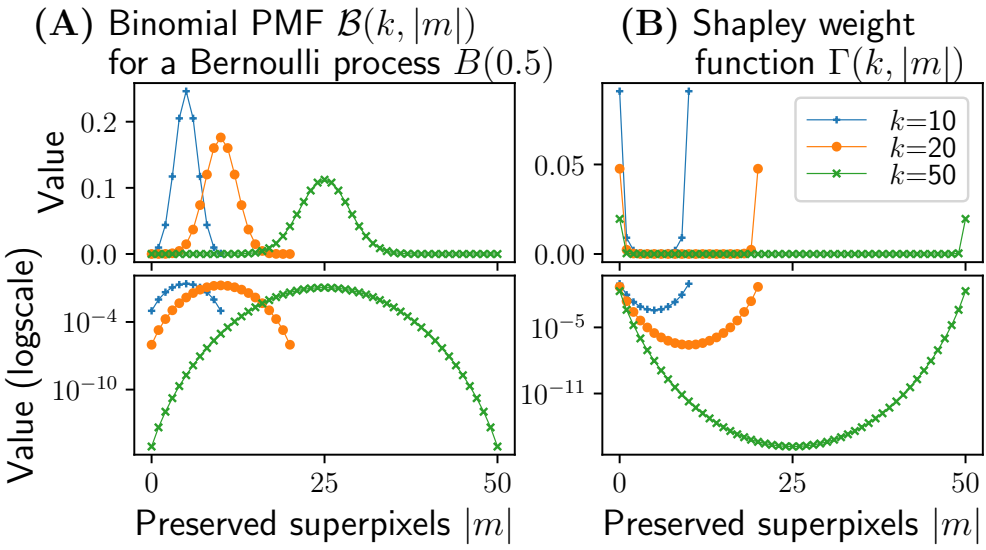
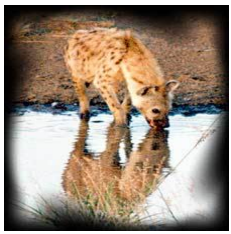


Figure 2.3: Binomial (A) and Shapley weight (B) distributions for $k = 10, 20$ and 50. Source: (Rashid et al., 2024)

sample points at the extremes drops rapidly. There is no indication of how many superpixels a LIME Image can manage, but both the default parameters and practical experience (Vermeire et al., 2022) shows that an image needs to be split into tens or even a few hundreds of superpixels, in order to have enough patches to correctly identify object borders. In that case, samples will distribute around ξ forming a sort of hypersphere, as illustrated in Fig. 2.2B, where almost no sample is really close to ξ, since the probability of the binomial distribution concentrates around samples having $\sim$50% of the superpixels masked. In that way, the local behaviour (i.e. samples with $|x|$ close to k) is under-represented in the neighborhood.

Dependent Variables Distribution. As seen in Fig. 2.3A, by increasing the superpixels k the probability of obtaining samples from the tails of the distribution is practically reduced to 0. This effect depends on both the model and the input image: if selecting randomly about 50% of the superpixels still allows the model to produce a "reasonable" distribution of the dependent variable Y, a linear regressor can be fit and an explanation can be produced. If however the Y distribution is flattened, no reasonable explanation can be produced, as the linear regressor will be fit on almost uniform

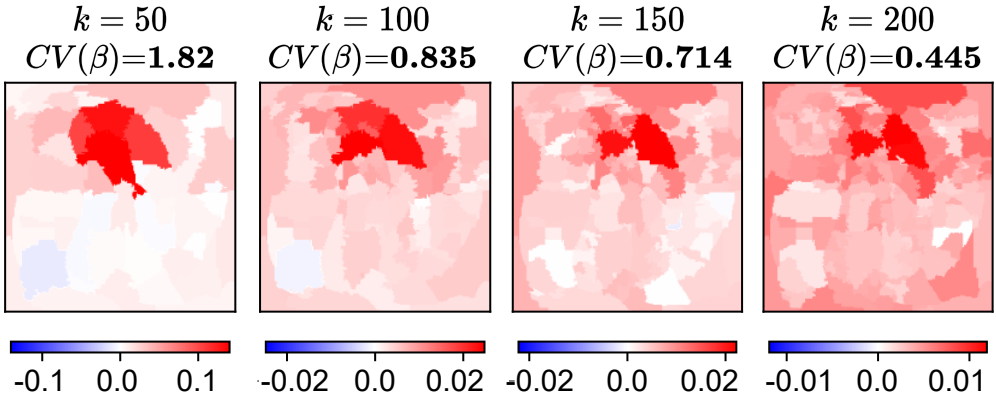
values.



(A) Input image ξ
Class: *hyena*
Probability: 99.46%
num_samples n = 1000
Using mean-filled $N(\xi)$

k	*max_dist*
50	8.691
100	4.956
150	4.092
200	3.632

(B) Feature importances β for four segmentations.



(C) Dependent variable distributions.

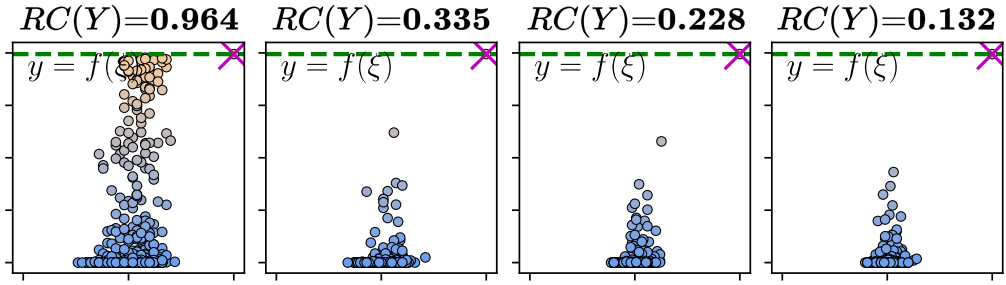


Figure 2.4: Dependent variable undersampling (low $RC(Y)$) results in confused explanations (low $CV(\beta)$). Source: (Rashid et al., 2024)

Fig. 2.4 shows an example of this behaviour. The input image **(A)** is correctly classified by the model as *hyena* with high probability. Feature importance vectors β and

the distribution of the dependent variables Y (versus the number of masked superpixels $|x|$) are shown in **(B)** and **(C)**, respectively, for four different segmentations ($k = 50, 100, 150$ and 200 superpixels, respectively). All values (heatmaps, $CV(\beta)$, $RC(Y)$) are averages of 10 computations, to reduce randomness in the reported results. With $k = 50$ segments (left), the Y distribution has enough variability to obtain a vector β that highlights which segments are more important. Increasing the number of superpixels reduces such variability in the Y distribution, resulting in explanations that are more and more "confused". On these distributions it is of course harder to fit a linear regressor that is truthful to the explanation. Intuitively, it is like Fig. 2.2B where the hypersphere is almost entirely far away from ξ. In that case, the explanation produced by the LIME Image will be progressively more meaningless.

In these problematic cases the values of the β vector also drops to very small numbers (scale is reported below each heatmap in **(B)**), and variability across the feature importances decreases. To quantitatively measure such form of "confusion", we employ the standard *coefficient of variation*, defined as:

$$CV(\beta) = \frac{\sigma_\beta}{\mu_\beta} \tag{2.10}$$

where σ_β and μ_β are the standard deviation and the mean of β, respectively. Ideally, a good $CV(\beta)$ should not be close to zero (which would mean that all superpixels have almost the same value, and no clear sub-region in the image is identified). The $CV(\beta)$ values for the example in Fig. 2.4 are reported in the **(B)** row.

We also want to quantify the (approximate) *range coverage* of the Y values in the synthetic neighborhood. Theoretically this range is $[0, f(\xi)]$, but of course it can have under- or over-shoots due to the nature of the classification model. To do so, we measure the proportion of that range that is contained in the $1\% - 99\%$ interquantile range (IQR) of the Y distribution, using:

$$RC(Y) = \frac{IQR_{1-99}(Y)}{f(\xi)} \tag{2.11}$$

Low values of $RC(Y)$ indicate that the sampled Y distribution is squashed into a small range of values, not covering the full $[0, f(\xi)]$ spectrum (like in Fig. 2.4C/right). Ideally $RC(Y)$ should be far from zero to have a good coverage of the probability range $[0, f(\xi)]$ by Y.

Sample Relevance. In recent years, the Shapley theory (S. M. Lundberg et al., 2017) has received a lot of attention in the context of model-agnostic explainability, due to its flexibility and its axiomatic formulation (Rozemberczki et al., 2022b). While LIME does not have a corresponding axiomatic definition, we can still learn some insights from how Shapley values are defined over a weight sample space.

The Shapley value for a superpixel i, that can be interpreted as an *importance* score, is defined by:

$$\phi_i = \sum_{x \in X^{[\![i]\!]}} \Gamma(k-1, |x|)\big(f(\xi_{x[i\leftarrow 1]}) - f(\xi_x)\big) \tag{2.12}$$

with $X^{[\![i]\!]}$ being the set of all masks x having $x[i] = 0$, and with the *Shapley importance* function as discussed in (Monderer et al., 2002)

$$\Gamma(k, |x|) = \frac{1}{(k+1)\binom{k}{|x|}} \tag{2.13}$$

Fig. 2.3B shows the Shapley importance function for a few k values. Higher values of $\Gamma(k, |x|)$ for a mask x mean that samples having that mask will weight more in the final value of ϕ_i. Comparing Figures 2.3A and B clearly shows that LIME Image samples the majority of the masks among those having the least importance (in the Shapley sense). In fact when $p = 0.5$ the following holds:

$$\mathcal{B}(k, |x|) \cdot \Gamma(k, |x|) = \frac{\binom{k}{|x|} p^{|x|}(1-p)^{k-|x|}}{(k+1)\binom{k}{|x|}} = \frac{0.5^k}{k+1}$$

i.e. the Shapley importance is the reciprocal (times a constant) of the binomial distribution $B(0.5)$ used by LIME. This is an informative detail of the Shapley theory, which motivates the proposed sampling theory.

Interestingly, Shapley value computation is not typically performed as a Monte Carlo sampling, but adopts other strategies to generate the samples (Okhrati et al., 2021; Mitchell et al., 2022). For instance, in (Guillermo Owen, 1972) Eq. (2.12) is rewritten as:

$$\phi_i = \int_0^1 \left(\sum_{x \in X_q^{[\![i]\!]}} \frac{1}{|X_q^{[\![i]\!]}|} \big(f(\xi_{x[i\leftarrow 1]}) - f(\xi_x)\big) \right) \mathrm{d}q \tag{2.14}$$

with $X_q^{[\![i]\!]}$ being a random subset of masks x, having $x[i] = 0$ and, for all $j \neq i$, $x[j] \sim B(q)$ with $B(q)$ a Bernoulli-distributed random variable having probability q.

Such strategy allows to get samples across the entire spectrum of $|x|$ values. In this paragraph we discuss a strategy for LIME Image where x values are not sampled from $B(0.5)$ as in Eq. (2.1) but from a modified version of Eq. (2.14).

Let us now describe a methodology based on stratified sampling of the X values, where each stratum has a uniform probability of being selected and represented in the samples of X. This oversamples the "rare" samples at the tail of the Y distribution, improving the samples over which the linear regressor is fit. However, this sampling could result in a form of *bias*. To avoid that, an adjustment factor is introduced to counterbalance the oversampled data points.

Let $\mathcal{X}$ denote the complete population of mask samples, having 2^k elements, and let $\mathcal{Y}$ be the dependent variable of $\mathcal{X}$. Let us consider a stratified partitioning. Let $\mathcal{X}^{(i)}$ be the set of all possible masks having $|x| = i$, i.e. for which exactly i super-pixel are preserved. Clearly, $\mathcal{X}^{(0)} \cdots \mathcal{X}^{(k)}$ forms a partitioning of all possible masks, and:

$$\{0,1\}^k = \bigcup_{i=0}^{k} \mathcal{X}^{(i)}$$

since any possible mask x appears in one (and only one) set $\mathcal{X}^{(|x|)}$. Moreover $\mathcal{X}^{(0)}{=}\{\vec{\mathbf{0}}\}$ and $\mathcal{X}^{(k)}{=}\{\vec{\mathbf{1}}\}$ (masks for the explained input sample with everything/nothing perturbed, respectively). Each stratum $\mathcal{X}^{(i)}$ does not have a uniform number of samples, but its size is known a-priori since they follow the binomial distribution, i.e.

$$|\mathcal{X}^{(i)}| = \binom{k}{i}, \qquad 0 \leq i \leq k \tag{2.15}$$

In an unbiased Monte Carlo sampling model, as in Eq. (2.1), the probability of selecting a sample x in a stratum $\mathcal{X}^{(i)}$, with $i = |x|$, is therefore proportional to that stratum probability in the overall population $\mathcal{X}$, i.e.:

$$Prob\{x \in \mathcal{X}^{(i)} \mid x \in X\} = \frac{|\mathcal{X}^{(i)}|}{\sum_{j=0}^{k} |\mathcal{X}^{(j)}|} = \frac{\binom{k}{i}}{2^k}$$

Let $\widehat{X}$ be an oversampled population, where the probability of taking samples from any of the $k+1$ strata is uniform and does not depend on the stratum size, i.e.:

$$Prob\{x \in \mathcal{X}^{(i)} \mid x \in \widehat{X}\} = \frac{1}{k+1}$$

Let $\widehat{Y}$ be the corresponding dependent variables for $\widehat{X}$. We can derive an *adjustment factor* for the $\widehat{X}$ samples to correct the bias introduced by the oversampling, which results for an arbitrary sample x in stratum $\mathcal{X}^{(i)}$ as:

$$adj(i) = \frac{Prob\{x \in \mathcal{X}^{(i)} \mid x \in X\}}{Prob\{x \in \mathcal{X}^{(i)} \mid x \in \widehat{X}\}} = \frac{(k+1)\binom{k}{i}}{2^k} \tag{2.16}$$

The weighted regression with the oversampled set $\widehat{X}$ can be obtained by inserting the adjustment factor as a multiplicative term in the existing weight equation of LIME. Let $\widehat{w}_{\widehat{x}}$ be the weight of sample $\widehat{x} \in \widehat{X}$ obtained from Eq. (2.3) multiplied by $adj(|\widehat{x}|)$, and let $\widehat{W} = \{\widehat{w}_{\widehat{x}} \mid \widehat{x} \in \widehat{X}\}$ be the set of weights for the set $\widehat{X}$. Then let:

$$\widehat{\beta} = (\widehat{X}^{\mathsf{T}}\widehat{W}\widehat{X})^{-1}\widehat{X}^{\mathsf{T}}\widehat{W}\widehat{Y} \tag{2.17}$$

be the weighted least square estimator of the regression coefficients of $\widehat{Y}$ on $\widehat{X}$ that takes into account the strata density of the oversampled set $\widehat{X}$.

The Mixture Model. The linear homoscedastic regression model of Eq. (2.4) adopted by LIME may not be particularly accurate when strata at the tails are severely undersampled, and these strata are significantly different from the mean. In that case, β is not globally unique across the sampled population, but varies by stratum:

$$\widehat{Y}^{(i)} = \widehat{X}^{(i)} \cdot \widehat{\beta}^{(i)} + \widehat{\epsilon}^{(i)} \tag{2.18}$$

Intuitively, the $\widehat{\beta}^{(i)}$ vectors represents the feature importance for stratum i, which is at uniform "distance" from the input sample ξ. The closer i is to k, the closer ξ_x is to ξ.

Impact of Stratified Sampling in LIME Image. The impact of using a weighted regression from a stratified sampling schema may not be negligible. We simplify the analysis by considering two cases.

Case (A): The mean and variance of $\widehat{\beta}^{(i)}$ are independent of the strata (i.e. the population structure is *homoscedastic*). Then it is easy to see that $\mathbb{E}[\beta] \approx \mathbb{E}[\widehat{\beta}^{(i)}]$, for any i. In that case, a weighted regression model of Eq. (2.17) is not needed, and the model computed by LIME using Monte Carlo sampling will not have issues due to the undersampling of the tails. In that case, the stratified sampling will converge to the same values, regardless of the strata ratios in the synthetic neighborhood.

Algorithm 2: Neighborhood sampling strategies

function *MonteCarloSampling*(n, k)
- $X \leftarrow n \times k$ matrix ;
- **for** i *between 1 and* n **do**
  - **for** j *between 1 and* k **do**
    - $X[i, j] \leftarrow B(0.5)$

function *StratifiedSampling*(n, k)
- $X \leftarrow n \times k$ matrix ;
- **for** i *between 1 and* n **do**
  - $q \leftarrow Uniform(0, 1)$;
  - **for** j *between 1 and* k **do**
    - $X[i, j] \leftarrow B(q)$;
    - $adj[i] \leftarrow (k + 1) \cdot \frac{1}{2^k} \cdot \binom{k}{|X[i]|}$

Case (B): The mean and variance of $\widehat{\beta}^{(i)}$ vary by stratum. In that case, the bias introduced by the Monte Carlo sampling scheme will not allow to consider systematic differences in the stratum, and a weighted regression or a mixed model built on a stratified sampling strategy is highly recommended (DuMouchel et al., 1983).

In a certain sense, Case (B) is even worse, because the undersampling of the neighborhood of ξ breaks the logic of building models that are locally faithful to the black box model f in the neighborhood of the explained sample, since the local neighborhood (close to ξ) that is really representing the local behavior is missing/undersampled.

Algorithm 2 outlines two sampling methods: the original Monte Carlo sampling used by LIME Image, and the introduced stratified sampling technique.

The *MonteCarloSampling* function computes the data matrix X (from Eq. 2.1) with replacement.

StratifiedSampling is one possible way to generate a stratified population, similarly to Eq. (2.14). For every sample i, a single coefficient q is randomly drawn from a uniform distribution that varies between 0 and 1.

The individual values of the i-th mask vector are then sampled from a Bernoulli random variable $B(q)$ with probability q. This will obtain a sample $X[i]$ in stratum $\widehat{X}^{(|X[i]|)}$, where strata now have the same probability of being selected. The adjustment factor $adj[i]$ for the sample i is also calculated.

Other strategies could also be used (Rao, 1977). An interesting approach suggested in (Konijn, 1962) for computing the coefficients would be to fit one linear regressor for every strata and then form a *mixed model* with the averages of the coefficients.

However, this approach requires more changes in the LIME code; thus we have favored the approach of Algorithm 2 which is more straightforward. Experimental results showing the effectiveness of the proposed technique are described in (Rashid et al., 2024). A Python implementation of the technique is publicly available on Github[3].

[3]https://github.com/rashidrao-pk/lime_stratified

2.2 Text

Thinking about technologies and algorithms allowing us to learn from unstructured text, the first thing that probably comes to our mind are modern *chatbots*, that is, tools which are able to answer a text message with another text message, aiming to emulate a conversation among human beings.

From this point of view, the public release of ChatGPT 3 at the end of 2022 catalyzed broad attention to language technologies by offering a natural, fluid, and contextually adapted dialog both to non-specialists and experts. What made this wave of *chatbots* compelling is not only their practical versatility but also the impression of "speaking our language"—accepting free-form natural language prompts and returning similarly unconstrained answers.

However, despite apparent intelligence, such systems remain engineered artifacts produced by training on vast corpora with statistical learning objectives. Their ability to *appear* to understand is grounded in vectorized representations, sequence modeling, and large-scale optimization rather than in semantic comprehension per se (Brown et al., 2020; Le Scao et al., 2022).

In parallel with these capabilities, the recent literature has started to question the rigidity of current architectures and training pipelines, in particular the sharp divide between offline pre-training and deployment-time usage. A new proposal dubbed *Nested Learning* (NL) aims to close this divide by organizing learning as a hierarchy of coupled optimization processes operating on multiple timescales, with the goal of mitigating catastrophic forgetting and enabling permanent structured adaptation (Behrouz et al., 2025).

This section aims to resume three complementary perspectives: (i) the historical and conceptual foundations from *bag-of-words* to embeddings and Transformers; (ii) the observed versatility and educational affordances of modern chatbots; and (iii) the NL paradigm as an integrated approach to continual learning and optimizer–architecture co-design (Mikolov et al., 2013; Vaswani et al., 2017; Raffel et al., 2020; Behrouz et al., 2025).

From Counts to Vectors: Embeddings as the Interface Between Text and Computation

Early computational treatments of text typically relied on the *bag-of-words* representation: tokenization over a fixed vocabulary followed by frequency counts, yielding sparse, high-dimensional vectors for each document. While useful for rudimentary similarity via dot products, this approach discards order, fails to encode synonymy,

and provides little inductive bias about relational structure among terms. For example, in a bag-of-words model the sentences "The cat chased the mouse" and "The mouse chased the cat" receive identical vector representations, even though they describe opposite situations in terms of who is acting on whom.

Classical `word2vec` models operationalize the idea of learning *word embeddings* through the definition and the optimization of a predictive objective. The core setup is a large corpus and a fixed vocabulary; each word w in the vocabulary is associated with two trainable vectors: an "input" embedding v_w and an "output" embedding u_w. Training proceeds by applying a sliding window over text and defining a prediction task:

- In the *CBOW* (Continuous Bag-of-Words) variant, the model is given the embeddings of the surrounding context words and must predict the center (target) word.

- In the *Skip-gram* variant, the model is given the embedding of the center word and must predict each context word within a fixed window.

In practice, *skip-gram with negative sampling* became the most widely used configuration (Mikolov et al., 2013). When we speak about *skip-gram with negative sampling*, we mean that, for a target word w_t and a context word w_c, the model maximizes the probability that this pair comes from real text, while simultaneously minimizing this probability for several randomly sampled "negative" words w_n.

An optimizing algorithm (usually *stochastic gradient descent* (Amari, 1993), or a close variant) is used to update both the input and output embeddings so that true co-occurring words are pulled closer together in the embedding space, whereas randomly paired words are pushed apart.

This simple predictive mechanism yields several notable properties: for example, words that appear in similar contexts (e.g., "doctor" and "physician") acquire similar vectors, embodying the distributional hypothesis ("you shall know a word by the company it keeps"). Moreover, the learned geometry often encodes semantic and syntactic relationships as approximately linear directions in the space (e.g., king $-$ man $+$ woman $\approx$ queen). In addition to that, negative sampling and related tricks (like subsampling very frequent words and using a tailored unigram distribution for negatives) make training feasible on billions of tokens with relatively small models. Finally, embeddings are typically characterized by a dimension between 100 and 400 features, capturing a much richer similarity structure than sparse one-hot or count-based vectors, while remaining compact and amenable to downstream models.

Thus, *word2vec* concretely implements the transition described above: instead of relying on raw co-occurrence counts or high-dimensional bag-of-words representations, it learns dense vectors directly from a predictive objective, jointly optimizing the embedding parameters and the shallow network that defines the prediction task. This shift underpins the success of many subsequent embedding and contextualization methods.

Transformers and the Attention Mechanism

Recurrent neural networks (RNNs), Long Short-Term Memory (LSTM) networks, and Transformers form a historical progression of architectures for sequence modeling, each addressing key limitations of its predecessors.

Classical RNNs process sequences step by step, maintaining a hidden state that is updated recurrently:

$$h_t = f(W_{xh}x_t + W_{hh}h_{t-1} + b_h), \quad y_t = g(W_{hy}h_t + b_y),$$

where x_t is the input at time t, h_t is the hidden state, and f, g are nonlinearities. This recurrence gives RNNs a form of memory: information from earlier time steps can, in principle, influence later predictions.

However, standard RNNs suffer from *vanishing and exploding gradients*: backpropagation through many time steps causes gradients to decay or blow up, making it hard to learn long-range dependencies. Moreover, each hidden state depends on the previous one, which prevents efficient parallelization over time steps and slows training.

LSTMs were introduced to address the temporal credit assignment and gradient problems. They augment the RNN with a cell state c_t and gating mechanisms that regulate information flow:

$$\begin{aligned}
f_t &= \sigma(W_f[h_{t-1}, x_t] + b_f) && \text{(forget gate)} \\
i_t &= \sigma(W_i[h_{t-1}, x_t] + b_i) && \text{(input gate)} \\
\tilde{c}_t &= \tanh(W_c[h_{t-1}, x_t] + b_c) && \text{(candidate state)} \\
c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t && \text{(cell update)} \\
o_t &= \sigma(W_o[h_{t-1}, x_t] + b_o) && \text{(output gate)} \\
h_t &= o_t \odot \tanh(c_t), && \text{(hidden state)}
\end{aligned}$$

where σ is the sigmoid function and $\odot$ denotes elementwise multiplication.

In terms of design, LSTMs improve the flow of gradients, because the cell state c_t

provides an almost linear path for gradients, alleviating vanishing gradients and allowing better long-range modeling. Also, gates dynamically control which information is stored, updated, or discarded, so important signals from the past can more effectively influence current predictions.

Yet, LSTMs retain core RNN drawbacks: computation is still sequential in time, limiting GPU parallelization, and modeling very long contexts remains challenging, especially when dependencies skip over many intermediate tokens.

Transformers introduce a decisive shift by replacing recurrence with *self-attention*, allowing massive parallelization and more flexible context modeling (Vaswani et al., 2017). Instead of propagating information solely through a chain of hidden states, self-attention allows each token to directly attend to all others in the sequence.

Given input representations $X = [x_1, \ldots, x_T]$, a self-attention layer computes queries Q, keys K, and values V:

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V,$$

thus the attention output is:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V.$$

This mechanism assigns dynamic, context-dependent weights to all input positions, allowing the model to *focus* on the most informative tokens at each step of generation or prediction. Multi-head attention repeats this process with multiple learned projections, letting the model capture different types of relationships in parallel.

Transformers complement attention with architectural components such as *positional encodings*: since recurrence is removed, explicit position information is added to token embeddings.

This design offers several advantages over RNNs/LSTMs: mostly, this is related to *parallelization*: all tokens in a sequence can be processed simultaneously, making Transformers highly efficient on GPUs. In addition to that, any token can attend directly to any other, so modeling distant relationships does not require stepping through many recurrent transitions.

The progression from RNNs to LSTMs to Transformers reflects a sequence of trade-off rebalancing:

1. RNNs introduced recurrent hidden states for sequence modeling, but struggled with vanishing gradients and long-range credit assignment.

2. LSTMs improved temporal credit assignment via gating and cell states, extending the effective memory of recurrent models, yet remained inherently sequential and limited in exploiting modern parallel hardware.

3. Transformers broke away from recurrence entirely, using self-attention to handle long-range dependencies and to fully leverage parallel computation. Residual connections and layer normalization further enhanced *trainability* at scale.

From a more practical point of view, the impact of LLM-based chatbots adhering to the Transformer paradigm shows up in two major ways. First, they are highly flexible in style. They do much more than answer technical questions: they can take on requested personas or teaching styles, rely on analogies, and produce many kinds of conceptual or creative content, including program code, structured outlines, or even sketches for music. This range of behaviors demonstrates that once the patterns and mechanics of language are captured well enough in a model, the machine can be used to support and structure human learning of those same mechanics. It can act as a scaffold that helps users practice, explore, and refine their understanding of language and related skills (Brown et al., 2020; Le Scao et al., 2022).

Second, they function as interactive language tutors. They can read and comment on user-provided text, point out mistakes, and propose improved wording or new vocabulary. This enables a learning-by-doing paradigm, where users get immediate, tailored feedback in natural language. Indeed, the model becomes a dynamic editor and coach, helping users iteratively improve their writing and linguistic competence.

Yet, this design also imposes limits on structural plasticity—that is, on the model's ability to change its internal representations in response to new information. This tension motivates research into new architectures and training approaches that could combine the robustness of current systems with mechanisms for continuously and reliably integrating new knowledge over time (Behrouz et al., 2025). Due to that, the question about how it is really possible to ”interpret” or ”explain” the output of a Large Language Model remains open. In this context, *reasoning models* have recently emerged as a promising direction to at least structure, if not fully resolve, this interpretability problem.

Reasoning Models

Reasoning-oriented LLMs, unlike purely generative ones, are explicitly trained or prompted to produce intermediate steps (often called *chain* of thought) before emitting their final answer. The central intuition is that, by making the internal decision

process more external and sequential, we obtain artifacts that humans can inspect, critique, and potentially verify. Instead of mapping an input directly to an output in a single opaque jump, the model is encouraged to: decompose the task into subproblems (e.g., identifying assumptions, extracting relevant facts, outlining a plan), generate explicit intermediate inferences (logical steps, sub-calculations, references to external tools or data sources) and finally reach a conclusion that is, at least to some extent, traceable back through these intermediate steps.

From an interpretability point of view, this moves part of the model's "reason" from its high-dimensional internal representations into a symbolic linear form. The resulting traces do not fully explain why the network's internal activities evolved the way they did, but they provide a human-readable proxy for the model's decision process.

Concretely, reasoning models attempt to address the interpretability issue along at least three dimensions:

- Transparency of process rather than only outcome. Standard LLMs provide answers that are often correct but bare: we see the conclusion without any narrative of how it was reached. Reasoning models instead expose a step-by-step process, offering users a structured object to scrutinize. This can support post-hoc explanation ("the model claims it used these premises and these inferential steps") even if it remains a behavioral, rather than mechanistic, account.

- Verifiability and error localization. When intermediate steps are explicit, users (or automated checkers) can verify them individually. Errors can be localized to specific stages—e.g., a mistaken assumption, a misapplied rule, or an arithmetic error—rather than being attributed to a monolithic black-box failure. This granularity can make the model's behavior more intelligible, even if it does not fully expose the underlying representation dynamics.

- Many reasoning models are trained or prompted to distinguish between an internal "reasoning phase" and an external "answer phase". This separation encourages the model to prioritize correctness and coherence in its internal steps, while preserving fluency and concision in the final response. Theoretically, this separation can make it easier to analyze and compare reasoning traces across different inputs, or to constrain them via external rules, thereby partially lifting the veil on how outputs are produced.

However, these advantages come with important caveats. Reasoning traces are generated text and thus remain susceptible to fabrication and post-hoc rationalization: the

model can produce plausible-looking chains of thought that do not correspond to its actual internal computation. In addition, the presence of an explicit reasoning trace does not eliminate the underlying opacity of the network's learned representations.

In short, reasoning models do not conclusively solve the problem of how to *interpret* or *explain* LLM outputs. Instead, they propose a pragmatic compromise by externalizing portions of the model's decision process into structured, human-readable steps. This shift from raw answers to answer-plus-reasoning can make model behavior more inspectable, auditable, and, in some cases, more aligned with human notions of explanation, even while the deeper question of their internal semantics remains unsettled. More recently, another new paradigm tried to address the problem of making Large Language Models more flexible and, to some extent, explainable: Nested Learning.

Nested Learning

In standard pipelines, models are trained offline on massive corpora and then deployed with frozen parameters. While retrieval-augmented generation and fine-tuning offer two routes to adaptivity, they present trade-offs. Retrieval externalizes knowledge but does not update core competence; fine-tuning updates parameters, but it is costly and risks eroding prior skills (catastrophic forgetting) unless carefully regularized and evaluated. More fundamentally, both approaches leave the discrete dichotomy between a training phase and an inference phase untouched, which is at odds with the continual nature of human learning (Goodfellow et al., 2016).

Nested Learning (NL), a new paradigm suggested in 2025, proposes a unifying perspective: it represents a model and its training procedure as a system of *nested*, potentially parallel optimization problems, each operating over its own context and at its own update frequency. Conceptually, NL arranges modules along a timescale hierarchy—from highly plastic short-term components to increasingly stable long-term components—coordinated by a *continuous memory system* that routes and consolidates information. This transforms learning from a single offline event into a continuous, structured process during the operational life of the model (Behrouz et al., 2025).

Formally, NL asserts that *architecture and optimization are not separable concerns*: the learning algorithm (optimizer) and the network blocks co-define the gradient flows and the compression of information over time, suggesting architecture-specific optimizers and explicit mechanisms for knowledge transfer between levels. Under this lens, familiar concepts—meta-learning, in-context learning, hyper-networks, and recurrent mechanisms—become special cases of cross-level information exchange in a unified

framework.

Within NL, classic Transformers can be reinterpreted as systems with extreme update polarization: attention operates at near-instantaneous frequency (adapting across tokens within a forward pass) while the main parameters remain fixed post-training. The absence of intermediate timescales impedes the persistent consolidation of new knowledge, confining adaptation to transient context effects. NL, on the contrary, inserts intermediate levels that continuously compress, stabilize, and propagate newly acquired information.

A corollary of NL is that different modules—serving different roles and timescales—should not necessarily share a single, uniform update rule. *Delta Gradient Descent* (DGD) has been proposed as an optimizer family whose updates depend not only on the prediction error but also on the internal state of the module, allowing trajectories to be adjusted to function and timescale. In practical terms, DGD constitutes a step towards optimizers that are *aware* of where they operate within the hierarchy, thus leveraging the structure of NL rather than fighting against it (Prados et al., 1989).

Preliminary experiments (e.g., with an NL-styled architecture named *Hope*) report improved adaptation and generalization on benchmarks that require learning under distribution shift and evolving task rules—settings where single-phase training falls short. Although broader validation and open baselines are needed, these results illustrate how rethinking learning as nested optimization can reduce the dependency on frequent and costly model re-training - while allowing more autonomous and personalized systems.

For relevant applications, such as in the finance domain, decision support—NL's promise is twofold. Firstly, by distributing plasticity, one can limit the risk of catastrophic interference while still incorporating fresh knowledge, enforcing governance on *where* and *how fast* adaptation occurs. Secondly, by aligning optimizers to module roles, the compute footprint of updates can be reduced and the sample efficiency relative to repeated end-to-end fine-tuning cycles can be increased. In human–AI interaction, this could lead to assistants that persistently internalize validated corrections and domain feedback, rather than repeatedly "re-teaching" them via prompts or fine-tuning.

At the same time, practical deployment raises open questions: (i) auditability and versioning of knowledge across timescales; (ii) safety and alignment when persistent updates can occur during usage; (iii) validation protocols that separate transient context performance from genuine competency gains; and (iv) operational triggers for promoting short-term adaptations to longer-term memory. Addressing these requires design patterns and tooling that expose NL's internal gradient flows to monitoring and

control.

Nested Learning frames models as hierarchies of coupled optimization processes across multiple timescales, advocating co-design of modules and optimizers (e.g., DGD) and providing paths to mitigate *catastrophic forgetting* (that is the loss of crucial information to host fresher, but less important one) while preserving stability. If validated on a bigger scale, NL could shift engineering practice from recurrent offline re-training to principled online consolidation, bringing AI a step closer to persistent and interpretable adaptation.

2.3 Graphs

Information can be represented in many ways; one of them is through a graph-like structure with a set of objects and the connection between them. A graph represents relations, called *edges* or *links*, between a collection of entities, called *nodes* or *vertices*. Mathematically, a graph G is defined as a tuple of a set of vertices V and a set of edges. Each edge is a pair of two vertices and represents a connection between them.

A graph can be further described as either directed or undirected. An undirected graph represents a relationship that is bidirectional. A way of visualizing the connectivity and directionality of a graph is through its adjacency matrix. A matrix-based representation of the vertices of the graph is shown in Figure 2.5:

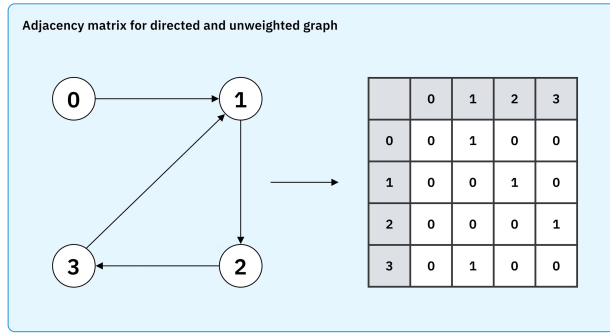


Figure 2.5: Graph and its adjacency matrix. Source: Noble, J. (2026) *What is a graph neural network*. IBM. Available at: https://www.ibm.com/think/topics/graph-neural-network

The adjacency matrix of Figure 2.5, indicates all the vertices and the direction of the edge that creates them. For example, node 0 in the directed graph connects to node 1, but the inverse is not true due to the directionality of the connection. Furthermore, the adjacency matrix could also represent different weights through connections by adding different values to the matrix, instead of being a binary relationship. Different types of analysis can be performed on graphs, each providing insights into different elements of data.

In a *graph-level task*, the model predicts a property of an entire graph. This type of pattern recognition can be framed as a form of graph classification because it classifies the entire graph. An example could be, on a social network, it could be interesting to

predict whether a group of individuals (with some relationships existing among them) is likely to be associated with a particular institution such as a university or college.

Node-level tasks are concerned with predicting the identity or role of each node within a graph. For instance, a node classification problem in a social network dataset might be to predict whether a user is likely to have a specific interest based on their friend network.

Edge-level tasks are concerned with predicting the connection between nodes within a graph. An example is image scene understanding. After identifying objects in an image, deep learning models can also predict the relationship between them. The nodes represent the objects in the image, and the prediction indicates which of these nodes shares an edge or what the value of that edge is.

Graph modeling

Graph Neural Networks, or GNN, are deep learning models designed to capture relationships, dependencies, and interactions between entities in graph-structured data. Graph Neural Networks can be built in different ways depending on how they aggregate information and update node representations. One of the most commonly used architectures is the Graph Convolutional Network (GCN).

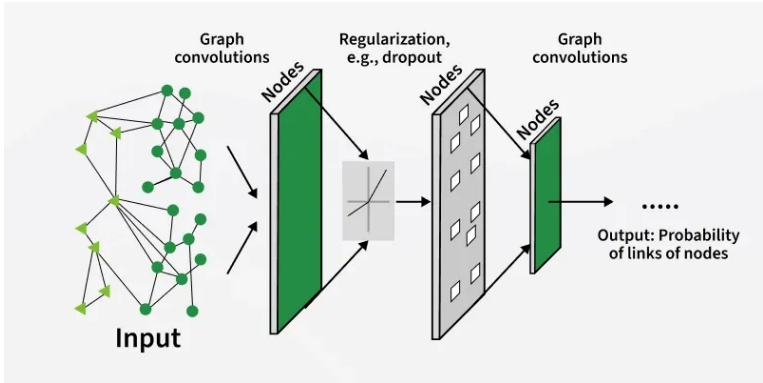


Figure 2.6: GNN Architecture: From input to output. Source: GeeksforGeeks (2025) *What are graph neural networks?* Available at: https://www.geeksforgeeks.org/deep-learning/what-are-graph-neural-networks/

As a sample of how a GNN can be structured, consider a simple GCN for classifying an entire graph. A basic GCN usually contains three main layers:

- *Convolutional Layer*: Aggregating features, this layer performs the convolution on each node to learn its connection.
- *Activation Layer*: This layer applies an *activation function* to the output of the convolution.
- *Output Layer*: This final layer sums the outputs, producing the final prediction for the graph.

First, a convolution is performed using each node in the convolution layer graph. This convolutional layer uses feature information from the neighbors of each node and aggregates it, updating the weights associated with each node. Then, the nonlinear activation function is applied to the output of the convolution layer. To reach the best accuracy, a network can use multiple convolutions and nonlinear activation layers stacked together. Finally, an output linear layer is used to predict which class a graph is most likely to belong to.

Moving on with this overview, let us observe that Graph Neural Networks can be designed to process graph-structured data in different ways.

Graph Attention Networks (GAN)

A Graph Attention Network is designed to update each node by aggregating information from its neighbors. Unlike other models, it learns how much weight to give to each neighbor. In addition to the already presented layers from the GCN, GATs present also: Attention layer and Softmax Normalization layer. The attention scores decide how important each node is to another when aggregating information. We can then aggregate and apply an activation function, just as an GCN would.

Graph Attention Networks offer diverse useful properties like computational efficiency: the computation of both attentional and aggregation coefficients can be parallelized across all nodes. Moreover, GATs do not depend on the global graph structure, as they have an edge-wise mechanism that attends the neighborhoods and can assign different weights to the different neighbors based on their relevance.

Unlike Graph Convolutional Networks (GCNs), which aggregate neighboring features using fixed or degree-based normalization schemes, GATs allow the model to adaptively weight each neighbor's contribution. This mechanism not only increases model accuracy, but it lso provides a potential interpretability advantage: the learned attention weights can be inspected to understand which neighboring nodes most strongly influence a given prediction (Veličković et al., 2018).

Graph Neural Networks and Explainability

Graph Neural Networks are powerful tools that can provide accurate predictions. Given their black-box nature, explaining the resulting GNN predictions is a challenge. GNNExplainer is a model-agnostic approach, meaning that it queries the model without needing any further internal information, that provides interpretable explanation for any GNN model. The explainer returns a small subgraph together with a subset of node features that are the most influential for the analyzed prediction. As for its model-agnostic nature, this approach can be used for any Machine Learning task, as discussed in (Ying et al., 2019).

It is worth observing that, together with being an *object of modeling and interpretation*, a graph could also be used as a *tool for interpreting a model*. More specifically, a technique representing *feature graphs*, starting from a rule-based model, allowing not only to determine *ranking of input features*, but also a *graphical representation of feature interactions*, according to how much they contribute to the ruleset, has been recently introduced (Sirocchi et al., 2025).

Feature Graphs

Starting from a layered rule representation introduced in (Liu et al., 2015), we consider three sets of nodes: features, rules, and classes. In our approach, connections between feature nodes and rule nodes are weighted edges, indicating the contribution of each feature to the corresponding rule (*feature relevance*). Similarly, connections between rule nodes and class labels are weighted edges, representing the contribution of each rule to class prediction (*rule relevance*). We propose a projection strategy to map this tripartite graph onto the feature set such that edges between features reflect their shared contribution to the same rules, and the centrality of each feature reflects its overall importance across all rules and serves as a feature importance metric. We extend this strategy to construct class-specific feature graphs and define a distance metric to compare graphs.

Let $\mathcal{D}$ represent a dataset comprising d samples denoted by $\boldsymbol{x}_s$, with s from 1 to d. Each sample is described by m input features, defined over the feature set $\mathcal{V} = \{v_1, v_2, \ldots, v_m\}$. For each input $\boldsymbol{x}_s$, y_s denotes the corresponding target. In classification tasks, the target takes discrete values in $\mathcal{T} = \{t_1, t_2, \ldots, t_r\}$.

A rule set $\mathcal{R}$ can be defined over $\mathcal{D}$, mapping instances to targets, and consisting of a set of rules each denoted by R. If $\mathcal{R}$ has n rules, then $\mathcal{R} = \{R^1, R^2, \ldots, R^n\}$. Each R is a logical expression where the antecedent or premise of the rule is a set of conditions over the features of the dataset, while the consequent is the outcome (here

class assignment) when all conditions specified by the antecedent are met. Formally, a rule R^k can be denoted as a pair $R^k = (C^k, T^k)$, with $C^k = \{c_1^k, c_2^k, \ldots, c_q^k\}$ and $T^k \in \mathcal{T}$, where C^k denotes the set of q conditions in the rule and T^k is the target associated with that rule, i.e.:

$$c_1^k \wedge c_2^k \wedge \cdots \wedge c_q^k \implies T^k$$

Let $\mathcal{I}_\square$ be a function that computes the relevance of a feature v for a rule R in a dataset $\mathcal{D}$. For a rule set $\mathcal{R}$, a feature relevance matrix $\boldsymbol{P}$ can be defined as the $n \times m$ matrix, with n the number of rules in $\mathcal{R}$ and m the number of features, of the elements $[p_{ij}]$, where:

$$p_{ij} = \mathcal{I}_\square(\mathcal{D}, v_i, R^j) \quad \forall v_i \in \mathcal{V},\ \forall R^j \in \mathcal{R}. \tag{2.19}$$

Additionally, let $\mathcal{I}_\triangledown$ be a function that computes the relevance of a rule R in a rule set $\mathcal{R}$ over a dataset $\mathcal{D}$. For a rule set $\mathcal{R}$, a rule relevance vector $\boldsymbol{q}$ of length n can be defined with elements q_j, where

$$q_j = \mathcal{I}_\triangledown(\mathcal{D}, R^j) \quad \forall R^j \in \mathcal{R}. \tag{2.20}$$

Graph Projection. Let $\boldsymbol{A}$ be a $m \times m$ matrix, with m the number of features, of the elements $[a_{ij}]$ such that:

$$a_{ij} = 1 - \prod_{k=1}^{n} (1 - p_{ki} \cdot p_{kj} \cdot q_k) \quad \forall i, j \in \{1, \ldots, m\} \tag{2.21}$$

$\boldsymbol{A}$ is then normalised such that the sum of all its elements is 100, to obtain $\boldsymbol{A}'$:

$$A'_{ij} = \frac{A_{ij}}{\sum_{i,j=1}^{m} A_{ij}} \cdot 100 \quad \forall i, j \in \{1, \ldots, m\}$$

$\boldsymbol{A}'$ is then the adjacency matrix of the weighted and undirected feature graph, mapping feature interactions within the rule set $\mathcal{R}$ over the dataset $\mathcal{D}$.

According to the proposed projection strategy, the product $p_{ki} \cdot p_{kj} \cdot q_k$ in 2.21 captures the joint relevance of features v_i and v_j with respect to rule R^k, scaled by q_k to also account for the relevance of the rule. These contributions are aggregated across all rules in a multiplicative manner, making the overall score more sensitive to instances where two features exhibit high joint relevance in at least one relevant rule, in contrast to simple summation. In fact, in rule sets, it is not expected that two features interact as strongly across all rules, and strong interactions can be noteworthy,

even if infrequent. Moreover, this projection strategy generates self-edges a_{ii} for each feature i in $\{1, \ldots, m\}$, quantifying its individual contribution across all rules. These self-edges also account for instances where a feature appears alone in a rule, which are often crucial to the rule set but are lost in most projection strategies.

Class-specific graph projection. A feature graph specific for a given class $t \in \mathcal{T}$ can be constructed by considering only rules having the given class as consequent. Let $\mathcal{R}_i$ be the subset of rules in $\mathcal{R}$ with target t_i, i.e. $\mathcal{R}_i = \{R^k \mid R^k \in \mathcal{R} \wedge T^k = t_i\}$. $\boldsymbol{A}'_i$ is the adjacency matrix defined as in 2.21 but where the matrix $\boldsymbol{P}$ and the vector $\boldsymbol{q}$ are defined over the rule set $\mathcal{R}_i$ rather than $\mathcal{R}$.

Graph distance. The distance between two graph representations can be computed as the distance between the respective adjacency matrices $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$:

$$d(\boldsymbol{A}_1, \boldsymbol{A}_2) = \|\boldsymbol{A}_1 - \boldsymbol{A}_2\|_F$$

where the Frobenius norm $\|\boldsymbol{A}\|_F$ of a matrix $\boldsymbol{A}$ is given by:

$$\|\boldsymbol{A}\|_F = \sqrt{\sum_{i,j=1}^{n} |A_{ij}|^2}$$

Feature importance. A feature importance score can be computed for features in $\mathcal{V}$ as the degree centrality of the nodes of the graph defined by $\boldsymbol{A}'$. Specifically, the importance of v_i is given by the sum of the elements in the i-th row of $\boldsymbol{A}'$:

$$\text{Importance}(v_i) = \sum_{j=1}^{m} A'_{ij} \tag{2.22}$$

This metric aggregates contributions from both self-edges and edges with other features, capturing both the independent and combined feature contributions.

Relevance metrics. Feature and rule relevance metrics, proposed in LLM and adopted in this example, leverage the concepts of *error* and *covering* based on the fraction of data samples assigned to a class and satisfying a rule. These metrics have already been defined in Section 1.2.

Moreover, adapting from (Ferrari et al., 2023), let R^k_{-h} be the rule obtained from R^k by removing the conditions on feature v_h, i.e.:

$$R^k_{-h} = (C^k_{-h}, T^k), \quad C^k_{-h} = \{c^k \mid c^k \in C^k \wedge V(c^k) \neq v_h\}$$

where $V(c^k)$ denotes the feature over which the condition is applied. Then, *feature relevance* can be computed as the increase in error to the rule as a result of removing the conditions over the feature.

The proposed approach was first evaluated on synthetic datasets to showcase the advantages of graph feature representation with respect to feature importance scores, particularly in terms of the ability to differentiate between features that are predictive of the target independently or combined. Three types of datasets were generated: (1) with all relevant features independently predictive of the target, (2) with all relevant features predictive to the target when combined, and (3) with a mix of independently and combined predictive features. Each dataset comprised 2000 instances and 8 features, with relevant features varying from 2 to 6, and 10 datasets were generated per configuration using different random seeds. All features were uniformly sampled in the range [0,1]. Independent predictive features were obtained by assigning a set of intervals in the range [0,1] to each feature. For each data sample, the target was set to 1 if the value of at least one feature fell within its corresponding predefined interval. In contrast, combined predictive features were generated by setting the target variable to 1 if the majority of the feature values in the data sample exceeded a given threshold. These two strategies were also combined to obtain datasets with some independent and some combined relevant features. For further implementation details, refer to the GitHub repository for reproducing the synthetic datasets.

The rule-based classifier adopted in this evaluation was DT, with experiments also conducted using LLM, yielding similar results (not shown). DTs were trained using 5-fold nested cross-validation with hyperparameter tuning and converted to rule sets by translating each path from the root to the leaves into if-then rules. Feature graphs were constructed from these rule sets using the proposed method, with adjacency matrices visualized as heatmaps. The Gini importance was also computed, serving as a standard feature importance metric for comparison.

The effectiveness of feature centrality as a measure of feature importance, assessed by its ability to identify the top-k features in a dataset, was evaluated across 15 benchmark datasets. Our graph-based importance score was compared against three feature importance metrics: permutation importance, Gini importance, and average SHAP values. DTs were trained using 5-fold nested cross-validation to derive rule sets and

generate feature graphs. Evaluation criteria encompassed both performance and robustness of the feature importance scores. Performance evaluation involved selecting the top 5 and top 10 features identified by each metric, training decision trees on these features, and computing prediction accuracy. Robustness was assessed by calculating the average pairwise Spearman's correlation across cross-validation folds for each importance score.

The evaluation of the proposed approach on synthetic datasets underscores the superiority of a graph-based feature representation over a one-dimensional feature importance score to uncover the collective roles of features in class prediction. Figure 2.7 presents the average adjacency matrix of feature graphs built on rule sets derived for synthetic datasets with varying numbers of relevant features and feature configurations. For independently predictive features (first row), the heatmap reveals heavy weights on the diagonal (self-edges), indicating that each rule primarily relies on a single highly predictive variable. Connections between different features show weaker weights, suggesting minimal influence from other variables. In contrast, combined predictive features (third row) exhibit heavier weights on edges connecting different features, indicating collaborative predictive power among multiple features.

While the Gini importance effectively distinguishes relevant from irrelevant features, it fails to differentiate between features that are predictive independently or combined. Intermediate scenarios, comprising a mix of these two types of features, were also explored, yielding similar insights. These results underscore the advantage of our feature graph approach, providing a more nuanced representation of feature interactions.

2.4 Financial Use Case

Let us consider a simple 2-layer GAT implementation applied to the Elliptic transaction dataset. Such dataset maps Bitcoin transactions to real entities belonging to legitimate categories (exchanges, wallet providers, miners, legitimate services, etc.) versus illicit ones (scams, malware, terrorist organizations, ransomware, Ponzi schemes, etc.), consequently serving as a benchmark for anti-money laundering research [4]. The proposed modeling pipeline aims to exemplify the steps needed to deal with this kind of problem, without heavily focusing on performance optimization. From this point of view, a more comprehensive discussion is proposed in (Nimmagadda et al., 2025).

```
1 # Imports
```

[4] www.elliptic.co

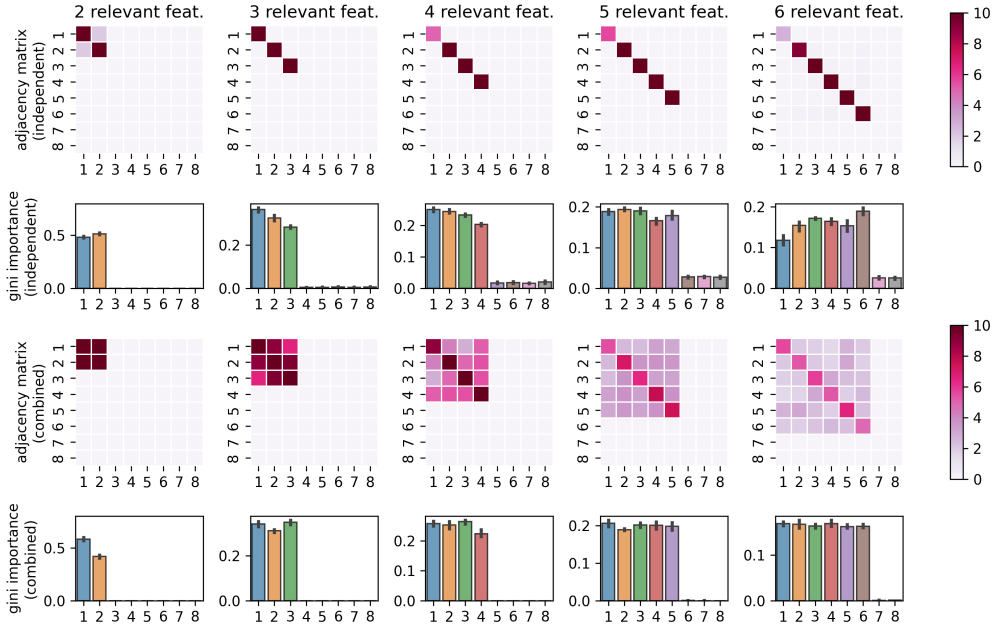


Figure 2.7: Adjacency matrices constructed according to the proposed approach and Gini importance indices obtained from decision trees trained on synthetic datasets comprising a number of relevant features ranging from 2 to 6, which are predictive of the target class either independently or combined. Source: (Sirocchi et al., 2025).

```
import torch
import torch.nn.functional as F
from torch_geometric.nn import GATConv
from torch_geometric.explain import Explainer, GNNExplainer
from torch_geometric.explain.config import ModelConfig
import matplotlib.pyplot as plt
import networkx as nx
import pandas as pd
import numpy as np
from torch_geometric.data import Data
from sklearn.metrics import accuracy_score, precision_score, recall_score,
    f1_score, balanced_accuracy_score, classification_report
```

```
# Import Elliptic Dataset
base_path     = r"C:\Users\xxx\DATASET"
classes_path  = base_path + r"\elliptic_txs_classes.csv"
edges_path    = base_path + r"\elliptic_txs_edgelist.csv"
features_path = base_path + r"\elliptic_txs_features.csv"

```

```
# Read dataset
classes_df  = pd.read_csv(classes_path)
edges_df    = pd.read_csv(edges_path)
features_df = pd.read_csv(features_path, header=None)

# Attributes rename
feature_cols = ['txId', 'time_step'] + [f'f_{i}' for i in range(features_df.
    shape[1] - 2)]
features_df.columns = feature_cols
```

Node labels are merged with the feature matrix through the transaction identifier. The original Elliptic labels are then mapped into a numerical target vector, where illicit transactions are assigned one class, legitimate transactions the other, and unknown transactions are marked with -1 so that they can remain in the graph without contributing to the supervised loss. In other words, only labeled nodes are split into training and test subsets, while unlabeled nodes remain part of the graph structure.

```
# Features and classes merge
df = features_df.merge(classes_df, on='txId', how='left')

# Class mapping: '1' = illicit , '2' = licit , 'else' = no label
def map_class(x):
    if str(x) == '1':
        return 0
    elif str(x) == '2':
        return 1
    else:
        return -1

df['y'] = df['class'].apply(map_class)

# Feature matrix
x = torch.tensor(
    df.drop(columns=['txId', 'time_step', 'class', 'y']).values,
    dtype=torch.float
)

y = torch.tensor(df['y'].values, dtype=torch.long)

# txId mapping
txid_to_idx = {tx_id: i for i, tx_id in enumerate(df['txId'].values)}

edges_filtered = edges_df[
    edges_df['txId1'].isin(txid_to_idx) & edges_df['txId2'].isin(txid_to_idx)
].copy()

edge_index = torch.tensor(
    [
        edges_filtered['txId1'].map(txid_to_idx).values,
        edges_filtered['txId2'].map(txid_to_idx).values
    ],
```

```
    dtype=torch.long
)
```

```
# Train-test mask
labeled_idx = np.where(df['y'].values != -1)[0]

np.random.seed(42)
np.random.shuffle(labeled_idx)

train_size = int(0.8 * len(labeled_idx))
train_idx  = labeled_idx[:train_size]
test_idx   = labeled_idx[train_size:]

train_mask = torch.zeros(len(df), dtype=torch.bool)
test_mask  = torch.zeros(len(df), dtype=torch.bool)

train_mask[train_idx] = True
test_mask[test_idx]   = True

data = Data(x=x, edge_index=edge_index, y=y)
data.train_mask = train_mask
data.test_mask  = test_mask

num_features = data.num_features
num_classes  = 2
```

A two-layer Graph Attention Network model is used, with multi-head attention in the first layer and a final classification layer producing log-probabilities for the two classes. The first layer employs multi-head attention, where multiple attention heads compute neighborhood aggregation, and their results are concatenated. The second layer serves as the output layer and uses a single attention head to compute class scores for each node.

```
# GAT model
class GAT(torch.nn.Module):
    def __init__(self):
        super().__init__()
        # First GAT layer: input features -> 8 features per head, 8 attention
    heads
        self.gat1 = GATConv(num_features, 8, heads=8, dropout=0.6)
        # Second GAT layer: hidden features -> number of classes
        self.gat2 = GATConv(8*8, num_classes, heads=1, concat=False, dropout
    =0.6)

    def forward(self, x, edge_index):
        # Dropout on input features
        x = F.dropout(x, p=0.6, training=self.training)
        x = F.elu(self.gat1(x, edge_index))
        x = F.dropout(x, p=0.6, training=self.training)
        # log probabilities for each class
        x = self.gat2(x, edge_index)
```

```
        return F.log_softmax(x, dim=1)

model = GAT()
```

```
train_y = data.y[data.train_mask]

num_illicit = (train_y == 0).sum().item()
num_licit   = (train_y == 1).sum().item()

class_weights = torch.tensor(
    [1.0 / num_illicit, 1.0 / num_licit],
    dtype=torch.float,
    device=data.x.device
)

class_weights = class_weights / class_weights.sum() * 2.0
```

During each training iteration, the network computes the loss on the training nodes, back-propagates the error through the attention mechanism, and updates the parameters accordingly.

```
# Define optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=0.005, weight_decay=5e-4)

# Train the model
model.train()
for epoch in range(200):
    optimizer.zero_grad()
    out = model(data.x, data.edge_index)
    loss = F.nll_loss(out[data.train_mask], data.y[data.train_mask], weight=
    class_weights)
    loss.backward()
    optimizer.step()
```

After training, model performance is evaluated on the test set by computing a set of performance metrics: accuracy, balanced accuracy, precision, recall and F1-score. The model can now predict the label of any node in the graph by considering both its own features and the features of its neighbors.

```
model.eval()
out = model(data.x, data.edge_index)
pred = out.argmax(dim=1)

y_true = data.y[data.test_mask].cpu().numpy()
y_pred = pred[data.test_mask].cpu().numpy()

acc = accuracy_score(y_true, y_pred)
bal_acc = balanced_accuracy_score(y_true, y_pred)
precision = precision_score(y_true, y_pred, average='binary', pos_label=0,
    zero_division=0)
recall = recall_score(y_true, y_pred, average='binary', pos_label=0,
    zero_division=0)
```

```
f1 = f1_score(y_true, y_pred, average='binary', pos_label=0, zero_division=0)
```

```
classes = ['illicit', 'licit']

def predict_node(node_id):
    model.eval()
    out = model(data.x, data.edge_index)
    prediction = out[node_id].argmax().item()
    tx_id = df.iloc[node_id]['txId']
    return {"node_id": node_id, "txId": tx_id, "predicted_class": classes[
    prediction]}

# Prediction Example
predict_node(10)
```

Let us now focus on expalinability and apply the GNNExplainer. The explainer needs to be initialized, to set it up for the GAT trained model.

```
from torch_geometric.explain import Explainer, GNNExplainer
from torch_geometric.explain.config import ModelConfig
import matplotlib.pyplot as plt
import networkx as nx
```

```
# GAT model
class GAT(torch.nn.Module):
    def __init__(self):
        super().__init__()
        # First GAT layer: input features -> 8 features per head, 8 attention
    heads
        self.gat1 = GATConv(num_features, 8, heads=8, dropout=0.6)
        # Second GAT layer: hidden features -> number of classes
        self.gat2 = GATConv(8*8, num_classes, heads=1, concat=False, dropout
    =0.6)

    def forward(self, x, edge_index):
        # Dropout on input features
        x = F.dropout(x, p=0.6, training=self.training)
        x = F.elu(self.gat1(x, edge_index))
        x = F.dropout(x, p=0.6, training=self.training)
        # log probabilities for each class
        x = self.gat2(x, edge_index)
        return x

model = GAT()
```

```
train_y = data.y[data.train_mask]

num_illicit = (train_y == 0).sum().item()
num_licit   = (train_y == 1).sum().item()

class_weights = torch.tensor(
    [1.0 / num_illicit, 1.0 / num_licit],
```

```
    dtype=torch.float,
    device=data.x.device
)

class_weights = class_weights / class_weights.sum() * 2.0
```

```
# Define optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

def train():
    model.train()
    optimizer.zero_grad()
    out = model(data.x, data.edge_index)
    loss  = F.cross_entropy(out[data.train_mask], data.y[data.train_mask],
    weight=class_weights)
    loss.backward()
    optimizer.step()
    return loss
```

```
model.eval()
out = model(data.x, data.edge_index)
pred = out.argmax(dim=1)

y_true = data.y[data.test_mask].cpu().numpy()
y_pred = pred[data.test_mask].cpu().numpy()

acc       = accuracy_score(y_true, y_pred)
bal_acc   = balanced_accuracy_score(y_true, y_pred)
precision = precision_score(y_true, y_pred, average='binary', pos_label=0,
    zero_division=0)
recall    = recall_score(y_true, y_pred, average='binary', pos_label=0,
    zero_division=0)
f1        = f1_score(y_true, y_pred, average='binary', pos_label=0,
    zero_division=0)
```

```
explainer = Explainer(
    model=model,
    algorithm=GNNExplainer(epochs=200),
    explanation_type="phenomenon",
    node_mask_type="attributes",
    edge_mask_type="object",
    model_config=dict(
        mode="multiclass_classification",
        task_level="node",
        return_type="raw",
    ),
)
```

The explainer analyzes a specific node and assigns importance scores to the edges and node features that are most influential in the model's classification. It works by learning masks over edges and features, so that the resulting scores indicate how much each graph component contributes to the predicted class.

```
# Candidate nodes: labeled, correct and illicit
candidate_nodes = torch.where((data.y != -1) & (pred == data.y) & (data.y == 0)
    )[0]

# Direct graph index
src = data.edge_index[0]
dst = data.edge_index[1]
deg = torch.bincount(torch.cat([src, dst]), minlength=data.num_nodes)

candidate_nodes = candidate_nodes[deg[candidate_nodes] >= 2]

if len(candidate_nodes) == 0:
    candidate_nodes = torch.where((data.y != -1) & (pred == data.y))[0]
    candidate_nodes = candidate_nodes[deg[candidate_nodes] >= 2]

# Select candidate node with max degree
best_idx = torch.argmax(deg[candidate_nodes]).item()
node_to_explain = int(candidate_nodes[best_idx])
```

To obtain a meaningful local explanation, a labeled, correctly classified and sufficiently connected node is chosen, so that the explanatory subgraph is informative and visually interpretable. We now generate the explanation for the selected node, using the class predicted by the model as explanation target.

```
target = pred

explanation = explainer(
    data.x,
    data.edge_index,
    target=target,
    index=node_to_explain
)

edge_mask = explanation.edge_mask.detach().cpu().numpy()
```

For visualization purposes, we transform the graph into an undirected graph.

```
from torch_geometric.utils import to_undirected

edge_index_undirected = to_undirected(data.edge_index)
edge_index_ud_np = edge_index_undirected.cpu().numpy()

edge_index_np = data.edge_index.cpu().numpy()
edge_mask = explanation.edge_mask.detach().cpu().numpy()
edge_mask_norm = (edge_mask - edge_mask.min()) / (edge_mask.max() - edge_mask.
    min() + 1e-8)

# Adj undirected graph
adj = {}
for i in range(edge_index_ud_np.shape[1]):
    u = int(edge_index_ud_np[0, i])
    v = int(edge_index_ud_np[1, i])
```

```
    adj.setdefault(u, set()).add(v)
    adj.setdefault(v, set()).add(u)
```

All the direct neighbors of the selected node are considered, to be explained: the explained node becomes the center of our local subgraph.

```
# 1-hop nodes connected to the central node
center = node_to_explain
one_hop = set(adj.get(center, set()))
```

To successfully transfer the edge-mask information of the directed graph into the undirected one, we associate a weight with each undirected edge, using the "maximum" between the directions "$u \to v$" and "$v \to u$".

```
edge_weight_map = {}
for i in range(edge_index_np.shape[1]):
    u = int(edge_index_np[0, i])
    v = int(edge_index_np[1, i])
    w = float(edge_mask_norm[i])
    key = tuple(sorted((u, v)))
    edge_weight_map[key] = max(edge_weight_map.get(key, 0.0), w)
```

The direct neighbors are sorted by the importance of the edge connecting them to the central node and we select a maximum of 10. Then, for each selected direct neighbor, we also select up to two second-level neighbors to enrich the local context of the graph. Such second-level neighbors are arbitrary choices for visualization purposes: this "pruning" phase is performed to show only the most relevant neighbors and to preserve the intelligibility of the graph.

```
neighbors_ranked = []
for n in one_hop:
    key = tuple(sorted((center, n)))
    w = edge_weight_map.get(key, 0.0)
    neighbors_ranked.append((n, w))

neighbors_ranked = sorted(neighbors_ranked, key=lambda x: x[1], reverse=True)

max_1hop = 10
selected_1hop = [n for n, _ in neighbors_ranked[:max_1hop]]

# 2-hop nodes connected to the best 1-hop
selected_nodes = {center} | set(selected_1hop)

max_2hop_per_1hop = 2

for n in selected_1hop:
    second_neighbors = adj.get(n, set()) - {center}
    second_ranked = []

    for m in second_neighbors:
        key = tuple(sorted((n, m)))
```

```
        w = edge_weight_map.get(key, 0.0)
        second_ranked.append((m, w))

    second_ranked = sorted(second_ranked, key=lambda x: x[1], reverse=True)
    best_second = [m for m, _ in second_ranked[:max_2hop_per_1hop]]
    selected_nodes.update(best_second)
```

The final subgraph is built and plotted using the previously selected nodes, the edges between them, and the weights derived from the edge mask.

```
G = nx.Graph()
for n in selected_nodes:
    G.add_node(n)

for u in selected_nodes:
    for v in adj.get(u, set()):
        if v in selected_nodes and u < v:
            key = tuple(sorted((u, v)))
            w = edge_weight_map.get(key, 0.0)
            G.add_edge(u, v, weight=w)

plt.figure(figsize=(9, 7))
pos = nx.spring_layout(G, seed=42, k=0.6)

weights = []
for u, v in G.edges():
    w = G[u][v]["weight"]
    weights.append(1.0 + 6.0 * w)

node_colors = []
for n in G.nodes():
    if n == center:
        node_colors.append("red")
    elif n in selected_1hop:
        node_colors.append("orange")
    else:
        node_colors.append("lightblue")

# Label center + 1-hop
labels = {center: str(df.iloc[center]["txId"])}
for n in selected_1hop:
    labels[n] = str(df.iloc[n]["txId"])

nx.draw(
    G,
    pos,
    node_color=node_colors,
    with_labels=False,
    width=weights,
    node_size=380
)

nx.draw_networkx_labels(G, pos, labels=labels, font_size=8)
```

```
44
45 plt.title(f"Local explanation subgraph for txId {df.iloc[center]['txId']}")
46 plt.show()
```

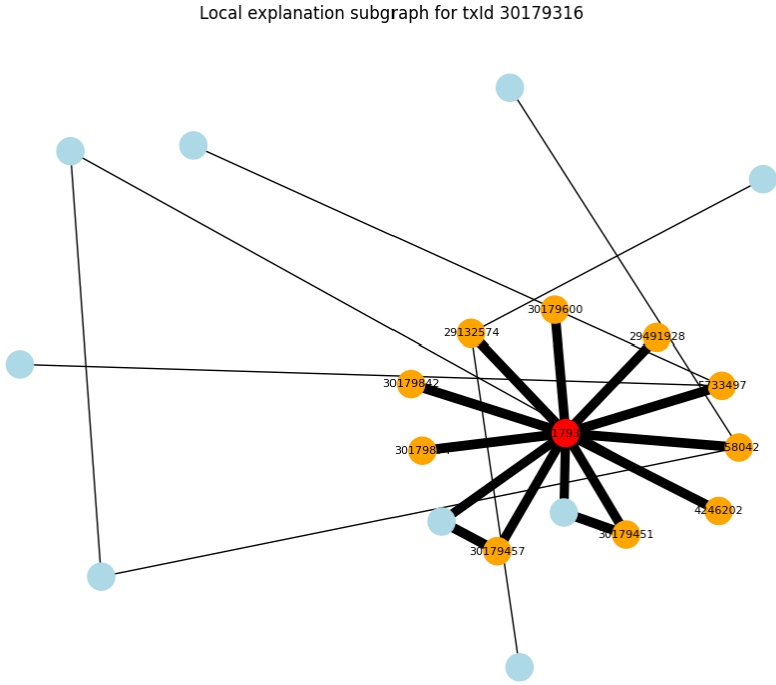


Figure 2.8: Local explanation subgraph for a selected high-degree transaction in the Elliptic dataset

Figure 2.8 shows the selected explanation subgraph. In particular, the red node represents the central node, the orange nodes correspond to the 10 selected 1-hop neighbors, and the light-blue nodes represent the 2-hop neighbors. To keep the explanation more interpretable, the local neighborhood is pruned so that only the most relevant nodes and connections are visualized. Edges are displayed with a thickness proportional to the importance scores assigned by the explainer, making it easier to identify which relations have the greatest influence on the prediction.

3 Optimization

Optimization is the branch of applied mathematics which deals with the selection of the best element from a set of alternatives, according to a specified criterion. In formal terms, an optimization problem seeks to determine a vector of decision variables x that minimizes a *cost function* or maximizes an *objective function* $f(x)$, while adhering to a set of constraints that define the feasible region (Nocedal et al., 2006).

Since these mathematical frameworks underpin modern decision-making systems, a structural classification is essential to contextually understand the challenges and the ways to face them.

3.1 Taxonomy

Optimization problems are categorized into multiple dimensions that directly influence both algorithmic selection and computational complexity (Papadimitriou et al., 1998). A fundamental distinction lies in the domain of the decision variables:

- *Continuous Optimization:* Decision variables are defined in a continuous domain ($x \in \mathbb{R}^n$). These problems typically rely on gradient-based techniques, exploiting the analytical properties of the objective function to locate local or global optima (Nocedal et al., 2006).
- *Discrete and Combinatorial Optimization:* Variables are restricted to integer values or discrete sets ($x \in \mathbb{Z}^n$).

Further classifications distinguish between *Deterministic* and *Stochastic* models—contingent on whether the parameters are fixed or subject to uncertainty—as well as the structural dichotomy between *Constrained* and *Unconstrained* formulations.

Solution Status and Feasibility

The set of all decision vectors x that meet the constraints of an optimization problem is known as the *Feasible Region* (Ω), which is defined by the constraints of the problem.

More specifically, a mathematical optimization problem can be classified into three distinct states according to the characteristics of Ω and the objective function:

- *Feasible:* The feasible region is not empty ($\Omega \neq \emptyset$). There is at least one optimal solution x^* that minimizes (or maximizes) the objective function if the region is also bounded.
- *Infeasible:* Due to the mutual contradiction of the constraints, the feasible region ($\Omega = \emptyset$) is empty. There is no solution that meets all the requirements at once in this situation.
- *Unbounded:* The feasible region allows a boundless improvement. This generally means that a constraint is missing in the model formulation.

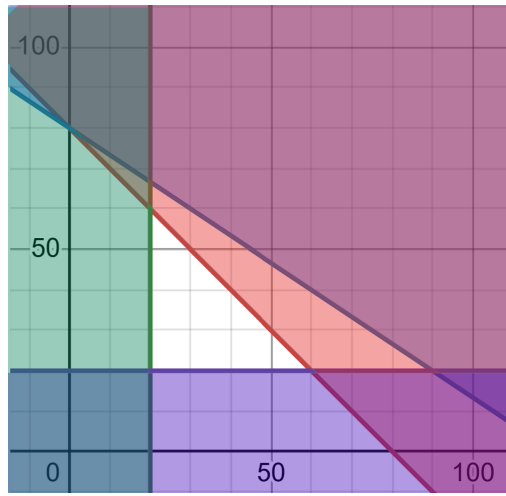


Figure 3.1: Feasible Region

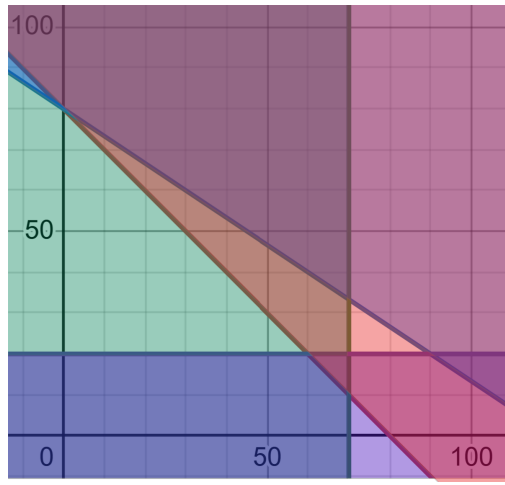


Figure 3.2: Infeasible Problem

3.2 Linear and Non-Linear Optimization

The class of Linear Programming (LP) is characterized by objective functions and constraints that depend linearly on the decision variables. This linearity ensures that the feasible region constitutes a convex polyhedron. The search for an optimal solution (if any) can be restricted to the vertices of this polytope, a property that underpins efficient solution algorithms such as the Simplex method (Dantzig, 1963). Within the linear programming framework, two scenarios can be identified:

- *Integer Linear Programming (ILP):* The linearity assumption is maintained, but all variables are strictly constrained to be integers ($x \in \mathbb{Z}^n$).
- *Mixed-Integer Linear Programming (MILP):* The problem involves a hybrid vector of variables, some constrained to be integers (often binary variables for logical choices) and others continuous ($x \in \mathbb{Z}^p \times \mathbb{R}^{n-p}$).

ILP and MILP are typically categorized as *NP-hard* problems (Papadimitriou et al., 1998), whereas standard LPs can be solved in polynomial time. In order to efficiently explore the search tree, different techniques like Branch-and-Bound are required due to the loss of convexity in the discrete solution space, which precludes the use of gradient-based approaches.

Problem Classification	Variable Type	Difficulty	Domain Example
Linear	Objective and all the constraints are linear functions	Easier	
Nonlinear	At least one constraint or the objective function is not linear	Harder	

Figure 3.3: Linear vs Nonlinear Optimization

When the linearity assumption is relaxed for the objective function or the constraints, the problem enters the domain of Non-Linear Optimization (NLP). The complexity of these models is heavily dependent on the curvature of the underlying functions. Within this domain, the distinction between *Convex* and *Non-Convex* formulations is fundamental (Boyd et al., 2004).

In convex optimization, any local minimum is guaranteed to be a global minimum, allowing efficient resolution. Conversely, non-convex landscapes may contain multiple local optima, ”trapping” gradient-based algorithms, and often requiring global search heuristics or relaxation techniques to identify the true best solution (Nocedal et al., 2006).

To address this, Global Search Heuristics such as *Genetic Algorithms (GAs)* are often employed. Unlike deterministic gradient methods, GAs mimic biological evolution by maintaining a population of candidate solutions that evolve through selection, crossover, and mutation (Goldberg, 1989). This stochastic mechanism allows the solver to escape local basins of attraction, making them essential for non-differentiable or highly multimodal optimization landscapes.

In the following, let us sketch the outline of some of the most popular optimization algorithms.

The Simplex Method

The *simplex method* solves linear programming (LP) problems with continuous variables, linear constraints, and a linear objective. It moves along the vertices (extreme

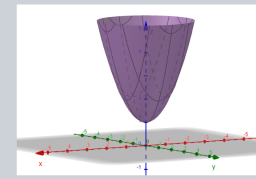
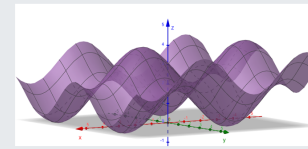
Problem Classification	Variable Type	Difficulty	Problem Example
Convex	Objective function and feasible region are a convex set	Easier, global minimum exists – no local minimum	
Nonconvex	Objective function and/or feasible region are nonconvex	Harder, may have multiple local minimum	

Figure 3.4: Convex vs Nonconvex Optimization

points) of the feasible region to reach an optimal solution. Here is the algorithm outline:

1. Convert the problem to a *standard form.* By this, we mean that the objective is to maximize a linear function of the variables; all constraints are linear inequalities of the form (linear expression) $\leq$ (constant) and all variables are constrained to be nonnegative ($x_i \geq 0$).
2. Build the simplex tableau.
3. *Pivot column*: select the column associated with the most negative reduced cost in the objective row.
4. *Pivot row*: among positive entries in the pivot column, choose the row with the smallest nonnegative ratio RHS / pivot-entry.
5. Perform the pivot to move to a new basic feasible solution.
6. Repeat until there are no negative reduced costs in the objective row (optimality reached).

Let us examine a toy example:

Model. Maximize $4x_1 + 3x_2$ subject to $2x_1 + x_2 \leq 10$, $x_1 + 2x_2 \leq 10$, $x_1, x_2 \geq 0$. In standard form with slack variables s_1, s_2:

$$\max z = 4x_1 + 3x_2 \quad \text{s.t.} \quad 2x_1 + x_2 + s_1 = 10,\ x_1 + 2x_2 + s_2 = 10.$$

Initial tableau.

	z	x_1	x_2	s_1	s_2	RHS
Obj	1	-4	-3	0	0	0
Cons1	0	2	1	1	0	10
Cons2	0	1	2	0	1	10

After a sequence of pivots (details omitted for brevity), the optimal solution is:

$$x_1 = \tfrac{10}{3}, \quad x_2 = \tfrac{10}{3}, \quad s_1 = s_2 = 0, \quad z^* = \tfrac{70}{3}.$$

Cutting Planes

Cutting planes solve integer programs by iteratively solving the LP relaxation and adding linear constraints (cuts) that eliminate fractional solutions while preserving all integer-feasible ones. Here is the algorithm outline:

1. Ignore integrality and solve the LP relaxation.
2. If the solution is integral, stop.
3. Otherwise, generate a valid cut that excludes the current fractional solution.
4. Add the cut to the model and repeat.

Let us analyse a toy example:
Maximize $x_1 + x_2$ subject to $2x_1 + x_2 \leq 3.4$, with x_1, x_2 integer.

1. LP relaxation yields $(x_1, x_2) = (0, 3.4)$ (fractional).
2. Add the cut $2x_1 + x_2 \leq 3$.
3. Resolving as a continuous LP gives $(x_1, x_2) = (0, 3)$, which is integral and optimal.

Branch and Bound

Branch and Bound (B&B) solves optimization problems by recursively splitting them into subproblems (branching), computing bounds to prune suboptimal regions (bounding), and discarding branches that cannot improve the incumbent solution (pruning). A common variant is *Branch and Cut*, which integrates cutting planes into B&B. Let us describe the algorithm in more detail using an illustrative example.

Goal: maximize $4x_1 + 3x_2$ subject to $2x_1 + x_2 \leq 10$, $x_1 + 2x_2 \leq 10$, with x_1, x_2 integer.

1. Root bound of LP relaxation: $z \approx 23.3$ at $(3.3, 3.3)$.
2. Branch on x_2: nodes $x_2 \leq 3$ and $x_2 \geq 4$.
3. Evaluate nodes: feasible integer solutions include $(3, 3)$ with value 21, $(2, 4)$ with value 20, and $(4, 2)$ with value 22.
4. The best integer solution found: $(x_1, x_2) = (4, 2)$ with objective 22 (final incumbent).

Gradient Descent

Gradient Descent (GD) minimizes a differentiable function by iteratively stepping opposite to the gradient.

Given the learning rate $\alpha > 0$ and iterate x_t, we update

$$x_{t+1} = x_t - \alpha \nabla f(x_t).$$

Let us describe the algorithm in more detail using an illustrative example.

Goal: minimize $f(x_1, x_2) = x_1^2 + x_2^2$ with $\alpha = 0.1$, starting from $(3, 2)$.

$$\begin{aligned} \nabla f(x_1, x_2) &= (2x_1, 2x_2), \\ \nabla f(3, 2) &= (6, 4), \\ (x_1, x_2) &\leftarrow (3, 2) - 0.1\,(6, 4) = (2.4, 1.6), \text{ etc.} \end{aligned}$$

The iterations converge to the optimum at $(0, 0)$.

A larger value α accelerates progress per step, but may hinder precise convergence; too large a value can cause divergence, trying to maximize an objective function whose value is commonly referred to as *fitness*.

Iteration	Solution	Gradient
2	(2.40, 1.60)	(4.80, 3.20)
3	(1.92, 1.28)	(3.84, 2.56)
4	(1.54, 1.02)	(3.07, 2.04)
⋮	⋮	⋮
N	(0.00, 0.00)	(0.00, 0.00)

Table 3.1: Solution evolving over different iterations

Simulated Annealing (SA)

Simulated Annealing (SA) is a randomized algorithm that approximates the global optimum of a function. Its stochastic nature implies that repeated runs on the same input may yield different outputs. The method is inspired by physical annealing: heating and slowly cooling a material so that its atoms can rearrange to reduce internal energy. Let us sketch the main steps of the algorithm:

1. Choose a random initial state s.
2. At each step, select a neighboring state s_{next}.
3. Acceptance rule:
   - If $E(s_{\text{next}}) < E(s)$, move to s_{next}.
   - Otherwise, accept s_{next} with probability $P\big(E(s), E(s_{\text{next}}), T\big)$, where T is the current temperature.
4. Track best: maintain the best state s_{best} seen so far.
5. Cooling schedule: gradually decrease the temperature T so that worse moves become less likely over time.
6. Stop: terminate when convergence is detected or a time/iteration budget is reached.

A high *temperature* T makes it more likely to accept upward moves (worse states), helping to escape local minima. As T decreases, the algorithm becomes increasingly greedy, focusing the search near promising regions.

Particle Swarm Optimization (PSO)

Particle swarm optimization (PSO) is a bio-inspired, population-based search algorithm used to find near-optimal solutions in a continuous (or discretized) search space. Unlike gradient-based methods, PSO requires only evaluations of the objective function and does not depend on derivatives.

Inspired by a flock of birds searching for food, each "particle" explores the space while sharing discoveries with the swarm, helping the group converge toward good solutions. PSO is heuristic: it cannot guarantee the true global optimum, but it often finds solutions very close to it in practice. Let us describe it in more detail through an example.

Let $\mathbf{x}$ be a vector of decision variables (e.g., a point (x, y) in the plane), and let $f(\mathbf{x})$ be the function to minimize. The PSO algorithm returns the parameter $\mathbf{x}$ it found that produces a (near) minimum of f.

As an illustrative (non-convex) example in two variables:

$$f(x, y) = (x - 3.14)^2 + (y - 2.72)^2 + \sin 3x + 1.41 + \sin(4y - 1.73). \tag{3.1}$$

Because this objective is not convex, a local minimum is not necessarily the global minimum. Let us introduce these definitions:

- *Energy function* $E(s)$: the function to be minimized (or maximized with sign change).
- *State space:* the domain of $E(\cdot)$; a *state* is any element in this domain.
- *Neighboring state:* a state "close" to the current state under a chosen neighborhood structure

Let us mention the main steps of the algorithm:

1. Initialize: place multiple random points (particles) in the search space.
2. Explore: let each particle move, updating its position and velocity.
3. At each step, a particle considers: (i) its own best position found so far (personal best), and (ii) the swarm's best position found so far (global or neighborhood best).
4. Iterate: repeat exploration and updates for a fixed number of iterations or until a stopping criterion is met.

5. Result: take the swarm's best-known position as the algorithm's estimate of the minimum value of the function.

Genetic Algorithms

Genetic Algorithms (GA) are population-based metaheuristics inspired by natural evolution. They evolve a population of candidate solutions via operators such as *selection*, *crossover* or *mutation* until convergence or a stopping criterion is met.

Below is an example-based explanation of how selection, crossover, and mutation work in a genetic algorithm. Let us suppose we want to maximize the function:

$$f(x) = x^2$$

with x being an integer represented by a 5-bit binary string (i.e., from 00000 to 11111, or 0 to 31 in decimal).

Let us start from this population of solutions:

- A: '01001' (decimal 9), fitness = $9^2 = 81$
- B: '00110' (decimal 6), fitness = $6^2 = 36$
- C: '11100' (decimal 28), fitness = $28^2 = 784$
- D: '10010' (decimal 18), fitness = $18^2 = 324$

The goal of the *selection* operator is to prefer individuals (that is, solutions) with higher fitness so that they are more likely to reproduce (that is, to generate new similar solutions). In the example, the new solutions to be tested would most likely be generated as a variation of C (highest fitness) and not as a variation of B (lowest fitness).

The *crossover* operator combines parts of different solutions to build a new one: for example, it could take the first three bits from C and the last two from D (the solutions with highest fitness).

The *mutation* operator, we may say, operates more in the background, allowing a small probability to also explore less promising areas of the search space and trying to fight the risk of being stuck in a local minimum. Its effect is to change the values of one or more of the bits of the solution under consideration with a given (usually small) probability.

Let us examine a toy example on which we employ a *genetic algorithm*: our goal is to maximize a pseudo-Boolean function:

$$f(x_1, x_2, x_3, x_4) = 2x_1 - 3x_2 + 4x_3 + x_1x_2 - 2x_2x_4 + 3x_3x_4, \quad x_i \in \{0, 1\}.$$

1. Random initial population: $s_1 = (0, 1, 0, 1)$, $s_2 = (1, 0, 1, 0)$, $s_3 = (1, 1, 0, 0)$, $s_4 = (0, 0, 1, 1)$.
2. Fitness: $f(s_1) = -5$, $f(s_2) = 7$, $f(s_3) = 0$, $f(s_4) = 7$.
3. Select elites s_2, s_4; perform crossover to form $s_5 = (1, 0, 1, 1)$ and $s_6 = (0, 0, 1, 0)$.
4. Mutate with a probability of 10% per variable (e.g., flip x_3 in s_6 to obtain $(0, 0, 0, 0)$).
5. Evaluate: $f(s_5) = 9$, $f(s_6) = 0$. If no improvement is achieved for 10 consecutive iterations, stop and return the best-so-far (here, s_5).

Gradient Evolution

Gradient Evolution (GE) combines gradient-based ideas with population-based search. Like GA, it maintains a population; like gradient methods, each solution is nudged toward directions suggested by better/worse peers.

For each solution x_t, we define randomized steps toward the current best (s_{best}), the current worst (s_{worst}), a better peer (s_{better}) and a worse peer (s_{worse}). Thus, we update as follows:

$$x_{t+1} = x_t + \alpha_{best}s_{best} + \alpha_{worst}s_{worst} + \alpha_{better}s_{better} + \alpha_{worse}s_{worse}.$$

We evaluate the new candidate and maintain the better of x_{t+1} and x_t. If a solution stalls for several iterations, we re-initialize it. We repeat until convergence, a maximum number of iterations, or with a time limit.

3.3 Solving Optimization Problems

One example of optimization problem is the Linear Assignment Problem (LAP). It requires finding a one-to-one matching between n agents and n tasks. Mathematically, given a cost matrix C where $c_{i,j}$ represents the cost of assigning the agent i to the task j, the problem is modeled using binary variables $x_{i,j} \in \{0, 1\}$:

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{i,j} x_{i,j} \tag{3.2}$$

Subject to the constraints that ensure that every agent is assigned to exactly one task, and every task receives exactly one agent:

$$\sum_{j=1}^{n} x_{i,j} = 1 \quad \forall i \in \{1, \ldots, n\} \tag{3.3}$$

$$\sum_{i=1}^{n} x_{i,j} = 1 \quad \forall j \in \{1, \ldots, n\} \tag{3.4}$$

This specific formulation possesses a special property known as *total unimodularity*, which allows it to be solved efficiently in polynomial time using algorithms such as the Hungarian Method (Burkard et al., 2012).

The Generalized Assignment Problem (GAP)

However, limiting the study to the linear variant proves to be restrictive, because a simple one-to-one mapping rarely holds in a real-world scenario. When tasks are characterized by specific capacities and agents consume resources, the problem turns into the Generalized Assignment Problem (GAP).

In this formulation, we assume a set of tasks j each having a maximum capacity Q_j, and each assignment of agent i to task j consuming a quantity $w_{i,j}$ of that capacity. The structural constraints change as follows:

$$\sum_{i=1}^{n} w_{i,j} x_{i,j} \leq Q_j \quad \forall j \tag{3.5}$$

While preserving a linear objective, the inclusion of strict capacity constraints eliminates the guarantee of total unimodularity. This modification fundamentally alters the solvability of the model, placing the Generalized Assignment Problem in the NP-hard complexity class (Martello et al., 1990).

The Interpretability Gap

Mathematical optimization is considered a fundamental pillar for decision-making processes in a wide range of sectors, from industry to the economic and logistics fields.

However, the adoption of such tools conflicts with a non-trivial concept: the comprehensibility of the proposed solutions.
Although the mathematical structure of optimization models — composed of an objective function, decision variables, and constraints — is explicit, the final user often perceives them as black-boxes. As argued by (Lumbreras et al., 2025), there is a dichotomy between the mathematical formulation of the problem, which is intrinsically transparent and deterministic, and the output of the model, whose underlying reasons for the optimal choice and trade-offs between alternatives remain inaccessible to non-experts.
This phenomenon gives rise to what is widely recognized a the "Interpretability Gap" (Lipton, 2018). As emphasized by (Rudin, 2019), while a model can be mathematically rigorous and produce an optimal solution, if the logical process driving that decision is indecipherable, the solution risks being viewed as hazardous or even arbitrary.
This lack of interpretability for the end-user has significant operational consequences. As stated by (Nezami et al., 2024), "trust" is a fundamental prerequisite for practical implementation. To address this, the authors introduced the IEMSO (Inclusive Explainability Metrics for Surrogate Optimization) framework. Initially, this provides a set of metrics applicable to any model or optimizer, subsequently generating intermediate and post-hoc explanations to build confidence in the model.
The issue is exacerbated in complex combinatorial or saturated cases—such as resource allocation—where the global optimum often results from counter-intuitive trade-offs between conflicting constraints. In such scenarios, simply presenting the decision-maker with a single vector of optimal variables x^* lacks the necessary context (e.g., clarifying which constraints are active or the solution's sensitivity to small perturbations) and is therefore insufficient.
As further highlighted in (Lumbreras et al., 2025), the focus of recent research is shifting away from pure computational efficiency (finding the minimum of $f(x)$ in the shortest time) towards a balance between Optimality and Explainability. Consequently, the central challenge is no longer only solving the mathematical problem; it is about providing tools that allow users to interrogate the solution. This means understanding not only what the best decision is, but also why it prevails over alternatives and what the structural implications of that choice are.
This shift necessitates a revision of classical methodologies, driving the development of Explainable Optimization techniques that can also leverage methods from Explainable Artificial Intelligence.

Ante-hoc and Post-hoc Approaches

Recent scientific literature categorizes Explainability strategies into two distinct macro-categories, defined by the timing of the explanation relative to the resolution process: Ante-hoc (or Inherently Interpretable) and Post-hoc (or Model-Agnostic) approaches. This distinction, formalized by (Lumbreras et al., 2025) among others, reflects two opposing attitudes toward addressing the trade-off between performance and transparency.

Ante-hoc

Ante-hoc approaches seek to integrate interpretability directly within the mathematical model, constraining the solution's structure—through sparsity or linearity, for instance—to ensure it remains inherently readable by humans. Here, the objective is not solely to solve the original problem with maximum precision, but also to find an optimal approximation expressible through understandable logical rules. For example, within their high-level framework, (Lumbreras et al., 2025) suggest the use of decision trees and their variants, such as optimal policy trees and optimal prescriptive trees.

A pivotal example of ante-hoc methods is found in the work of (Goerigk et al., 2023), who propose a framework for inherently interpretable optimization under uncertainty. Rather than returning a single robust solution or a complex decision rule, their method employs Mixed-Integer Programming (MIP) to simultaneously determine the optimal solutions for the leaf nodes and the structure of a binary decision tree. This tree acts as an interpretable function that maps the cost scenarios realized to specific decision vectors. The model minimizes an aggregate objective (e.g., expected cost) subject to constraints on the tree's depth and structure: specifically, binary variables d_i^q dictate that each node q performs a "univariate split" on a single feature i, ensuring the resulting policy remains human-readable.

However, applying Ante-hoc methods often entails a significant cost, formalized by (Bertsimas et al., 2019) as the "Price of Interpretability." The authors argues that this phenomenon manifests itself as a quantifiable trade-off, representable via a Pareto frontier that contrasts model accuracy with structural complexity. While (Rudin, 2019) argue that this trade-off is often mythical in classification tasks—suggesting that interpretable models can match black-box performance with sufficient effort—in highly constrained optimization environments, the friction is tangible. To force the solution into simple rules (e.g., limiting decision tree depth), it is often necessary to sacrifice objective function optimality or to relax complex operational constraints. In contexts characterized by "saturated" constraints and resource scarcity, such simplification risks

rendering solutions inefficient or even inadmissible, thereby necessitating the exploration of Post-hoc alternatives.

Post-hoc

Conversely, Post-hoc methods embrace the complexity of the original optimization model (black-box) to ensure maximum operational efficiency, deferring the explanation to a subsequent stage. In this scenario, as described by (Ribeiro et al., 2016b), the solver is free to explore the entire feasible solution space to locate the global optimum, while a secondary algorithm analyzes the output to provide insight.

This category encompasses various strategies to justify the output by analyzing the relationships between the input, the output, and the context, without altering the primary decision-making process. The main approaches involve using historical data to identify analogies, using perturbation techniques to estimate the importance of characteristics (Data-Driven explainability), or generating alternative scenarios to test decision robustness, known as Counterfactual Analysis (Wachter et al., 2017). These methodologies, detailed in the following section, currently represent the standard for integrating interpretability into complex industrial models without compromising performance.

Let us briefly consider another example of optimization problem, that is portfolio optimization, particularly relevant in the fintech sector. In such context, our goal would be to maximize the return and minimize the risk associated to the investment of a given capital.

Hierarchical Risk Parity (HRP)

Traditional portfolio optimization approaches, such as those proposed by Markowitz, have well-known limitations. Considering the full correlation structure among all asset returns leads to computational complexity, as all pairwise relationships enter the optimization via the covariance matrix (Fabozzi et al., 2007). Not all assets are strongly correlated, so incorporating every possible correlation can be unnecessary (Vyas, 2020). Estimation error in expected returns and covariances can yield unstable or suboptimal allocations (Michaud, 1989). The classical model is also rigid with respect to regime changes and market dynamics (Ilmanen et al., 2012), it relies on distributional assumptions that often fail (e.g., heavy tails and asymmetries) (Rocco, 2014), and focuses on variance while neglecting other forms of risk, including extreme events (Mandelbrot et al., 1963).

These limitations justify alternative methods. One of the most successful is *Hierarchical Risk Parity* (HRP), which distributes risk more evenly, reduces dependence on expected returns, and improves diversification in volatile settings. HRP is an unsupervised Machine Learning procedure that replaces direct inversion of the covariance matrix with a cluster-based allocation. Using a hierarchical cluster structure rather than the raw covariance matrix itself, HRP fully exploits the information embedded in the covariances while restoring the stability of portfolio weights (Burggraf, 2021). The algorithm proceeds in three steps: *Hierarchical Tree Clustering*, *Matrix Seriation*, and *Recursive Bisection.*

Hierarchical Tree Clustering

Hierarchical Tree Clustering partitions assets into a hierarchy of clusters so that allocations can subsequently flow down a binary tree. HRP is "hierarchical" because its first step assigns assets to clusters using an agglomerative method (Lopez de Prado, 2016). Each cluster can be split into sub-clusters down to the level of individual assets, forming a tree that recursively organizes the investment universe (Vyas, 2020).

Let the returns be organized in a matrix of size $T \times N$, where T is the time dimension and N the number of assets. The procedure is:

(1) Correlations. Compute the $N \times N$ correlation matrix ρ among asset returns.

(2) Distances. Convert correlations to distances D using:

$$D_{i,j} = \sqrt{\tfrac{1}{2}\left(1 - \rho_{i,j}\right)} \tag{3.6}$$

(3) Secondary distances. Compute a second distance matrix $\bar{D}$ based on the pairwise Euclidean distance between the columns of D:

$$\bar{D}_{i,j} = \sqrt{\sum_{k=1}^{N} \left(D_{k,i} - D_{k,j}\right)^2} \tag{3.7}$$

Here, $D_{i,j}$ measures a direct dissimilarity between two assets, while $\bar{D}_{i,j}$ measures the similarity of their dissimilarity profiles with respect to all other assets.

(4) First merge. Let $\mathcal{U}$ be the set of clusters. The first merge (i^*, j^*) is:

$$\mathcal{U}[1] \;=\; \arg\min_{(i,j)} \bar{D}_{i,j} \tag{3.8}$$

	a	b	c	d	e
a	0	17	21	31	23
b	17	0	30	34	21
c	21	30	0	28	39
d	31	34	28	0	43
e	23	21	39	43	0

Table 3.2: Hierarchical Tree Clustering — working principle (step 1). Source: Vyas (2020).

(5) Linkage update (single linkage). Update distances to the new cluster using nearest-point (single linkage). Remove the rows/columns of the merged items (e.g., a and b) and and, for any asset i outside the cluster, define:

$$\bar{D}_{i,\mathcal{U}[1]} \;=\; \min\big(\bar{D}_{i,a},\, \bar{D}_{i,b}\big). \tag{3.9}$$

	(a,b)	c	d	e
(a,b)	0	21	31	21
c	21	0	28	39
d	31	28	0	43
e	21	39	43	0

Table 3.3: Hierarchical Tree Clustering — working principle (step 2). Source: Vyas (2020).

(6) Iterate to one cluster. Repeat merging and updating until a single cluster remains. For example:

$$\bar{D}_{d,\mathcal{U}[2]} \;=\; \min\big(\bar{D}_{d,(a,b)},\, \bar{D}_{d,c},\, \bar{D}_{d,e}\big) \;=\; 28. \tag{3.10}$$

	((a,b),c,e)	d
((a,b),c,e)	0	28
d	28	0

Table 3.4: Hierarchical Tree Clustering — working principle (step 3). Source: Vyas (2020).

Matrix Seriation

Seriation (quasi-diagonalization) reorders the covariance (or correlation) matrix according to the hierarchical ordering so that similar assets are adjacent and large covariances concentrate near the diagonal. This reordering makes the underlying cluster structure explicit and prepares the matrix for top-down allocation.

Recursive Bisection

In the final step, portfolio weights are assigned by traversing the tree from the root to the leaves, splitting capital between sibling sub-clusters in proportion to inverse variances. Here the procedure is:

(1) Initialize. Start with equal cluster weight at the root. (If desired, initialize each asset with $w_i = 1/N$ before scaling through the tree.)

(2) Intra-cluster inverse-variance weights. For a given cluster with covariance matrix V, define the intra-cluster weights as:

$$\mathbf{w} \;=\; \frac{\operatorname{diag}(V)^{-1}}{\operatorname{trace}\Big(\operatorname{diag}(V)^{-1}\Big)}. \tag{3.11}$$

(3) Sub-cluster variances. For the left/right child clusters with covariance matrices V_1 and V_2 and corresponding intra-cluster weights $\mathbf{w}_1$ and $\mathbf{w}_2$:

$$\tilde{V}_1 \;=\; \mathbf{w}_1^\top V_1 \mathbf{w}_1, \qquad \tilde{V}_2 \;=\; \mathbf{w}_2^\top V_2 \mathbf{w}_2. \tag{3.12}$$

(4) Allocate between siblings. Rescale the parent cluster weight between the two children according to inverse variances. One convenient form is:

$$\alpha_1 \;=\; 1 - \frac{\tilde{V}_1}{\tilde{V}_1 + \tilde{V}_2}, \qquad \alpha_2 \;=\; 1 - \alpha_1, \tag{3.13}$$

which is equivalent to allocating proportionally to $(1/\tilde{V}_1,\ 1/\tilde{V}_2)$ up to normalization.[1]

(5) Conservation. At each split, the sum of the child cluster weights equals the parent cluster weight; the final asset weights sum to one.

3.4 Financial Use Case

It is interesting to observe how the optimization principles that have been discussed in this Section can also be linked to the themes explored in the previous Sections, that is, to XAI.

For example, in (Gaggero et al., 2024) the aim is, firstly, to identify an effective allocation of a given capital across 336 candidate stocks, in the Turkish market[2].

The weights corresponding to the share of capital that should be invested in each candidate stock is obtained through the HRP optimization process. Together with being a first actionable information, this serves as the basis for building a supervised learning problem.

The 24 stocks receiving an HRP weight greater than 0.7% are, in fact, labeled as *selected*, while all others are labeled as *not selected*. More specifically, the question that is still open, after observing the result of the HRP algorithm, is: "Why was this stock recommended, while this other was not?".

To reach an answer, what we can do, for example, is to work on the dataset so that it can be seen as a *binary classification problem* (selected stock vs non-selected stock) and then extract an interpretable model from this dataset, such as the *Logic Learning Machine*.

This has also been done in (Gaggero et al., 2024), applying this procedure to a dataset representing the behavior of a set of stocks in the Turkish stock market .

[1] Connection to minimum-variance: the solution of $\min \frac{1}{2}\,\mathbf{w}^\top \Sigma \mathbf{w}$ s.t. $\mathbf{e}^\top \mathbf{w} = 1$ is $\mathbf{w}^\star = \Sigma^{-1}\mathbf{e}/(\mathbf{e}^\top \Sigma^{-1}\mathbf{e})$. For diagonal Σ, $w_i \propto 1/\sigma_{ii}$; for two assets, $w_1 = \frac{1/\sigma_1}{1/\sigma_1 + 1/\sigma_2} = 1 - \frac{\sigma_1}{\sigma_1+\sigma_2}$.

[2] Dataset: https://www.kaggle.com/datasets/gokhankesler/borsa-istanbul-turkish-stock-exchange-dataset

To explain HRP's allocation decisions, both static and dynamic features of the stocks are incorporated into the LLM model: static features include sector classification, market segment, index inclusion, float ratio, and foreign ownership ratio; dynamic features include historical average return, volatility, and derived metrics such as the average standard deviation ratio. These features aim to capture the structural and behavioral characteristics that influence asset selection.

Since only 7% of the stocks are labeled as *selected*, a Weighted Classification System (WCS) is applied. This system assigns higher weights only to selected stocks with higher HRP weights and not to selected stocks with lower HRP weights. This procedure mitigates *class imbalance* and emphasizes more informative samples during training.

The LLM produces a symbolic rule-based model with a constrained complexity (maximum of three conditions per rule, maximum allowed error of 5%). A total of nine rules are generated, six of which have sufficient coverage ($> 20\%$).

The most recurrent explanatory variable is **standard deviation**, confirming that volatility plays a central role in HRP selection:

- Selected stocks tend to have relatively *low* volatility (e.g., SD ≤ 3.06 or ≤ 4.12).
- Additional criteria for selection include positive mean returns and float ratio above 22%.
- Not selected stocks typically exceed volatility thresholds (e.g., SD > 3.02 or > 4.02).
- Sectoral and market-segment characteristics also influence exclusion.

In this way, coupling *optimization* with *XAI*, we were able to identify how to use our decision variables in the most effective way, as well as to provide some *understanding* and *explanation* about how the model operates this choice.

Damiano Verda

Damiano Verda earned the master's and Ph.D. degrees in computer engineering from the University of Genova, in 2010 and 2014, respectively, and the second master's degree in management engineering from Uninettuno Telematic University, in 2020. Since 2014, he has been working with Rulex (a complany producing the Rulex Platform software, inserted in the 2026 Gartner Magic Quadrant for Decision Intelligence) and became the Head of the Data Science Team, in March 2019. His work in Machine Learning and robotic perception has resulted in more than fifty scientific publications. His research interests include explainable AI, rule-based models, and Machine Learning applications. He is a Senior Member of the IEEE-Computational Intelligence Society and an author of multiple papers published by Risk Management Magazine, edited by AIFIRM.

Bibliography

Aizenstein, Howard and Leonard Pitt (1995). "On the learnability of disjunctive normal form formulas." In: *Machine Learning* 19.3, pp. 183–208.

Akiba, Takuya, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama (July 2019). "Optuna: A Next-generation Hyperparameter Optimization Framework." In: pp. 2623–2631. ISBN: 978-1-4503-6201-6. DOI: 10.1145/3292500.3330701.

Amari, Shun-ichi (1993). "Backpropagation and stochastic gradient descent method." In: *Neurocomputing* 5.4-5, pp. 185–196.

Auricchio, Gennaro, Adelaide Emma Bernardelli, Paolo Giudici, and Giuseppe Toscani (2026). "On Rank Graduation Metrics for High Dimensional Ordinal Data." In: *Mathematical Models and Methods in Applied Sciences*.

Azur, Melissa J, Elizabeth A Stuart, Constantine Frangakis, and Philip J Leaf (2011). "Multiple imputation by chained equations: what is it and how does it work?" In: *International journal of methods in psychiatric research* 20.1, pp. 40–49.

Babaei, Golnoosh, Paolo Giudici, and Emanuela Raffinetti (2025). "A rank graduation box for SAFE AI." In: *Expert systems with applications* 259, p. 125239.

Babaei, Golnoosh, Paolo Giudici, and Lunshuai Wu (2026). "Explainable Fairness in Mortgage Lending." In: *Explainable Artificial Intelligence*. Ed. by Riccardo Guidotti, Ute Schmid, and Luca Longo. Cham: Springer Nature Switzerland, pp. 378–398. ISBN: 978-3-032-08330-2.

Behrouz, Ali, Meisam Razaviyayn, Peilin Zhong, and Vahab Mirrokni (2025). "Nested learning: The illusion of deep learning architectures." In: *arXiv preprint arXiv:2512.24695*.

Bertsimas, Dimitris, Arthur Delarue, Patrick Jaillet, and Sebastien Martin (2019). *The Price of Interpretability*. Preprint, Massachusetts Institute of Technology.

Bodria, Francesco, Fosca Giannotti, Riccardo Guidotti, Francesca Naretto, Dino Pedreschi, and Salvatore Rinzivillo (2023). "Benchmarking and survey of explanation methods for black box models." In: *Data Mining and Knowledge Discovery*, pp. 1–60.

Boyd, Stephen and Lieven Vandenberghe (2004). *Convex Optimization*. Cambridge, UK: Cambridge University Press.

Brown, Tom B., Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, et al. (2020). "Language Models are Few-Shot Learners." In: *Advances in Neural Information Processing Systems (NeurIPS)*. URL: https://arxiv.org/abs/2005.14165.

Burggraf, Tobias (2021). "Beyond risk parity–A machine learning-based hierarchical risk parity approach on cryptocurrencies." In: *Finance Research Letters* 38, p. 101523.

Burkard, Rainer, Mauro Dell'Amico, and Silvano Martello (2012). *Assignment Problems*. Philadelphia, PA: SIAM.

Calzarossa, Maria Carla, Paolo Giudici, and Rasha Zieni (2025a). "An assessment framework for explainable AI with applications to cybersecurity." In: *Artificial Intelligence Review* 58.5, p. 150.

— (2025b). "How robust are ensemble machine learning explanations?" In: *Neurocomputing* 630, p. 129686.

Consumer Financial Protection Bureau (2017). *Home Mortgage Disclosure Act (HMDA) Data*. https://www.consumerfinance.gov/data-research/hmda/.

Dantzig, George B. (1963). *Linear Programming and Extensions*. Princeton, NJ: Princeton University Press.

Dubey, Pradeep, Abraham Neyman, and Robert James Weber (1981). "Value theory without efficiency." In: *Mathematics of Operations Research* 6.1, pp. 122–128.

DuMouchel, William H and Greg J Duncan (1983). "Using sample survey weights in multiple regression analyses of stratified samples." In: *Journal of the American Statistical Association* 78.383, pp. 535–543.

Fabozzi, Frank J, Petter N Kolm, Dessislava A Pachamanova, and Sergio M Focardi (2007). *Robust portfolio optimization and management*. John Wiley & Sons.

Ferrari, Enrico, Damiano Verda, Nicolò Pinna, and Marco Muselli (2023). "Optimizing water distribution through explainable AI and rule-based control." In: *Computers* 12.6, p. 123.

Gaggero, Giacomo, Pier Giuseppe Giribone, Marco Muselli, Erenay Ünal, Damiano Verda, et al. (2024). "Portfolio optimization and risk management through Hierarchical Risk Parity and Logic Learning Machine: a case study applied to the Turkish stock market." In: *Risk Management Magazine* 19.1, pp. 26–49.

Garreau, Damien and Ulrike Luxburg (2020). "Explaining the explainer: A first theoretical analysis of LIME." In: *Int. Conf. on artificial intelligence and statistics*. PMLR, pp. 1287–1296.

Giudici, Paolo and Vasily Kolesnikov (2026). "SAFE AI metrics: An integrated approach." In: *Machine Learning with Applications* 23, p. 100821. ISSN: 2666-8270. DOI: https://doi.org/10.1016/j.mlwa.2025.100821. URL: https://www.sciencedirect.com/science/article/pii/S266682702500204X.

Giudici, Paolo and Emanuela Raffinetti (2025). "RGA: a unified measure of predictive accuracy: P. Giudici, E. Raffinetti." In: *Advances in Data Analysis and Classification* 19.1, pp. 67–93.

Goerigk, Marc and Michael Hartisch (2023). *A Framework for Inherently Interpretable Optimization Models*.

Goldberg, David E (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. MIT Press. URL: https : / / www . deeplearningbook.org/.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (2016). "Deep residual learning for image recognition." In: *Proceedings of the IEEE Conf. on computer vision and pattern recognition*, pp. 770–778.

Ilmanen, Antti and Jared Kizer (2012). "The death of diversification has been greatly exaggerated." In: *The Journal of Portfolio Management* 38.3, pp. 15–27.

Kolesnikov, Vasily, Damiano Verda, Nicolò Pinna, Danilo Franco, and Paolo Giudici (2026). "Evaluating Logic Learning Machine model with SAFE-AI Metrics." In: *xAI (Late-breaking Work, Demos, Doctoral Consortium), accepted for publication*.

Konijn, HENDRIK S (1962). "Regression analysis in sample surveys." In: *Journal of the American Statistical Association* 57.299, pp. 590–606.

Le Scao, Teven, Angela Fan, Christopher Akiki, et al. (2022). "BLOOM: A 176B-Parameter Open-Access Multilingual Language Model." In: *arXiv preprint arXiv:2211.05100*. URL: https://arxiv.org/abs/2211.05100.

Lipton, Zachary C. (2018). "The Mythos of Model Interpretability." In: *Queue* 16.3, pp. 31–57.

Liu, Han, Alexander Gegov, and Mihaela Cocea (2015). "Network based rule representation for knowledge discovery and predictive modelling." In: *2015 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*. IEEE, pp. 1–8.

Loh, Wei-Yin (2011). "Classification and regression trees." In: *WIREs Data Mining and Knowledge Discovery* 1.1, pp. 14–23. DOI: https://doi.org/10.1002/widm.8.

López, Susana and Martha Saboya (2009). "On the relationship between Shapley and Owen values." In: *Central European Journal of Operations Research* 17, pp. 415–423.

Lopez de Prado, Marcos (2016). "Building diversified portfolios that outperform out-of-sample." In: *Journal of Portfolio Management*.

Lumbreras, Sara and Pedro Ciller (2025). "Interpretable optimization: why and how we should explain optimization models." In: *Applied Sciences* 15.10, p. 5732.

Lundberg, Scott (2020). *The SHAP Partition Explainer*.

Lundberg, Scott M and Su-In Lee (2017). "A Unified Approach to Interpreting Model Predictions." In: *Advances in Neural Information Processing Systems 30*, pp. 4765–4774.

Mandelbrot, Benoit et al. (1963). "The variation of certain speculative prices." In: *Journal of business* 36.4, p. 394.

Martello, Silvano and Paolo Toth (1990). *Knapsack Problems: Algorithms and Computer Implementations*. Chichester, UK: John Wiley & Sons.

Michaud, Richard O (1989). "The Markowitz optimization enigma: Is 'optimized'optimal?" In: *Financial analysts journal* 45.1, pp. 31–42.

Mikolov, Tomas, Kai Chen, Greg Corrado, and Jeffrey Dean (2013). "Efficient Estimation of Word Representations in Vector Space." In: *Proceedings of the International Conference on Learning Representations (ICLR) Workshops*. URL: https://arxiv.org/abs/1301.3781.

Mitchell, Rory, Joshua Cooper, Eibe Frank, and Geoffrey Holmes (2022). "Sampling permutations for Shapley value estimation." In: *The Journal of Machine Learning Research* 23.1, pp. 2082–2127.

Monderer, Dov and Dov Samet (2002). "Variations on the Shapley value." In: *Handbook of game theory with economic applications* 3, pp. 2055–2076.

Museli, Marco and Enrico Ferrari (2011). "Coupling Logical Analysis of Data and Shadow Clustering for Partially Defined Positive Boolean Function Reconstruction." In: *IEEE Transactions on Knowledge and Data Engineering* 23.1, pp. 37–50. DOI: 10.1109/TKDE.2009.206.

Nezami, Nazanin and Hadis Anahideh (2024). *Building Trust in Black-box Optimization: A Comprehensive Framework for Explainability*.

Nimmagadda, Nidhi and Madiajagan M (2025). "Graph Neural Networks for Illicit Transaction Detection in the Elliptic Bitcoin Dataset." In: *2025 5th International Conference on Evolutionary Computing and Mobile Sustainable Networks (ICECMSN)*, pp. 1121–1127. DOI: 10.1109/ICECMSN68058.2025.11382645.

Nocedal, Jorge and Stephen J. Wright (2006). *Numerical Optimization*. 2nd. New York: Springer Science & Business Media.

Okhrati, Ramin and Aldo Lipani (2021). "A multilinear sampling algorithm to estimate Shapley values." In: *25th Int. Conf. on Pattern Recognition (ICPR)*. IEEE, pp. 7992–7999.

Owen, Guillermo (1972). "Multilinear extensions of games." In: *Management Science* 18.5-part-2, pp. 64–79.

— (2013). *Game theory, 4th Ed.* Emerald Group Publishing.

Owen, Guilliermo (1977). "Values of games with a priori unions." In: *Mathematical economics and game theory: Essays in honor of Oskar Morgenstern*. Springer, pp. 76–88.

Papadimitriou, Christos H. and Kenneth Steiglitz (1998). *Combinatorial Optimization: Algorithms and Complexity*. Mineola, NY: Dover Publications.

Prados, D. L. et al. (1989). "Neural Network Capacity Using Delta Rule." In: *Electronics Letters* 25.3, pp. 197–199.

Quinlan, J Ross (2014). *C4. 5: programs for machine learning*. Elsevier.

— (1986). "Induction of decision trees." In: *Machine learning* 1, pp. 81–106.

Raffel, Colin, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu (2020). "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer." In: *Journal of Machine Learning Research* 21.140, pp. 1–67. URL: http://jmlr.org/papers/v21/20-074.html.

Rao, T. J. (1977). "Optimum Allocation of Sample Size and Prior Distributions: A Review." In: *Int. Statistical Review* 45.2, pp. 173–179. ISSN: 03067734, 17515823.

Rashid, Muhammad, Elvio G Amparore, Enrico Ferrari, and Damiano Verda (2024). "Using stratified sampling to improve lime image explanations." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 38. 13, pp. 14785–14792.

Ribeiro, Marco Tulio, Sameer Singh, and Carlos Guestrin (2016a). ""Why should I trust you?" Explaining the predictions of any classifier." In: *Proc. Int. Conf. ACM SIGKDD, 22nd*, pp. 1135–1144.

— (2016b). "Why should I trust you? Explaining the predictions of any classifier." In: *Proceedings of the 22nd ACM SIGKDD int. Conf.* Pp. 1135–1144.

Rocco, D (2014). "Heavy tails and asymmetry in asset returns." In: *Quantitative Finance* 14.12, pp. 2189–2207.

Rozemberczki, Benedek, Lauren Watson, Péter Bayer, Hao-Tsung Yang, Olivér Kiss, Sebastian Nilsson, and Rik Sarkar (2022a). "The Shapley Value in Machine Learning." In: *IJCAI-22*, pp. 5572–5579.

— (2022b). "The Shapley value in machine learning." In: *IJCAI, arXiv:2202.05594*.

Rudin, Cynthia (2019). "Stop Explaining Black Box Machine Learning Models for High Stakes Decisions and Use Interpretable Models Instead." In: *Nature Machine Intelligence* 1.5, pp. 206–215.

Shapley, Lloyd S (1953). "A value for n-person games." In: *The Shapley value. Essays in honor of Lloyd S. Shapley*, p. 31.

Shrikumar, Avanti, Peyton Greenside, and Anshul Kundaje (2017). "Learning important features through propagating activation differences." In: *International conference on machine learning*. PMLR, pp. 3145–3153.

Sirocchi, Christel and Damiano Verda (2025). "Enhancing interpretability of rule-based classifiers through feature graphs." In: *arXiv preprint arXiv:2506.13903*.

Stock, James A. and Mark W. Watson (2011). *Introduction to Econometrics*. Pearson. ISBN: 978-0138009007.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin (2017). "Attention Is All You Need." In: *Advances in Neural Information Processing Systems (NeurIPS)*. URL: https://arxiv.org/abs/1706.03762.

Vedaldi, Andrea and Stefano Soatto (2008). "Quick shift and kernel methods for mode seeking." In: *Computer Vision–ECCV 2008, Proceedings, Part IV 10*. Springer, pp. 705–718.

Veličković, Petar, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, Yoshua Bengio, et al. (2018). "Graph attention networks." In: *International conference on learning representations*. Vol. 6. Ithaca, p. 2.

Vermeire, Tom, Dieter Brughmans, Sofie Goethals, Raphael Mazzine Barbossa de Oliveira, and David Martens (2022). "Explainable image classification with evidence counterfactual." In: *Pattern Analysis and Applications* 25.2, pp. 315–335.

Vyas, A (2020). "The hierarchical risk parity algorithm: An introduction." In: *Hudson Thames.[Online]. Available at: https://hudsonthames. org/an-introduction-tothe-hierarchical-risk-parity-algorithm/[Accessed: 2023, may 2]*.

Wachter, Sandra, Brent Mittelstadt, and Chris Russell (2017). "Counterfactual explanations without opening the black box: Automated decisions and the GDPR." In: *Harvard Journal of Law & Technology* 31, p. 841.

Ying, Rex, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec (2019). "GNN Explainer: A Tool for Post-hoc Explanation of Graph Neural Networks." In: *CoRR* abs/1903.03894. arXiv: 1903.03894. URL: http://arxiv.org/abs/1903.03894.

Yoo, Seong Ki, Simon L Cotton, Paschalis C Sofotasios, Michail Matthaiou, Mikko Valkama, and George K Karagiannidis (2017). "The Fisher–Snedecor F Distribution: A Simple and Accurate Composite Fading Model." In: *IEEE Communications Letters* 21.7, pp. 1661–1664.

Index

GNN Explainer: A Tool for Post-hoc Explanation of Graph Neural Networks, 60
On the relationship between Shapley and Owen values, 38
The SHAP Partition Explainer, 33, 38
The Shapley Value in Machine Learning, 37, 38
Why should I trust you? Explaining the predictions of any classifier, 21, 33

A Framework for Inherently Interpretable Optimization Models, 88
A multilinear sampling algorithm to estimate Shapley values, 43
A rank graduation box for SAFE AI, 27
A Unified Approach to Interpreting Model Predictions, 24, 33, 38, 43
A value for n-person games, 38
Aizenstein, Howard, 13
Akiba, Takuya, 30
Amari, Shun-ichi, 49
An assessment framework for explainable AI with applications to cybersecurity, 21
Assignment Problems, 86
Attention Is All You Need, 48, 51
Auricchio, Gennaro, 27
Azur, Melissa J, 4

Babaei, Golnoosh, 27, 29
Backpropagation and stochastic gradient descent method, 49
Behrouz, Ali, 48, 52, 54
Benchmarking and survey of explanation methods for black box models, 39
Bertsimas, Dimitris, 88
Beyond risk parity–A machine learning-based hierarchical risk parity approach on cryptocurrencies, 90
BLOOM: A 176B-Parameter Open-Access Multilingual Language Model, 48, 52
Bodria, Francesco, 39
Boyd, Stephen, 78
Brown, Tom B., 48, 52
Building diversified portfolios that outperform out-of-sample, 90
Building Trust in Black-box Optimization: A Comprehensive Framework for Explainability, 87
Burggraf, Tobias, 90
Burkard, Rainer, 86

C4. 5: programs for machine learning, 9
Calzarossa, Maria Carla, 21
Classification and regression trees, 9

Combinatorial Optimization: Algorithms and Complexity, 75, 77
Consumer Financial Protection Bureau, 29
Convex Optimization, 78
Counterfactual explanations without opening the black box: Automated decisions and the GDPR, 89
Coupling Logical Analysis of Data and Shadow Clustering for Partially Defined Positive Boolean Function Reconstruction, vii, ix, 9, 12, 14, 15

Dantzig, George B., 77
Deep Learning, 54
Deep residual learning for image recognition, 35
Dubey, Pradeep, 37
DuMouchel, William H, 36, 46

Efficient Estimation of Word Representations in Vector Space, 48, 49
Enhancing interpretability of rule-based classifiers through feature graphs, vii, 60, 65
Evaluating Logic Learning Machine model with SAFE-AI Metrics, 29
Explainable Fairness in Mortgage Lending, 29
Explainable image classification with evidence counterfactual, 40
Explaining the explainer: A first theoretical analysis of LIME, 36
Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer, 48

Fabozzi, Frank J, 89
Ferrari, Enrico, 10, 63

Gaggero, Giacomo, 93
Game theory, 4th Ed., 38
Garreau, Damien, 36
Genetic Algorithms in Search, Optimization, and Machine Learning, 78
Giudici, Paolo, 27–29
Goerigk, Marc, 88
Goldberg, David E, 78
Goodfellow, Ian, 54
Graph attention networks, 59
Graph Neural Networks for Illicit Transaction Detection in the Elliptic Bitcoin Dataset, 64

He, Kaiming, 35
Heavy tails and asymmetry in asset returns, 89
Home Mortgage Disclosure Act (HMDA) Data, 29
How robust are ensemble machine learning explanations?, 21

Ilmanen, Antti, 89
Induction of decision trees, 9
Interpretable optimization: why and how we should explain optimization models, 87, 88
Introduction to Econometrics, 6

Knapsack Problems: Algorithms and Computer Implementations, 86
Kolesnikov, Vasily, 28, 29
Konijn, HENDRIK S, 47

Language Models are Few-Shot Learners, 48, 52
Le Scao, Teven, 48, 52
Learning important features through propagating activation differences, 38
Linear Programming and Extensions, 77
Lipton, Zachary C., 87
Liu, Han, 60
Loh, Wei-Yin, 9
Lopez de Prado, Marcos, 90
Lumbreras, Sara, 87, 88
Lundberg, Scott, 33, 38
Lundberg, Scott M, 24, 33, 38, 43
López, Susana, 38

Mandelbrot, Benoit, 89
Martello, Silvano, 86
Michaud, Richard O, 89
Mikolov, Tomas, 48, 49
Mitchell, Rory, 43
Monderer, Dov, 43
Multilinear extensions of games, 43
Multiple imputation by chained equations: what is it and how does it work?, 4
Muselli, Marco, vii, ix, 9, 12, 14, 15

Nested learning: The illusion of deep learning architectures, 48, 52, 54
Network based rule representation for knowledge discovery and predictive modelling, 60
Neural Network Capacity Using Delta Rule, 55
Nezami, Nazanin, 87
Nimmagadda, Nidhi, 64
Nocedal, Jorge, 75, 78
Numerical Optimization, 75, 78

Okhrati, Ramin, 43
On Rank Graduation Metrics for High Dimensional Ordinal Data, 27
On the learnability of disjunctive normal form formulas, 13
Optimizing water distribution through explainable AI and rule-based control, 10, 63
Optimum Allocation of Sample Size and Prior Distributions: A Review, 47
Optuna: A Next-generation Hyperparameter Optimization Framework, 30
Owen, Guillermo, 38, 43
Owen, Guilliermo, 38

Papadimitriou, Christos H., 75, 77
Portfolio optimization and risk management through Hierarchical Risk Parity and Logic Learning Machine: a case study applied to the Turkish stock market, 93
Prados, D. L., 55

Quick shift and kernel methods for mode seeking, 34
Quinlan, J Ross, 9
Quinlan, J. Ross, 9

Raffel, Colin, 48
Raffinetti, Emanuela, 27, 28
Rao, T. J., 47
Rashid, Muhammad, vii, 37, 39–41, 47
Regression analysis in sample surveys, 47

RGA: a unified measure of predictive accuracy: P. Giudici, E. Raffinetti, 28
Ribeiro, Marco Tulio, 21, 33, 39, 89
Robust portfolio optimization and management, 89
Rocco, D, 89
Rozemberczki, Benedek, 37, 38, 43
Rudin, Cynthia, 87, 88

SAFE AI metrics: An integrated approach, 28
Sampling permutations for Shapley value estimation, 43
Shapley, Lloyd S, 38
Shrikumar, Avanti, 38
Sirocchi, Christel, vii, 60, 65
Stock, James A., 6
Stop Explaining Black Box Machine Learning Models for High Stakes Decisions and Use Interpretable Models Instead, 87, 88

The death of diversification has been greatly exaggerated, 89
The Fisher–Snedecor F Distribution: A Simple and Accurate Composite Fading Model, 7
The hierarchical risk parity algorithm: An introduction, ix, 89–92
The Markowitz optimization enigma: Is 'optimized' optimal?, 89
The Mythos of Model Interpretability, 87
The Price of Interpretability, 88
The Shapley value in machine learning, 43
The variation of certain speculative prices, 89

Using sample survey weights in multiple regression analyses of stratified samples, 36, 46
Using stratified sampling to improve lime image explanations, vii, 37, 39–41, 47

Value theory without efficiency, 37
Values of games with a priori unions, 38
Variations on the Shapley value, 43
Vaswani, Ashish, 48, 51
Vedaldi, Andrea, 34
Veličković, Petar, 59
Vermeire, Tom, 40
Vyas, A, ix, 89–92

Wachter, Sandra, 89
Why should I trust you? Explaining the predictions of any classifier, 39, 89
Wu, Lunshuai, 29

Ying, Rex, 60
Yoo, Seong Ki, 7

AIFIRM Edizioni – Educational Book Series

Notes on Explainability and Optimization

Damiano Verda

This volume aims to be a synthesis of AI techniques and algorithms that can support the decision-making process, especially in such a challenging and rapidly evolving environment as finance. More specifically, the goal is to make the decision process not only more effective, but also more informed and trustworthy. To this end, the emphasis is on the field of explainable AI, which allows the user not only to be supported in a decision but also to understand why the algorithm is suggesting a given action, as well as on the field of optimization, which enables to make the most of the available decision variables, even when quick response times are needed. The book is divided into three sections: XAI on structured data, XAI on unstructured data, optimization. At the end of each section, applications of the described techniques to financial case study are referenced and briefly described.

ISBN Online: 979-12-80245-39-7