UNIVERSITÀ DEGLI STUDI DI GENOVA

Scuola Politecnica

CORSO DI LAUREA IN ENGINEERING TECHNOLOGY FOR STRATEGY (AND SECURITY)

Tesi di Laurea

Dipartimento di Ingegneria navale, elettrica, elettronica e delle telecomunicazioni

“Rule-based out-of-distribution detection for image classification”

Relatori:
Maurizio Mongelli,
Agostino Bruzzone
Correlatori:
Giacomo De Bernardi,
Sara Narteni

Candidato
Andrea Cappelli

anno accademico 2022/2023

Contents

Abstract

In the context of implementing Deep Neural Networks (DNNs) in perception pipelines for real-time decision-making systems, ensuring their safe usage becomes a paramount concern. The main challenge lies in detecting efficiently and accurately inputs that belong outside the boundaries of the training distribution, known as Out-of-Training-Distribution inputs.

This concern also applies to the detection of Out-of-Distribution (OOD) input data when using Deep Neural Networks. To deal with this problem, we suggest a simple yet effective technique to enhance the resilience of various OOD detection methods, specifically in label change scenarios. This enhancement boosts the robustness and reliability of these methods, making them applicable in diverse real-world situations.

OoD Detection (ODD) is undoubtedly one of the most crucial challenges that arise while deploying machine learning models. Data analysts are responsible for ensuring that operational data correspond to the training phase and remain vigilant to any environmental changes that may compromise autonomous decision-making processes.

The approach used in this thesis is rooted in eXplainable Artificial Intelligence (XAI), which uses different metrics to identify similarities between In-Distribution (ID) and OOD data, as perceived by the XAI model. This novel approach is not reliant from assumptions relating to distribution. Through testing, including in intricate situations like preventive maintenance, vehicle coordination, and covert digital communication, it has been validated as precise in discovering and assessing the proximity between training and operational conditions. The ultimate intention of this inclusive strategy is to overcome the various complex obstacles encountered during practical implementation of machine learning.

Notation

In this chapter, a table will be presented which contains the notations to be used throughout this agreement. It is important to establish these notations in order to ensure clarity and consistency in the document. Technical term abbreviations will be explained upon first use:

OOD	Out-of-Distribution
ID	In-Distribution
ODD	OoD detection
ML	Machine learning
XAI	eXplainable Artificial Intelligence
TR	Training set
OP	Operational set
tr_i	i-th training subset
op_i	i-th operational subset
n_s	number of data samples in a split
N_{tr}	Number of training splits
N_{op}	Number of operational splits
$\mathcal{R}_{tr}$	Training reference ruleset
r_i	i-th rule
h_i^j	j-th hit for the i-th rule
l_p	l_p norm
FN	False Negatives
FP	False Positives
H&R	Hazard&Robots dataset

Chapter 1

Introduction

1.1 Out-of-Distribution

With the proliferation of machine learning, Out-of-Distribution Detection (ODD) has arisen as a significant issue in the field.

Imagine, for instance, that we intended to learn our classifier to distinguish between various dishes based on images. What if a user fed our algorithm a picture of something that is not food, or of an unknown cuisine?

The classifier attempts to approximate the received input based on its knowledge, for example by matching it to a known food that most closely resembles the input based on features established during its previous training phase. However, this approach can lead to an incorrect output. Fortunately, the resulting damage is minimal, as the incorrect output would not cause any harm.

Figure 1.1: Out-of-Distribution example.

But let us consider another example, where a machine that was trained to detect the state of a system, anticipate impending failures and make decisions grounded in the detected state. In case the machine receives a system state that it has not encountered before during its training, it will approximate the seen state to a familiar one leading to an erroneous output, as previously discussed. However, in this instance, the machine may make erroneous judgments, which could result in serious consequences. This approach can prevent potentially harmful decisions from being made. Hence, it is imperative that the machine refrains from rendering an approx-

imation that does not mirror the genuine state of the system. Instead, it is much more suitable for the machine to express its ignorance of the input, indicating that it lies outside the distribution with which it has been trained.

The examples provided are merely two potential cases used to elucidate the concept of OOD.

To provide a clear definition of OOD and ODD, we begin by assuming that numerous current machine learning models are trained based on the assumption of a closed system [1][2], where data for the testing phase is presumed to have the same distribution as the data used for training. Such data is referred to as In-Distribution (ID) data. However, it is likely that these models will be utilized in an open-world context [3], meaning that they will receive input data that is not part of their original distribution, also known as OOD data. A dependable detection system should, therefore, produce precise forecasts in ID scenarios while also being able to identify and dismiss out-of-distribution data [4]; this is referred to as ODD.

The previous examples were presented to illustrate the greatest level of severity concerning OOD, namely, the scenario in which the system fails to fulfil its purpose.

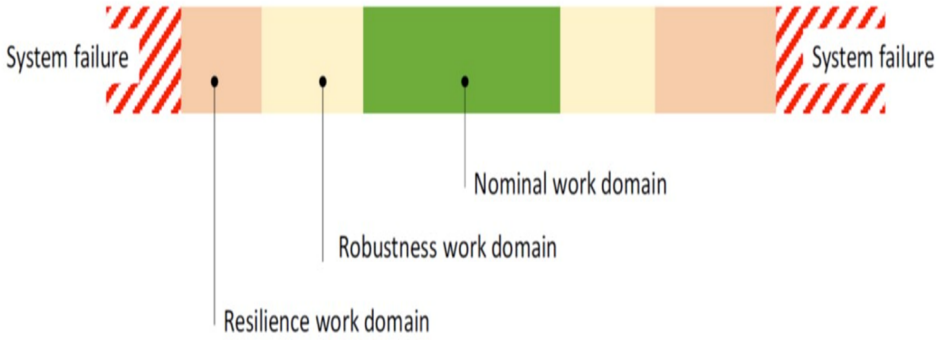


Figure 1.2: EASA illustration of work domains as reported in [5].

As illustrated in Figure 1.2, system failures are not the result of every OOD input as various levels of severity exist. The green bar indicates an in-distribution input, for which the machine's performance is expected, resulting in a correct output. Beginning from the yellow bar and onward, all inputs are OOD. The yellow bar segment poses no issues as the autonomous functions continue to provide precise predictions without any risk. The orange bar denotes the failure of the autonomous functions, but it does not result in any hazardous situations, unlike the red bar scenario where degradation of the system occurs [6].

Figure 1.3: Manarola in 3 different weather conditions.

Why might a machine receive OOD images as input? Apart from instances where the system is given data that is entirely unrelated to its training (such as a classifier intended to identify dog breeds being shown an image of a vehicle), various factors can contribute to the presence of OOD inputs, many of which can be traced to suboptimal training.

In the example presented in Figure 1.3, it is possible that the impact of environmental conditions such as rain or snow as opposed to standard sunny conditions was not taken into account. Other contributing factors include data shortage during training or the rarity of certain inputs in the open world. To mitigate the number of OOD instances, the data augmentation technique [7] could be implemented.

1.2 Contribution

To address this issue, the present thesis introduces a model that will be outlined in detail.

The approach is founded on a thorough analysis centred on an assessment of a rule hits table, which is generated from the validation instances yielded by subjecting a rule-based model to the dataset. This evaluation is crucial in gauging the model's adaptability and performance.

During the initial phase of the system's training, a table is developed, showcasing a unique footprint that captures the characteristics and underlying patterns present within the dataset. This table acts as a reference for subsequent real-time analysis. As the system transitions from the training phase to the operational phase, the table serves as an indispensable reference point to assess incoming data.

The notion of a "significant difference" between tables is pivotal to this process. If the table created during runtime diverges significantly from the established training table, this signals that the incoming data is outside the norm. This deviation serves as a dependable indicator of OOD data, which may require an alternative approach or further checks.

It is notable that this methodology significantly deviates from conventional techniques such as K-Nearest Neighbors (K-NN) and Neural Networks distance calculations. K-NN utilizes a single distance criterion to establish proximity, whereas the Neural Network distance method relies on a single prescribed metric for evaluating similarity. In contrast, the proposed approach adopts a more diverse perspective, allowing for the derivation of similarity measures through a wide range of metrics. The model's flexibility enhances its ability to recognise various patterns and complexities in the data, thereby improving its overall effectiveness.

Chapter 2

Related Works

As mentioned before, the matter of OOD carries great significance in the field of machine learning. It is undeniably essential to ensure that a system is capable of making accurate predictions within its known domain and can also identify unfamiliar territory. This system possesses the dual capacity of comprehending familiar content and responsibly navigating uncharted territories, effectively avoiding negative repercussions [8].

Several solutions proposed to address the OOD challenge in video analysis are based on robust distributional assumptions in feature space [8]. Alternatively, certain individuals presume availability of probability density functions (PDF) for incoming and outgoing data in the training phase [9]. However, such presumptions often result in being impractical in real-world scenarios. Furthermore, several statistical tests encounter difficulties in accurately estimating the authentic distribution of training data due to data scarcity and pdfs' inherent complexity [10].

The issue of neural networks exhibiting overconfidence with out-of-distribution data was first highlighted in [11]. This discovery has led to an increase in research attention towards several productive directions:

1. One approach has attempted to detect OOD samples through designing score functions. These functions include the OpenMax score [12], maximum softmax probability [13], ODIN score [14], deep ensembles [15], Mahalanobis distance-based score [8], and energy score [16]. [17]; [18]; [19]. It is essential to explore activation rectification (ReAct) [20], gradient-based score [21], and ViM score [22]. The authors of [23] demonstrated that methods developed for CIFAR datasets might not translate effectively to a large-scale ImageNet benchmark. Therefore, it is necessary to evaluate OOD detection methods in real-world settings. To date, no previous studies have explored the potential of using non-parametric nearest neighbour. Our research aims to fill this gap by conducting the first investigation into the efficacy of using the nearest-neighbour distance for ODD. We demonstrate remarkable results on various OOD detection benchmarks, emphasising the significant potential of using this approach.

2. Another promising avenue tackled OOD detection through training-time regularization [16, 24–38]. For instance, models are encouraged to provide predictions with a uniform distribution [24] [34] or higher energies [16] [30] [39] [31] for outlier data. Most regularization methods require auxiliary OOD data

availability. Recently, VOS [40] alleviates this need by automatically generating virtual outliers that can meaningfully regularize the model's decision boundary during training.

3. More recently, several works have explored the role of representation learning for ODD. In particular, CSI [41] investigates the type of data augmentations that are particularly beneficial for ODD. Other works [42] [43] verify the effectiveness of applying off-the-shelf multiview contrastive losses such as SimCLR [44] and SupCon [45] for ODD. These two works both use Mahalanobis distance as the OOD score and make strong distributional assumptions by modelling the class-conditional feature space as a multivariate Gaussian distribution. [46] propose a prototype-based contrastive learning framework for OOD detection, promoting stronger ID-OOD separability than SupCon loss. Despite benefiting from high-quality representations, our method and previous works differ fundamentally in the OOD detection approach. Particularly, KNN is a non-parametric method that doesn't impose ID priors. In terms of performance, our method significantly outperforms SSD and is easily applicable in practical scenarios.

Regarding KNN for anomaly detection, KNN has been explored for anomaly detection [47] [48] [49], aiming to detect abnormal input samples from a single class. We focus on ODD, which additionally requires performing multi-class classification for ID data. Some recent works [50] [51] [52] explore the effectiveness of KNN-based anomaly detection for tabular data. The potential of utilizing KNN for ODD in deep neural networks remains underexplored. Our work provides both new empirical insights and theoretical analysis of employing the KNN-based approach for ODD [10] [53].

That said, rule-based ODD methods have not been explored yet. This work has been prepared precisely on this method and specifically studies the XAI ODD method for images, previously applied only to longitudinal data. The novelty lies in applying the XAI method to a compressed image feature space. This will be addressed in the next chapter.

Chapter 3

Materials and Methods

3.1 Uniform Manifold Approximation and Projection for Dimension Reduction (UMAP)

The Uniform Manifold Approximation and Projection (UMAP) [54] is a learning technique to reduce dimensions. It is based on a theoretical framework of Riemannian geometry and algebraic topology. This produces an algorithm that is scalable and practical for real-world data. UMAP is comparable to t-SNE for visualization quality and probably retains more global structure with faster runtime performance. Moreover, UMAP presents no computational limitations on embedding dimension, rendering it a viable general-purpose technique for dimension reduction in machine learning.

UMAP can be described, constructed, and operated as a weighted graph from a practical computational perspective. Therefore, it can be categorized as a k-nearest neighbour-based graph learning algorithm, much like t-SNE [55]. Like other algorithms in this category, we can split it into two phases with an identical fundamental structure that can be summarized as follows:

1. Graph Construction:
   (a) Construct a weighted k-nearest neighbour graph.
   (b) Apply certain transformations to the edges to obtain a local ambient distance.
   (c) Address the intrinsic asymmetry of the k-nearest neighbour graph.
2. Graph Layout:
   (a) Define an objective function that preserves desired features of this k-nearest neighbour graph.
   (b) Find a low-dimensional representation that optimizes this objective function [54].

Some notes:

1. For the construction of the graph, it is crucial to note that membership strengths to fuzzy sets decline as one moves away from them, eventually becoming negligible. Therefore, it is enough to compute them for the nearest neighbours of

each point. To achieve this, an algorithm that quickly (although approximately) calculates nearest neighbours in any dimensional space is vital. This can be accomplished with any nearest neighbour search algorithm or approximate nearest neighbour algorithm, however, the Nearest Neighbour Descent algorithm [56] has been selected. Calculations will only be performed locally for each point at this stage, resulting in high efficiency.

2. For the optimization process, we employed Stochastic Gradient Descent (SGD) [57]. In order to simplify the gradient descent problem, it is necessary for the final objective function to be differentiable. To achieve this, we have utilized a suitably versatile family of functions to approximate the actual membership strength function in the low-dimensional representation. The selected family of functions is of the form $\frac{1}{1+ax^{2b}}$, where 'a' and 'b' represent the weights (and probabilities of existence) of the arcs of two edges.

To avoid addressing every potential boundary, the negative sampling technique (also utilized by other dimension reduction algorithms, such as word2vec and LargeVis) is utilized to sample negative examples as required (Yang et al., 2020). Finally, the use of spectral embedding techniques [58] is required to initialize the low-dimensional representation in a favourable state because the Laplacian of the topological representation approximates the Laplace-Beltrami operator [59] of the manifold.

3.1.1 Hyper-parameters

Hyper-parameter tuning is a persistent issue in machine learning, which is why it's worth discussing:

The UMAP algorithm has the option to choose four hyperparameters:

1. The number of neighbours to consider when approximating the local metric, n;
2. The target inclusion dimensionality, d;
3. The desired separation between neighbouring points in the embedding space, *min-dist*;
4. The number of training epochs to use for optimizing the low-dimensional representation, *n-epochs*;

The effects of d and *n-epochs* are easily understood, however, the effects of n and *min-dist* are not as clear.

The value of n represents the number of nearest neighbours to consider before their membership strengths are negligible, as previously explained. Choosing a low value for n retains the local manifold structure on a small scale but sacrifices a degree of accuracy in overall view. On the other hand, if the number of neighbours is increased, then the large-scale manifold structures are retained, but this results in some loss of small-scale structures.

As for the *min-dist* parameter, it is mainly aesthetic and more beneficial for UMAP's visualisation purposes. This parameter replaces the distance to the nearest

neighbour, which is used to ensure local connectivity and governs how closely points can group together in the low-dimensional representation. Low values result in densely populated regions, which may lead to potential overlaps, while preserving the manifold's structure with greater accuracy. Higher values, however, distribute the points more, resulting in less preserved structure but facilitating visualization.

The impact of varying these two hyperparameters on the representation of a series of randomly sampled values from a three-dimensional colour cube is visualized in Figure 3.1. Using lower values for parameter 'n' can result in the algorithm detecting clusters that may not actually exist, leading to an incomplete perspective. Similarly, opting for higher values for the 'min-dist' parameter can help maintain the overall perspective of the analysis. Therefore, it is advisable to use higher values for the 'neighbors' parameter. Similarly, opting for higher values for the 'min-dist' parameter can help maintain the overall perspective of the analysis.

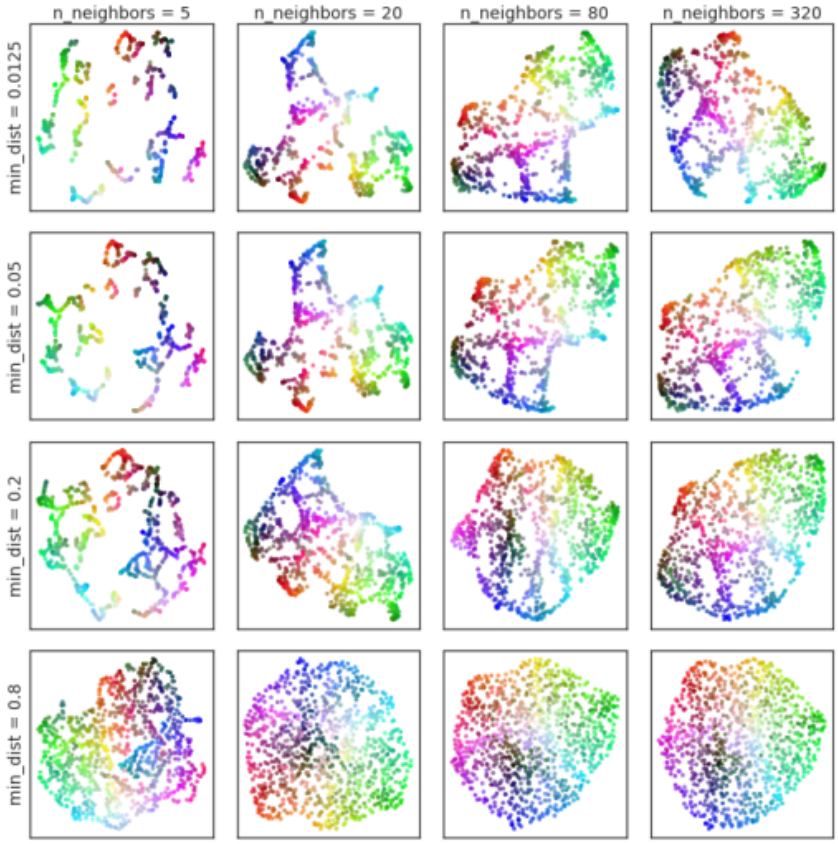


Figure 3.1: Variation in the representation of randomly sampled data from a three-dimensional color cube based on the parameters *n* and *min-dist*. For easier visualization, the three-dimensional coordinates are represented using RGB coloring.

In Figure 3.2, the same methodology is applied to the PenDigits dataset [60] [61]. As clusters exist within the dataset, and we aim to retain them, we would choose a moderate-to-small value for neighbours. The decision for *min-dist* would then depend on the extent to which one wants to enlarge the cluster structure to

depict its density.

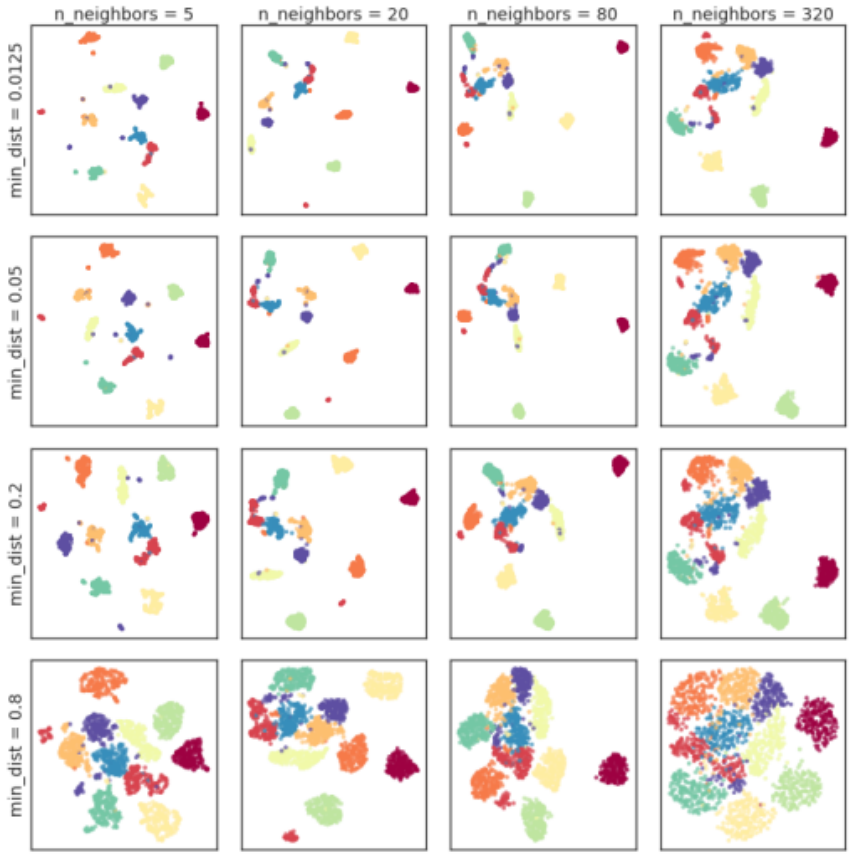


Figure 3.2: Variation in the representation of the PenDigits dataset based on the parameters n and *min-dist*. Each point is an 8x8 grayscale image of a hand-written digit.

3.1.2 Weaknesses

Finally, let's analyse some issues of the algorithm or scenarios where it might be less efficient:

- *Interpretability of results in reduced dimensions*: If interpretability is a top priority, UMAP may not be the most suitable option. Therefore, UMAP may not be the best choice for scenarios where interpretability is crucial. Unlike other algorithms, such as Principal Component Analysis (PCA), the dimensions in the embedding space do not carry specific meanings, and the algorithm relies on distances between observations rather than the original features.
- *Need for manifold structures in data*: In cases where small datasets contain noisy data or there are large-scale manifold structures, it is possible for the algorithm to mistakenly identify manifold structures within the dataset's noise.
- *Preference for local distances over long range distances*: UMAP, along with t-SNE and LargeVis, tends to prioritize local distances over long-range distances.

While UMAP is likely to preserve better overall structure compared to the other two algorithms mentioned previously, it may not be the best choice if the main focus is on global structure.

- *Maintenance of nearest neighbour structure not explicitly preserved*: The omission of explicitly maintaining nearest neighbour structure in high-dimensional spaces may result in the emergence of "reverse-nearest neighbours" in the conventional k-nearest neighbour graph. Moreover, since UMAP prioritizes the preservation of topology instead of purely metric structures, the algorithm's performance may not be optimal when handling datasets that heavily rely on preserving metric structures.

- *UMAP attempts to discover a manifold where data is uniformly distributed*: This implies a desire to maintain ambient distances within the data. In scenarios where preserving ambient distances is crucial, UMAP might not be the most suitable choice.

- *Importance of Large Datasets*: It is crucial to ensure a sufficiently large dataset to obtain optimal outcomes. UMAP utilizes approximations to improve computational efficiency. However, this may lead to inadequate results for datasets with less than 500 samples [54].

3.2 Logic Learning Machine (LLM)

Logic learning machine (LLM) is a machine learning method based on the generation of intelligible rules. It is an efficient implementation of the Switching Neural Network (SNN) [62] paradigm, developed by Marco Muselli and created by Rulex (https://www.rulex.ai/rulex-explainable-ai-xai/), Senior Researcher at the Italian National Research Council CNR-IEIIT in Genoa.

An LLM aims to construct a classifier *g(x)*, described by a set of rules, structured in the form

if *<premise>* **then** *<consequence>*

where *<premise>* is a logical product (AND) of conditions on the input features, while *<consequence>* corresponds to the output class.

The model generation goes through three steps:

1. *Latticization (discretization and mapping to a Boolean lattice)* [63]: Each variable is transformed into a string of binary data in a designated Boolean lattice using the inverse only-one binarization. Finally, for each sample, all these strings are concatenated into one large string.

2. *Shadow Clustering*: A set of binary values (implicants) is generated, facilitating the identification of groups of data points associated with specific classes.

3. *Rule generation*: All the implicants are converted into a collection of simple conditions and eventually combined to form a set of comprehensible rules.

In rule-based models, conditions for a rule are constructed by considering all the involved variables at the same time. This leads to interdependent conditions within the rules. Specifically for LLMs, implicants (binary strings within a Boolean lattice, as previously explained) are used to build conditions. These implicants define unique data point groups associated with a specific class. The groups of points in the Boolean space are later turned into regulations that combine a portion of joint variables. It is easy to generate a clear rule from an implicant that shows a logical product of threshold conditions found through the discretization phase. When generating implicants using the Shadow Clustering technique in LLMs, which examines the complete training set, the resulting rules may overlap and represent distinct important aspects of the phenomenon under study[64][65].

3.2.1 Feature and Value Ranking

An eXplainable Artificial Intelligence (XAI) model enables the examination of its results through feature and value ranking.

Consider a set of m rules $\mathbf{r}k, k = 1, \ldots, m$, each comprising d_k conditions $cl_k, l_k = 1_k, \ldots, d_k$. Let $X_1, \ldots, X_n$ be the input variables, such that $X_j = x_j \in \mathcal{X} \subseteq \mathbb{R}$ for all $j = 1, \ldots, n$. Let $\hat{y}$ also represent the class assigned by the rule and y_j the actual output of the j-th instance.

A condition c_{l_k} involving the variable X_j can take one of the following forms:

$$X_j > s, \quad X_j \leq t \text{ or } s < X_j \leq t.$$

where s, $t \in \mathcal{X}$.

For each rule, it is possible to define a confusion matrix that consists of four indices: $TP(\mathbf{r}_k)$ and $FP(\mathbf{r}_k)$, defined as the number of instances (x_j, y_j) that satisfy all the conditions in rule $\mathbf{r}_k$ with $\hat{y} = y_j$ and $\hat{y} \neq y_j$ respectively; $TN(\mathbf{r}_k)$ and $FN(\mathbf{r}_k)$, defined as the number of examples (x_j, y_j) that do not satisfy at least one condition in rule $\mathbf{r}_k$, with $\hat{y} \neq y_j$ and $\hat{y} = y_j$, respectively.

Consequently, we can derive the following metrics:

$$C(\mathbf{r}_k) = \frac{TP(\mathbf{r}_k)}{TP(\mathbf{r}_k) + FN(\mathbf{r}_k)}$$

$$E(\mathbf{r}_k) = \frac{FP(\mathbf{r}_k)}{TN(\mathbf{r}_k) + FP(\mathbf{r}_k)}$$

The covering $C(\mathbf{r}_k)$ is adopted as a relevance measure for a rule $\mathbf{r}_k$; in other words, the greater the covering, the more general the corresponding rule is considered. The error $E(\mathbf{r}_k)$ measures how many data points are incorrectly covered by the rule. Both covering and error are used to define feature ranking and value ranking.

Feature ranking (FR) offers a ranking of the features utilized in the rule conditions based on a relevance measure. To obtain the relevance $R(c_{l_k})$ for a condition, we consider rule $\mathbf{r}_k$ in which condition c_{l_k} appears, and the same rule without condition c_{l_k}, denoted as $\mathbf{r}'_k$. Since the premise part of $\mathbf{r}'_k$ is less restrictive, we deduce that $E(\mathbf{r}'_k) \geq E(\mathbf{r}_k)$, and thus the quantity $R(c_{l_k}) = (E(\mathbf{r}'_k) - E(\mathbf{r}_k)) \cdot C(\mathbf{r}_k)$ can be used as a measure of relevance for the specific condition c_{l_k}. Each condition c_{l_k} pertains to a specific variable X_j and is verified by certain values $\nu_j \in \mathcal{X}$. In this

way, a relevance measure $R\hat{y}(\nu_j)$ for every value assumed by X_j is derived using the following equation:

$$R_{\hat{y}}(\nu_j) = 1 - \prod_k \left(1 - R\left(c_{l_k}\right)\right),$$

where the product is computed over the rules $\mathbf{r}k$ that include a condition cl_k verified when $X_j = \nu_j$. As $R_{\hat{y}}(\nu_j)$ takes values in $[0, 1]$, it can be interpreted as the probability that the value ν_j occurs in predicting $\hat{y}$. The same argument can be extended to intervals $I \subseteq \mathcal{X}$, thus defining the *Value Ranking (VR)*. Relevance scores are then ranked, revealing the most sensitive interval of the feature with respect to each class [66].

3.3 Out-of-Distribution Detection Algorithm

The algorithm belongs to the category of groupwise methods which utilise an entire collection of data points for evaluation instead of classifying each individual one as IN or OUT, unlike pointwise methods. This approach guarantees accuracy as outlier data points won't cause errors in the overall trend of the data.

The algorithm can be divided into 4 parts:

1. *Feature Reduction*: The first step involves train a UMAP model using the training dataset and then apply that model to the operational one or reducing the number of features in the dataset containing both training and operational data (as we did during the tests). This step is crucial to simplify and reduce the computational cost of subsequent steps. However, it's important that the feature reduction doesn't lead to information loss in the dataset.

2. *Rule Creation*: The second step involves creating rules for each output class using the Rulex program as if training a classifier using the training dataset. The program generates a set of rules describing the classes we consider ID. In the case of very simple classes or classes with few images, it might happen that Rulex generates only one rule for that specific class. Having just one rule could pose problems for the algorithm, as we will see later.

3. *Rule Hits Tables Creation*: Once rules for ID classes are obtained, rule hits tables are created. Two types of tables are generated: one for training and one for operational purposes. Training tables use the dataset used to generate rules, divided by each output class. Operational tables are generated from a dataset containing data that our algorithm must recognize as ID or OOD. For both datasets, random splits are created by randomly selecting data from the respective dataset (randomly selected data can appear in multiple splits but not multiple times in the same split). Rule hits tables are matrices **n** x **m**, where **n** is the number of splits, and **m** is the number of rules. The training table for each class is created as follows, starting from the first split:

   (a) For each image in the current split, the algorithm checks if it satisfies each rule generated for that class.

(b) The algorithm counts how many times each rule is satisfied by the images in the split and divides this number by the total number of images in the split (resulting in 1 if a rule is satisfied by every image in the split).

(c) If there are multiple images in that split or if there are multiple splits, the process continues with the next image or split.

Operational tables are created similarly to training tables, except that rules are compared with images from the operational dataset. In the end, for each rule class, we will have a training rule hits table and an operational one.

In a more mathematical vein, let $\mathcal{R}_{tr}$ represent a set of rules generated from a training set, where N_r signifies the number of rules it comprises. Let N_{tr} and N_{op} denote the number of splits in the training domain and operational domain, respectively, resulting in a total of $N_h = N_{tr} + N_{op}$ splits. Consider n_s as the quantity of data samples within a split. Within each split, samples may or may not satisfy individual rules a specific number of times, which we term the "number of hits" for that rule. Consequently, N_h vectors are defined, the value of which will be normalised by the split size n_s:

$$\mathbf{h}^j = \left\{ h_i^j \right\}, \quad h_i^j \in [0,1], i = 1, \ldots, N_r, j = 1, \ldots, N_h$$

Each vector $\mathbf{h}^j$ is a table [6]. The structure of training and operational tables is as shown in Table 3.1 and 3.2.

	tr_1	...	$tr_{N_{tr}}$
r_1	$h_1^{tr_1}$	...	$h_1^{tr_{N_{tr}}}$
r_2	$h_2^{tr_1}$	...	$h_2^{tr_{N_{tr}}}$
.	.	.	.
.	.	.	.
r_{N_r}	$h_{N_r}^{tr_1}$	...	$h_{N_r}^{tr_{N_{tr}}}$

Table 3.1: Training numbers of hits table. Each column refers to a training split tr_i and each row to a rule $r_i \in \mathcal{R}_{tr}$.

4. *Out-of-Distribution Detection*: Once all the required tables have been acquired, the algorithm moves onto the ODD algorithm. For each class, the algorithm compares the corresponding training table with the operational table by comparing the divisions of the two tables. If the observed difference surpasses a certain interval, the pair of divisions currently being analysed is deemed unsatisfactory. If more than half of the analysed pairs are deemed unsatisfactory, the table is considered unsatisfactory itself.

   The OOD algorithm can be split into three phases, commencing from the first class.

	op_1	...	$op_{N_{op}}$
r_1	$h_1^{op_1}$	...	$h_1^{op_{N_{op}}}$
r_2	$h_2^{op_1}$	...	$h_2^{op_{N_{op}}}$
.	.	.	.
.	.	.	.
r_{N_r}	$h_{N_r}^{op_1}$	...	$h_{N_r}^{op_{N_{op}}}$

Table 3.2: Operational numbers of hits table. Each column refers to an operational split op_i and each row to a rule $r_i \in \mathcal{R}_{tr}$.

(a) *Interval Determination*: The first step of the algorithm involves finding the interval that determines whether a calculated distance between a split of the training table and the operational table is considered IN or OUT. To do this, we use the l_p norm with $p = 1$ or 2:

$$l_p(tr_i, tr_j) = \left[\sum_{r=1}^{N_r} \left(|h_r^{tr_i} - h_r^{tr_j}| \right)^p \right]^{\frac{1}{p}}, \quad \forall i, \forall j$$

for each combination of splits within the training table. Once all distances between splits are calculated, the algorithm selects the maximum and minimum. These two values form our ID interval.

(b) *Operational Table Evaluation*: Using the l_p norm again, we calculate distances between splits of the training table and those of the operational table. Each time a distance is calculated, it's compared with the maximum and minimum found in step (a). If more than 50% of the calculated distances fall outside the maximum and minimum interval, the table is considered OOD.

(c) *Class Iteration and Evaluation*: At this point, there are three possible options:

i. If the table that has just been calculated is in-distribution for that specific class, the algorithm terminates. If this happens, the operational dataset is considered *In-Distribution*.

ii. If the table that has just been calculated is out-of-distribution for that specific class and there are other classes, the algorithm returns to step (a) with the next class.

iii. If the table that has just been calculated is out-of-distribution for that specific class and there are no other classes, the algorithm terminates. If we have reached this point, the dataset is then considered *Out-of-Distribution*.

N.B.: In the explanation above, a single method of evaluating the "distance" between tables (l_1 or l_2 norm) was considered. It's possible to use multiple methods simultaneously (alongside other methods not covered in this study).

Please refer to Chapter A.1 for further explanation by analysing an example of code used in the study.

Figure 3.3 illustrates a summary diagram of the initial implementation of the ODD algorithm, demonstrating The chart illustrates the separation between the design time and operation time. If applied in real-time settings, subsequent iterations would replace the 'design time' portion of the chart with some incremental technique for incorporating new frames.

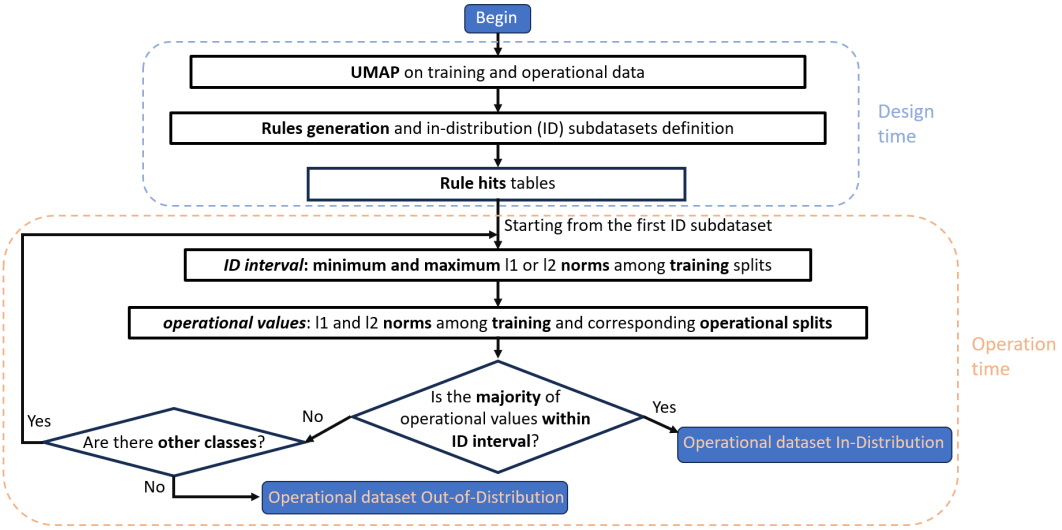


Figure 3.3: Summary diagram of the ODD algorithm.

Chapter 4

Performance Evaluation

4.1 Datasets

During this study, the following two datasets were used:

1. *MNIST*: The MNIST (Modified National Institute of Standards and Technology database) Digits is a widely used dataset of handwritten digits for training and testing in the field of machine learning. Created from the NIST database, it was modified by reducing the number of data points to 60.000 for the training set and 10.000 for the test set: the number of distinct writers to 250, and by normalizing and centering each digit. Each image is composed of 28x28 pixels [67]. In the specific case, two versions of the dataset were used. The first one is from the Scikit-Learn library [68] (we will refer to it as "small MNIST") which further reduces the number of images to 1797 and their dimensions to 8x8 pixels, resulting in a dataset size of 1.797,65 (64 columns for pixels and 1 column for the target) with approximately 180 images per digit. The other one (we will refer to it simply as "MNIST"), downloaded from https://www.kaggle.com/datasets/oddrationale/mnist-in-csv, contains, as already mentioned, 60.000 images for the training set (we will only use this dataset), composed of 28x28 pixels, resulting in a dataset size of 60.000,785, with 6.000 images per digit.

2. *CIFAR-10*: The CIFAR-10 (Canadian Institute For Advanced Research) is a dataset containing images of 10 different subjects: automobiles, airplanes, birds, cats, deer, dogs, frogs, horses, ships, and trucks. Along with its larger version, CIFAR-100, CIFAR-10 is also extensively used in the field of machine learning, similar to MNIST. The dataset consists of 60,000 small images with dimensions of 32x32 pixels. Unlike MNIST, CIFAR-10 is in colour, with 1024 pixels for each colour channel (RGB), resulting in a total of 3072 pixels per image. The total size of the dataset is therefore 60.000,3073. In the specific case, the version of the dataset from the Keras [69] library was used. In this version, the dataset is already split between the training set and the test set. The training set consists of a total of 50.000 images (5.000 per class), and it will be the dataset used for testing purposes.

Figure 4.1: Small MNIST dataset example

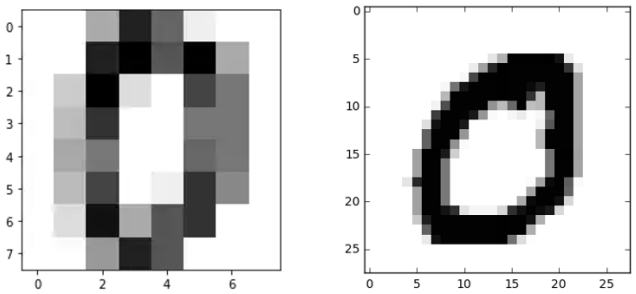


Figure 4.2: Example of the digit ”0” with size 8x8 (Small MNIST) and 28x28 (MNIST) pixels.

3. *Hazards&Robots:* The third and final dataset used is the Hazard&Robots dataset from the University of Lugano (USI-SUPSI) [70]. This dataset differs from the other two mentioned as it is a ”real” dataset. Unlike MNIST and CIFAR-10, this dataset consists of images that have not been pre-processed to make them easier to identify. In fact, the collection of images is nothing more than a set of frames obtained from a recording made by a robot moving within various environments. The dataset contains three types of scenarios: Tunnel, Factory, and Corridors, each of them structured into training sets, validation sets, and test sets. Among the three available versions, the most recent version was chosen, which contains only the final version of Corridors. It consists of 324.408 frames, each measuring 521x521 pixels, and is divided into 21 classes of various corridors. These 21 classes include one ”normal” class and 20 classes that represent anomalies in the robot’s path: box, cable, cones, debris, defects, door, floor, human, misc, tape, trolley, clutter, foam, sawdust, shard, cellophane, screws, water, obj. on robot, obj. on robot2. Similar to the

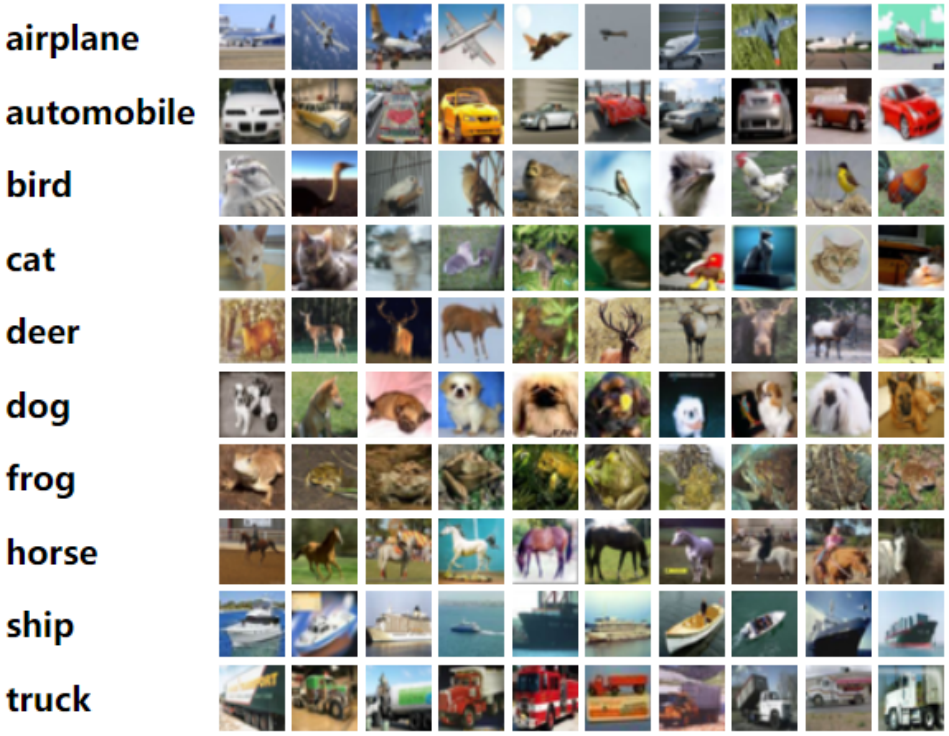


Figure 4.3: CIFAR-10 dataset example

CIFAR-10 dataset, only the training set containing 144.603 frames was used in this case. Unlike the other two datasets, in this case, the 512 features of the images had already been extracted using a CLIP ViT-B/32 model [71], and individual pixels will not be used.

4.2 Experiments settings

During the study, numerous tests were conducted, which were also useful for refining and perfecting the technique. In this section and the next, we will analyse 8 experiments that I consider the most significant and interesting. Therefore, transient examples in which only a few classes were used will be omitted. For reasons of speed and code reusability, it has always been chosen to exclude the last class of each dataset from the training distribution (and therefore it will be used as an operational class). The 8 tests that we will analyse are as follows:

1. *Small MNIST False Negatives rate:* In this experiment, classes 0 to 8 were considered as in-distribution classes, and class 9 was used as an OOD class. We may encounter FN if an image sub-dataset for class 9 is deemed to be incorrectly ID in relation to any training class. Using UMAP, the original 64 features (4.6) were reduced to only 3 (in 4.7, you can see the graph of the reduction to two features/dimensions). The goal of the experiment is for the dataset of class 9 to be recognized as OOD compared to each of the other 8 classes. Once it was certain that the algorithm was functioning correctly,

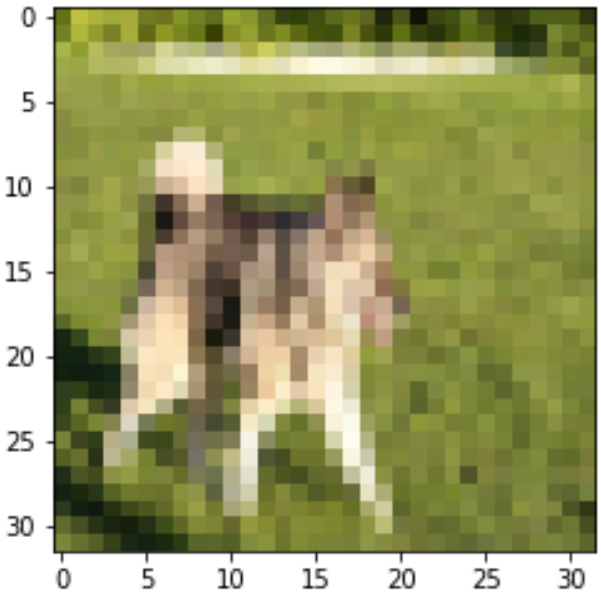


Figure 4.4: Example of a ”dog” class 32x32 pixels image in CIFAR-10.

the number of times, despite the correct classification as OOD, the algorithm calculated that a distance between the training and operational rule hit tables was in-distribution was calculated. Indeed, it will be the majority of results that classify a split as IN or OUT, and it will be the majority of splits that consider a table as IN or OUT. Considering a distance classified as OOD as positive, we will call this value the False Negatives rate.

Number of generated rules per class: {”0”, 1 rules; ”1”, 2 rules; ”2”, 3 rules; ”3”, 2 rules; ”4”, 1 rules; ”5”, 2 rules; ”6”, 1 rules; ”7”, 1 rules; ”8”, 2 rules.}

2. *Small MNIST False Positives rate:* After observing that the algorithm correctly classified an OOD class, it was necessary to ensure that it did not consider everything as OOD. For this reason, in this experiment, all 10 classes were used, and approximately 1/5 of the images from each dataset (before being used to generate the rules) were extracted and used as operational datasets. Since the images for each class were only about 180, the operational datasets were very small (approximately 35 images), and it was not possible to conduct too many experiments by changing the number of data for splits and the number of splits themselves. Having modified the training dataset, the generated rules may differ from those of the previous experiment. In this case, instead of creating operational rule hit tables with an operational dataset and rules for all classes, the various sub-datasets previously extracted, along with the rules of their respective classes, will be used and subsequently compared with their corresponding training tables. The result of these comparisons should obviously result in in-distribution. This time, we will examine when a distance is mistakenly considered as OOD and will then calculate the False Positives value. We will therefore regard it as a False Positive when a set of images

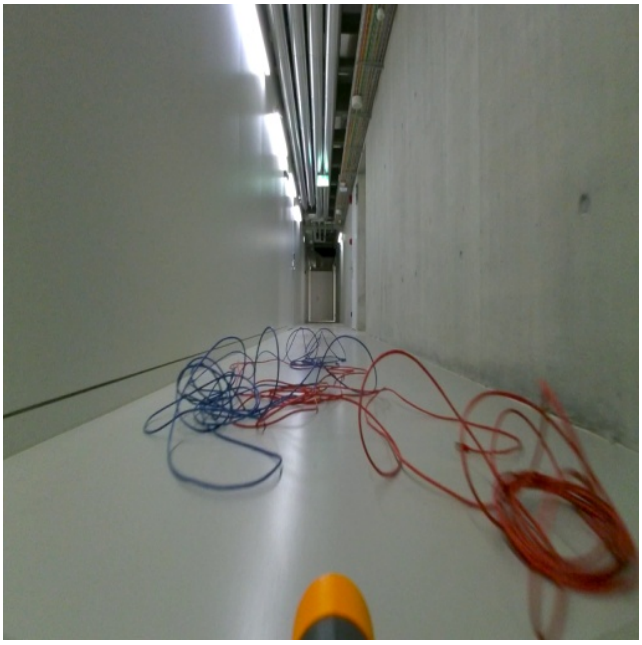


Figure 4.5: Example of a frame from the Hazard&Robots dataset.

from any given class is erroneously labelled as OOD when compared with the corresponding training class.

Number of generated rules per class: {"0", 1 rules; "1", 2 rules; "2", 2 rules; "3", 2 rules; "4", 2 rules; "5", 2 rules; "6", 1 rules; "7", 1 rules; "8", 2 rules; "9", 4 rules.}

3. *MNIST False Negatives rate:* We then moved on to a larger dataset, the complete MNIST. The original 784 features were reduced to 5 using UMAP. In this case, since it is not possible to visualize a graph in more than 3 dimensions, the value 5 was chosen by trial and error. In Section 5, we will talk about how to determine the number of features to reduce an image based on the number of original pixels. Here too, it was chosen to create the rules with classes from 0 to 8 and use class 9 as the operational class to be classified as OOD and calculate the False Negatives rate.

   Number of generated rules per class: {"0", 5 rules; "1", 4 rules; "2", 11 rules; "3", 10 rules; "4", 7 rules; "5", 8 rules; "6", 7 rules; "7", 6 rules; "8", 13 rules.}

4. *MNIST False Positives rate:* In this case, as well, it was also checked that the algorithm did not fail to recognize a class as in-distribution. Once again, it was decided to extract 1/5 of the 6.000 images from each class to create operational rule hit tables. The operational tables were then compared with their respective training tables, and the value of False Positives was calculated.

   Number of generated rules per class: {"0", 5 rules; "1", 4 rules; "2", 11 rules; "3", 11 rules; "4", 7 rules; "5", 8 rules; "6", 7 rules; "7", 8 rules; "8", 14 rules; "9", 9 rules.}

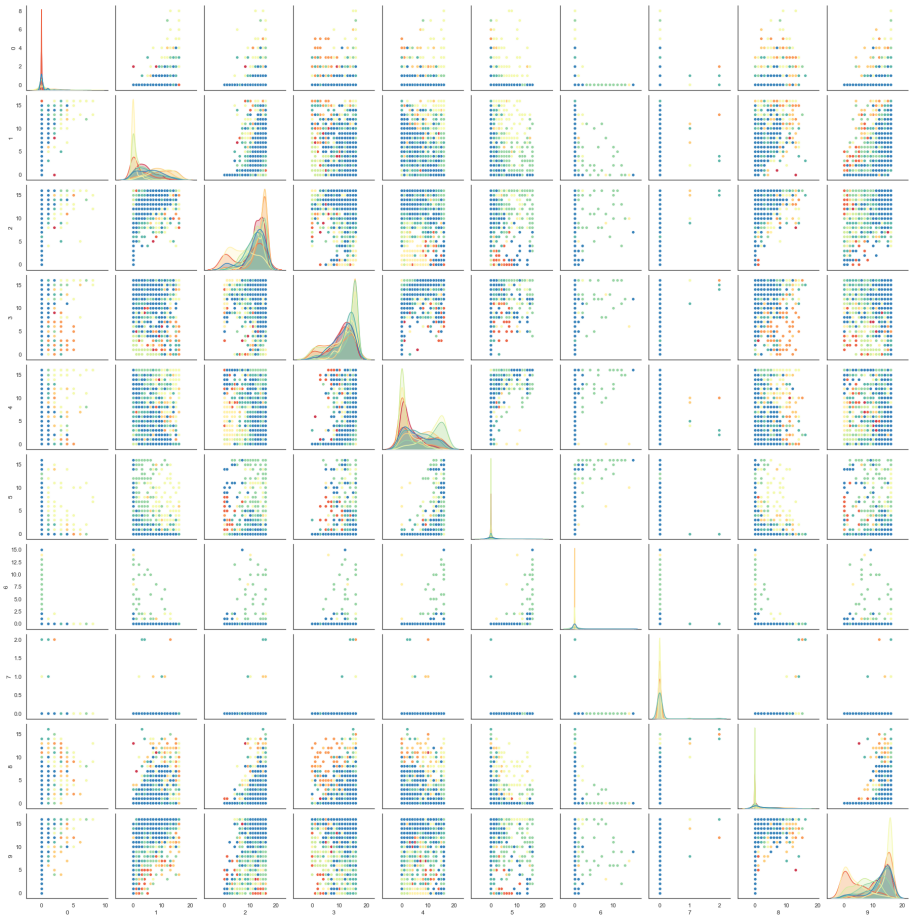


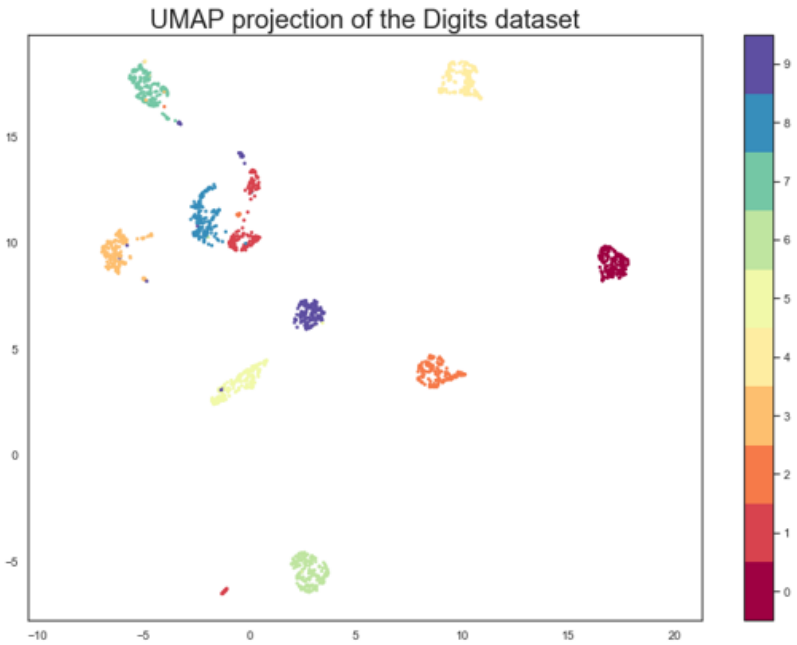
Figure 4.6: Original Small MNIST dataset features.

5. *CIFAR-10 False Negatives rate:* The 3072 features were reduced to 10 with UMAP. For CIFAR-10, it was chosen to use the class of "truck" as the operational class, being the last class in the dataset. The goal was to classify the "truck" dataset as OOD and calculate the False Negatives rate with a dataset even larger than MNIST. We shall therefore consider a False Negative when a collection of images of the class 'truck' is considered to be wrongly ID when compared with any training class.

   Number of generated rules per class: {"Airplane", 24 rules; "Automobile", 26 rules; "Bird", 27 rules; "Cat", 29 rules; "Deer", 24 rules; "Dog", 26 rules; "Frog", 24 rules; "Horse", 25 rules; "Ship", 20 rules.}

6. *CIFAR-10 False Positives rate:* For CIFAR-10 as well, it was ensured that there were no issues with in-distribution classification. Operational rule hit tables were still created using 1/2 of the original datasets for each class. Again, we will therefore regard it as a False Positive when a set of images from any given class is erroneously labelled as OOD when compared with the corresponding training class.

   Number of generated rules per class: {"Airplane", 19 rules; "Automobile",




```
const char *ApplyRules(float umap_0, float umap_1, float umap_2) {
  if ((umap_0 > 7.368073)) return "0";
  if ((umap_0 <= -1.175914) && (umap_2 <= 3.976575)) return "1";
  if ((umap_1 <= -4.553245)) return "1";
  if ((umap_2 > 13.235634)) return "2";
  if ((umap_0 > -3.044371 && umap_0 <= -2.937567)) return "2";
  if ((umap_1 > 8.761285) && (umap_2 > 6.563538 && umap_2 <= 9.046923)) return "3";
  if ((umap_1 > 10.132623) && (umap_2 > 3.520785)) return "3";
  if ((umap_2 <= -1.269070)) return "4";
  if ((umap_1 > 15.051433 && umap_1 <= 15.381656)) return "4";
  if ((umap_1 > 0.118048 && umap_1 <= 6.028403)) return "5";
  if ((umap_1 > -1.835151 && umap_1 <= -1.828176)) return "5";
  if ((umap_1 > -4.553245 && umap_1 <= 0.118048)) return "6";
  if ((umap_1 > 13.955429)) return "7";
  if ((umap_0 > -2.937567 && umap_0 <= 1.339758) && (umap_2 > 3.871140 && umap_2 <= 8.748409)) return "8";
  if ((umap_0 > -3.025130 && umap_0 <= -0.883662) && (umap_1 > 10.001479)) return "8";
  if ((umap_1 > 4.609788 && umap_1 <= 7.756487)) return "9";
  if ((umap_1 > 1.902258 && umap_1 <= 9.667917) && (umap_2 > 8.506213 && umap_2 <= 8.924102)) return "9";
  if ((umap_0 <= -0.453387) && (umap_1 > 10.193263 && umap_1 <= 15.866141)) return "9";
  if ((umap_0 > -1.630122 && umap_0 <= -1.625978)) return "9";
```

Figure 4.7: Small MNIST dataset features reduction to a 2-dimensional space and related rules.

23 rules; "Bird", 24 rules; "Cat", 24 rules; "Deer", 23 rules; "Dog", 23 rules; "Frog", 20 rules; "Horse", 24 rules; "Ship", 18 rules; "Truck", 23 rules.}

7. *H&R False Negatives rate:* The 512 features extracted using the CLIP ViT-B/32 model were reduced to 4 using UMAP. Since it is a "real" dataset, we wanted to maintain its original purpose without altering it. This is why only the "normal" class was used as IN, and the 20 anomaly classes were used as OUT. Having only one IN class required splitting it into two ("normal1" and "normal2") in order to generate rules. This division couldn't be done arbitrarily within the dataset, as two nearly identical images (two consecutive frames) belonging to different classes could potentially confuse our LLM (Language Model). Therefore, it was decided to split the dataset after the first 3,185 frames, which is the point of the first scenario change. The aim of this test is to classify each anomaly class as OOD (Out-of-Distribution) and calculate the

False Negative rate in a "real" dataset with a different approach to features. We shall therefore consider a False Negative when a collection of images of one of the "abnormal" classes (like the frame of Image 4.5) is considered to be wrongly ID when compared with "normal" classes.

Number of generated rules per class: {"normal1", 16 rules; "normal2", 5 rules.}

8. *H&R False Positives rate:* For this last dataset as well, it was ensured that the algorithm did not make any misclassifications in the in-distribution class. Unlike the other datasets where all classes were used, in this case, the decision was made to use only the "normal" class (once again divided as in the previous case) to preserve the original intent of the authors. Consequently, only four rule hit tables were created: two for training and two for operational purposes. The operational tables were still created using half of the original datasets for the two classes. Finally, we will therefore regard it as a False Positive when a set of images from any "normal" class is erroneously labelled as OOD when compared with the corresponding class.

   Number of generated rules per class: {normal1", 15 rules; "normal2", 4 rules.}

For the three datasets containing multiple images (MNIST, CIFAR-10, and H&R), it was decided to use the same values for the number of splits and the number of images per split in all six tests. This decision was made to facilitate a more accurate comparison of the results across the different tests. In the case of Small MNIST, this was not possible due to the small number of images. Nevertheless, it was a useful test to observe the behaviour of the algorithm even with limited training data.

After performing these tests, further tests were performed considering a single operational split, similar to how it would be in a real application case where only the last n images would be chosen as the operational dataset and placed in a single split. The purpose of these tests was to see if the algorithm would also work correctly using only one operational split and, if necessary, to see how many images would be needed to perform well. For all tests, 50 splits and 300 images per split were used as base values for training. It was also chosen not to perform these tests with the Small MNIST dataset. Both False Negatives and False Positives rates are performed for each dataset.

1. *MNIST*: For the MNIST dataset it was decided to start with 100 images per split in operational and then decrease the number to find the minimum value.

2. *CIFAR*: For the CIFAR dataset it was also decided to start with 100 images per split into operational. Due to the values obtained, tests were then performed with different values in training, which will be discussed in the 4.3 section.

3. *H&R*: Finally, also for the H&R dataset it was decided to start from 100 images per split in operational and then decrease the number to find the minimum value.

N.B.: All tests were performed several times for each value used to ensure that the result was reliable and did not depend only on the data that had been selected for the operational dataset.

4.3 Results

In this section, we will present and analyse the individual results of all the tests described earlier in Section 4.2. For a more general discussion of all the compared results, please refer to Section 4.3.1. Before analysing the results, I would like to remind you of how they were generated: during the comparison between the training and operational rule hit tables, distances between every possible combination of splits are calculated using some metrics (in this case, L1 and L2 norms). If the calculated distance falls within the minimum and maximum values obtained from comparing the distances of the training table splits with themselves, it is considered in-distribution; otherwise, it is considered OOD. In cases where we need to classify a dataset as OOD, we count how many times the distance resulted in IN out of the total calculated distances (False Negatives rate). Conversely, if our intention is to consider a given dataset as in-distribution, we calculate how many times a distance resulted in OOD relative to the total.

In Figure 4.8, the results of the test with the Small MNIST dataset, considering the class of digit 9 as OOD, are presented. It can be seen that in this case, the l_2 norm performs slightly better than the l_1 norm. The highest error percentage occurs in cases where there are few images per split, reaching a peak of approximately 7-8%. This error may be due to the fact that each split will easily contain all the images that the others do not, creating tables with many differences between columns. I would like to clarify that, despite the relatively high error with few images per split, there were no classification errors for any class. The number of splits also seems to influence the result. Tests with 20 splits had more errors in 3 out of 4 cases compared to their 10-split counterparts. Again, this could be related to the previous explanation. Once the threshold of 30 images per split is exceeded, the error becomes 0.

- Best case: from 30,10 onwards (0% error) with both norms.
- Worst case: 10,20 with L1 norm (7.6% error).

In Figure 4.9, the results of the test with the Small MNIST dataset, in which the algorithm should have recognized the operational datasets as in-distribution, are presented. In this case, as well, there were no classification errors. Due to the scarcity of data, only a few tests could be performed, and the only case with no errors at all was 30,10, the test was conducted with the maximum possible amount of data. For fewer splits, sporadic errors occurred, except for the 10,10 case, where the error still settled at around 3-4%. In the previous test case, the error settled at 0 starting from 30,10, but since there was no possibility to increase the number of images per split further, we can only speculate that it would have followed a similar trend. The fact that errors, more or less significant, occur below the 30,10 case could be due to the randomness of the images selected for each test.

- Best case: from 30,10 (0% error) with both norms.
- Worst case: 10,10 (approximately 3% error) with both norms.

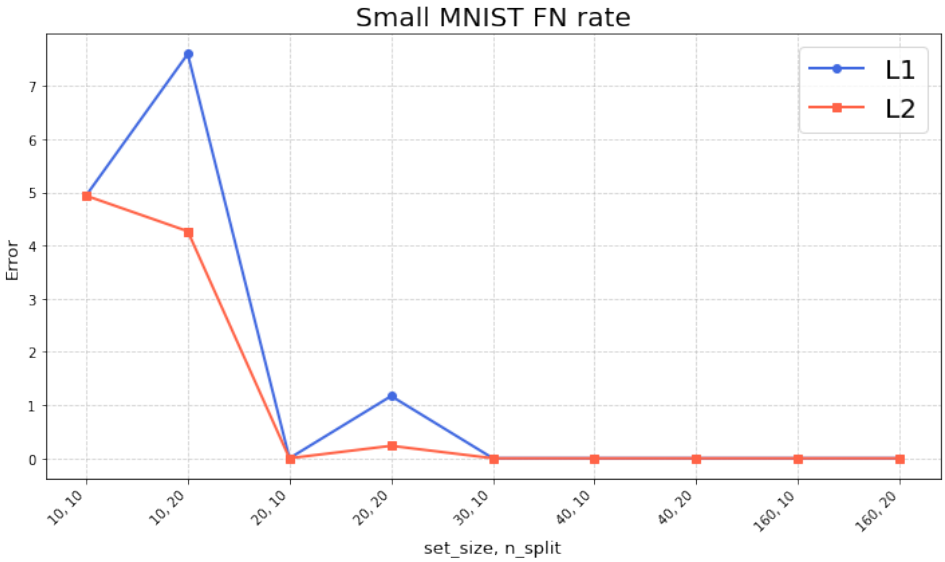


Figure 4.8: L1 and L2 norms errors for the Small MNIST dataset with an operational dataset that should be classified as OOD.

In Figure 4.10, the results of the test with the MNIST dataset, considering the class of digit 9 as OOD, are presented. With a much larger dataset available, tests with many more images and splits were conducted. The algorithm consistently managed to correctly classify the dataset, achieving 0% error in almost all tests. The only cases that differed from this behaviour were the ones where fewer images were used. However, even in these cases, classification was correct, and the error did not exceed 3% of the calculations on split distances. In this case, too, the error could be attributed to the variance between splits due to the small number of images selected for each split compared to the total number used to generate the rules.

- Best case: from 200,20 onwards (0% error) with both norms.
- Worst case: 10,10 (3% error) with L2 norm.

In Figure 4.11, the results of the test with the MNIST dataset, in which the algorithm should have recognized the operational datasets as in-distribution, are presented. This is also a unique case. Overall, the L1 norm performed better than the L2 norm, but there was never a case with 0 errors. However, errors were always very low, never exceeding 1.6%, and there were no classification errors. It is easily noticeable that, in all cases with 50 splits, the error approached 0, while the peaks were due to cases with only 20 splits. This could be due to the fact that, with 50 splits instead of 20, there is a higher probability of finding wider maximum and minimum values, causing the distances between training and operational rule hit tables to fall within this range. In this case, it seems that the number of images per split is not relevant.

- Best case: 200,50 with L1 norm, 700,50, and 1000,50 with both norms (error $< 0.1\%$).

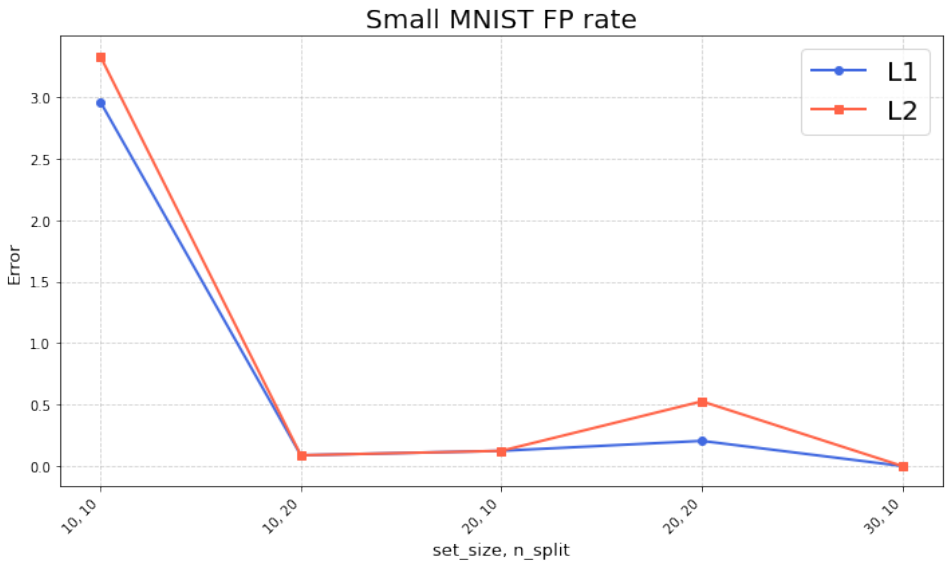


Figure 4.9: L1 and L2 norms errors for the Small MNIST dataset with an operational dataset that should be classified as In-Distribution.

- Worst case: all cases with 20 splits (error $\leq 1.6\%$).

In Figure 4.12, the results of the test with the CIFAR-10 dataset, considering images of trucks as OOD, are presented. Passing to an even larger dataset (in terms of features), one would expect a greater number of errors, which partially did not occur. Except for the two cases where only 50 images were used, cases that also resulted in many classification errors, once 200 images per split were reached, the error dropped below 2%, eventually reaching 0 from 300 images per split.

- Best case: from 300,50 onwards with both norms (0% error).
- Worst case: 50,50 with both norms (error $\geq 60\%$ and classification errors).

In Figure 4.13, the results of the test with the CIFAR-10 dataset, where the algorithm should have recognized the operational datasets as in-distribution, are presented. This is also a rather peculiar case because it is the only case where the error increases as the number of images increases. The case with fewer images and fewer splits, 50,20, is also the case with the lowest error percentage. However, the error remains very low until the 300,50 case, after which it rises (without causing classification errors) to about 5% in cases with 700 images. In the 1.000,20 case, the error exceeds 20%, causing a classification error with one of the classes. Finally, in the 1.000,50 case, the error drops to about 10%, without causing classification errors. The number of splits also seems to contribute to the results, as cases with fewer splits have higher errors. This is perhaps the most complex behaviour to explain due to the greater complexity of the dataset compared to the previous ones.

- Best case: 50,50 with both norms (error $\leq 0.1\%$).

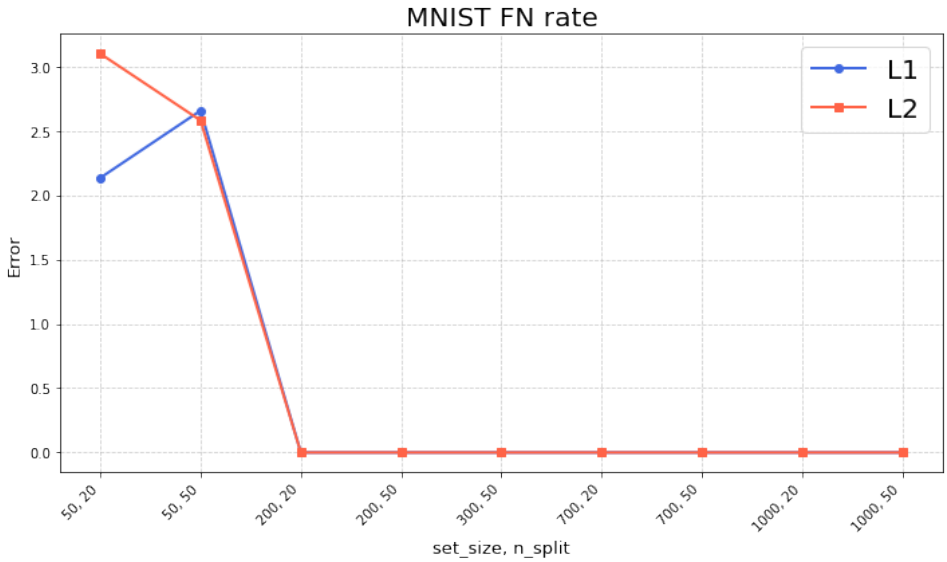


Figure 4.10: L1 and L2 norms errors for the MNIST dataset with an operational dataset that should be classified as OOD.

- Worst case: 1000,20 with both norms (error $>$ 20% and classification errors).

In Figure 4.14, the results of the test with the H&R dataset, considering the "normal" class as IN and all the anomalies classes as ODD. Although it appears that the L2 norm performs slightly better, the results are almost the same as in previous cases. With a small number of images (50) per split, the algorithm makes some errors, even committing classification errors. As for the classification errors, they are due to the fact that only one rule was generated for the classes that led to errors. Having only one rule generated makes it highly likely that the training images all satisfied that rule, which is not obvious even for those that were removed from the dataset before rule generation. In this case, even though the difference is very small, having a training rule hit table with all values equal to 1 leads to both the maximum and minimum distances between splits being 0. With a maximum and minimum distance of 0, any value that is not exactly 1 in the operational table is considered OOD. The error becomes 0 from the case of 200 images per split onwards.

- Best case: from 200,20 onwards with both norms (0% error).
- Worst case: 50,50 with both norms (error $\geq$ 5% and classification errors).

In Figure 4.15, the results of the test with the H&R dataset, where the algorithm should have recognized the operational datasets as in-distribution, are presented. In general, the error rate is very low in all the analysed cases, never exceeding 1%. As in the previous case of FN for H&R, some errors can be explained by the presence of certain classes for which only one rule was generated. Nevertheless, even in this case, the tests with 50 splits yielded better results than those with 20.

- Best case: 50,50 and 200,50 with both norms (error $\leq$ 0,07%).

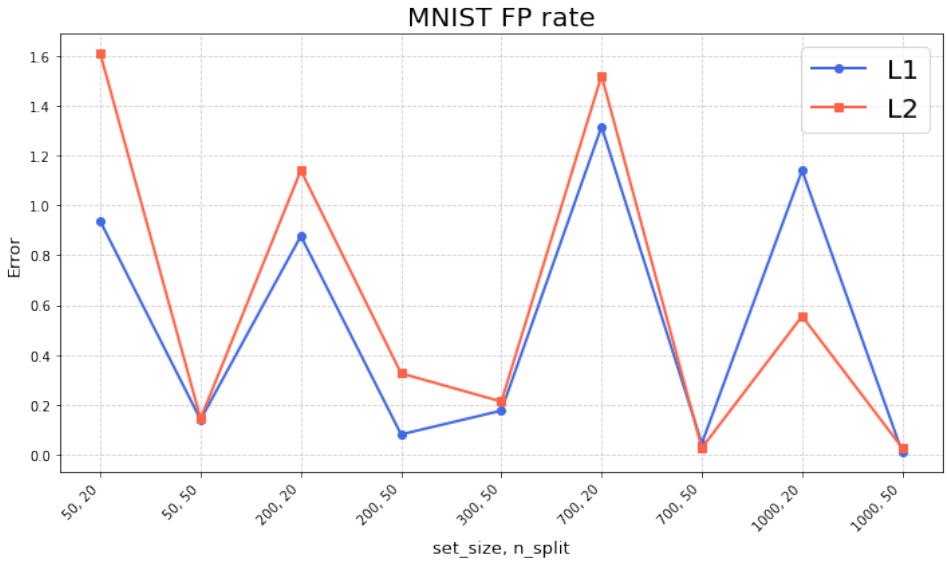


Figure 4.11: L1 and L2 norms errors for the Small MNIST dataset with an operational dataset that should be classified as In-Distribution.

- Worst case: all cases with 20 splits (error $\geq$ 0,2%).

Let us now examine the experiments carried out using a single operational division. It was observed that the outcomes from these tests were consistent, hence, they will be scrutinised together. Based on the conducted tests, it was inferred that in order for the algorithm to be effective with only one operational split, the count of operational images should coincide (or slightly differ) from that of the training splits. Initially, FN rates underwent assessment for all three datasets using 50 splits and 300 images per split for training and 100 for operational, yielding no errors. Even with reduced numbers of operational images, no errors occurred in the false negative rate tests.

However, the FP rate tests proved varied in results. Despite operating with 100 images, depending on randomly selected images, the algorithm either functioned perfectly or misclassified one of the ID classes. In the following tests, implementing identical amounts of images for each split during both the training and operational phases eliminated errors, even in high FP rate instances. Upon conducting trials with 300 images per split, we lowered the number to 50 images, which produced the same outcome.

The limitation may be due to greater variance in splits with fewer images. In the aforementioned tests with 1000 images per split, the values of each split were almost identical, resulting in an almost equal value distribution across all splits. Consequently, the maximum value obtained from the rule hit table during training was lower than in a scenario with only 300 images, where variability between the splits was higher. Hence, if one were to choose a varying number of images per split during training and operation, the comparison would entail splits with differing degrees of variability. This may result in erroneous classifications.

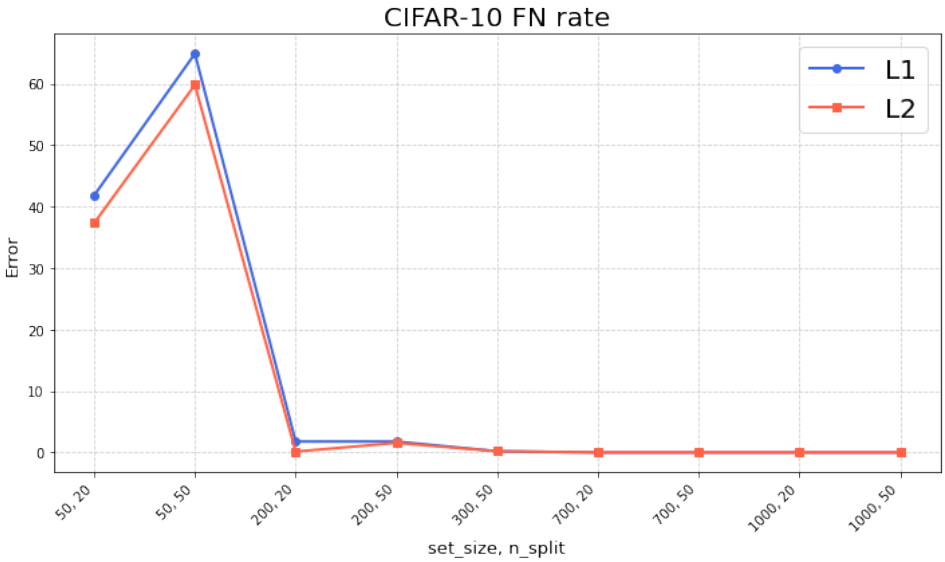


Figure 4.12: L1 and L2 norms errors for the CIFAR-10 dataset with an operational dataset that should be classified as OOD.

4.3.1 Summary of Results

In order to establish best practices for algorithm usage, it is necessary to examine whether a general behaviour can be identified from the tests conducted. The tests show that a low number of images per split consistently results in a high error rate. However, in my opinion, the magnitude of this error appears to be quite random. The degree of resulting error is impacted by the selection of images for the training and operational diagrams, which introduces randomness. When a limited number of images are used, splits within the same table can differ considerably, thereby leading to significantly greater disparities between the minimum and maximum distances. Another clear cause of errors was the classes generated with only one rule, as previously explained, which may easily result in classification errors.

It should be noted that during all conducted tests, particularly around and exceeding 30 images for the Small MNIST dataset and around 200/300 images for the other datasets, the error rate is generally very low. However, this figure will be dependent on the specific dataset in question. Additionally, it is evident that the results are influenced by the number of splits used; an increased number of splits provides better outcomes. This is demonstrated in tests of MNIST with respect to False Positives, as well as in the CIFAR-10 test, where once again False Positives are concerned.

Cases where only one rule per class is generated can be problematic. As explained previously, when only one rule describes a class, it often results in all images in the training dataset of that class satisfying the rule. Consequently, the resulting table will consist of only one row of '1's. This suggests that the distance between the individual segments of the training histogram will always be zero. Hence, an operational table that is not composed only of '1's will lead to an incorrect OOD, even with a minimal error. When considering a scenario where a dataset mainly comprises images that are considered to be in-distribution, a single image within

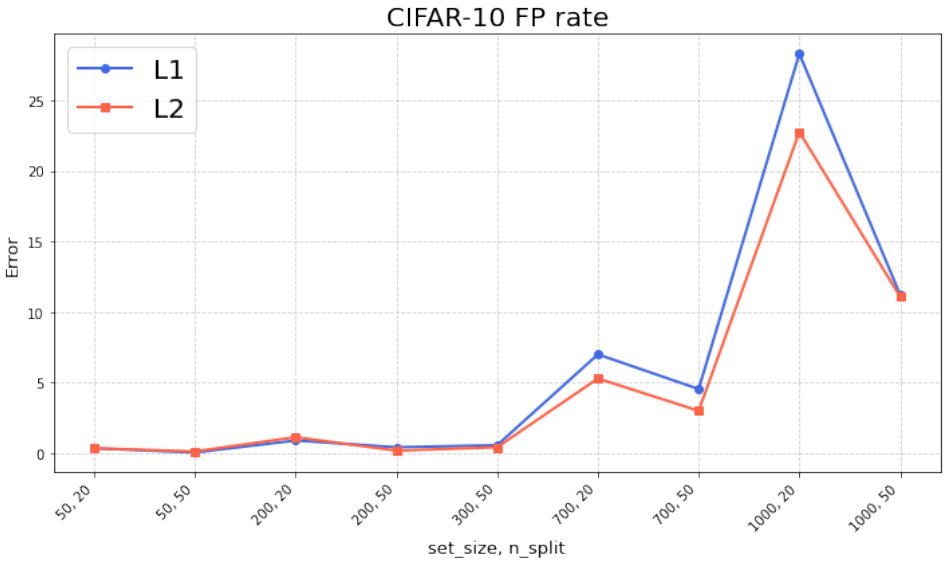


Figure 4.13: L1 and L2 norms errors for the CIFAR-10 dataset with an operational dataset that should be classified as In-Distribution.

the segments is enough to classify the segment as OOD.

Regarding tests conducted with a single operational split, the previous section has discussed the topic comprehensively, so I will only provide a summary of the findings. To avoid any discrepancies while comparing rule hit tables created with different variability indices, it is recommended to use an identical number of images in both training and operational sets (or a similar number). If we aimed to obtain 100 images for the operational dataset (2 seconds for a 50 frames per second camera), we would need to allocate an equivalent quantity of images for the training tasks. Moreover, we may need to augment the number of subsections if we aspire to employ a more extensive dataset for training.

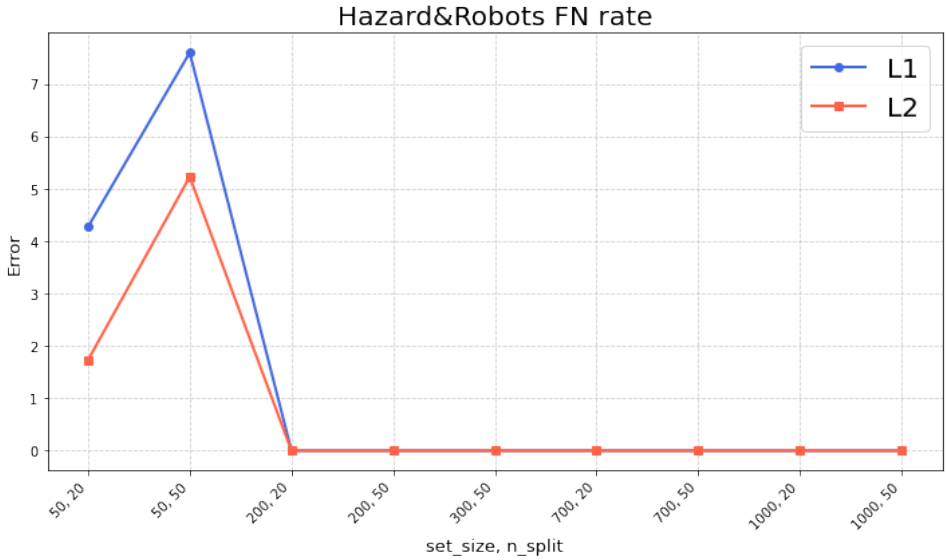


Figure 4.14: L1 and L2 norms errors for the H&R dataset with an operational dataset that should be classified as OOD.

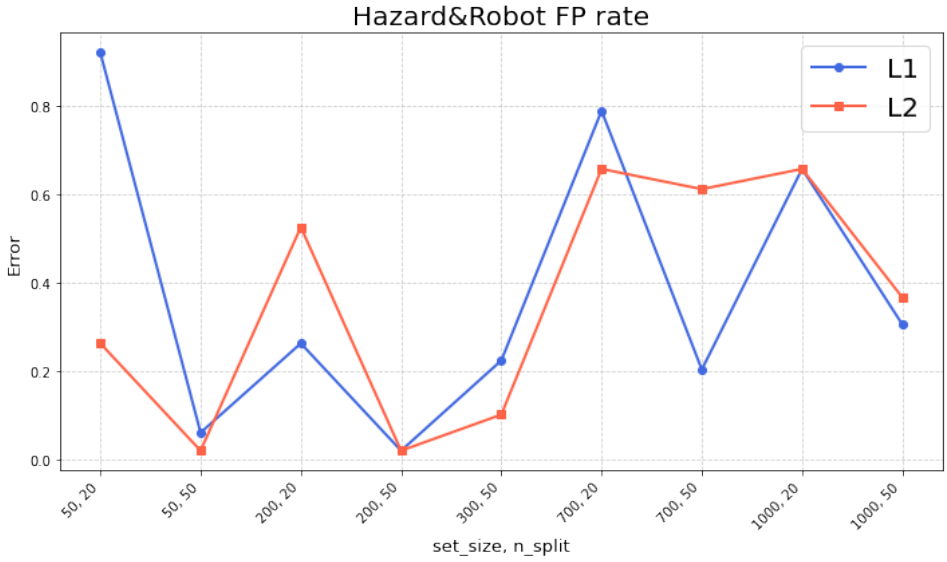


Figure 4.15: L1 and L2 norms errors for the H&R dataset with an operational dataset that should be classified as In-Distribution.

Chapter 5

Conclusions

The obtained results validate the effectiveness of the algorithm developed for this thesis, as it performs well on both benchmark and real datasets, and with either multiple or single operational splits. Notably, the H&R dataset with only one operational split is the most intriguing scenario for a potential future deployment, as it yielded impeccable outcomes when limiting the number of images per split.

Regarding potential real-world applications, further steps must be taken that were not taken in this study due to time constraints and the inability to conduct real-time camera testing. Initially, all tests were conducted by selecting images for the operational dataset and determining whether they were ID or OOD at that moment. In a practical scenario, evaluating datasets must be done continuously and in real-time by continually updating them. The oldest frame should be removed to make way for the latest recording captured by the camera. To avoid reapplying the algorithm from scratch, it is better to use some incremental method that changes only the operational rule hits table while keeping the rules and training table constant. Contrary to the approach taken during testing, a model should be trained using the UMAP library and then applied to every new frame captured by the camera. The previous frame's contribution to the operational rule hits table is subtracted, and the new frame's contribution is added.

Moving onto the issues that arose during testing, they require further investigation for future improvements.

Let us proceed systematically and start from UMAP. The investigation conducted insufficient tests on datasets, rendering it incapable of deriving any rule or mathematical model indicating the number of features to which original features could be reduced. In this study, we reduced the number of features in the Small MNIST from 64 to 3 (yet, even 2 would have sufficed (cf. Figure 4.7)), in the MNIST from 784 to 5, in the CIFAR-10 from 3072 to 10 and in the H&R from 512 to 4. Future research may explore varying the quantity of features to evaluate performance. Our aim was to minimize computational expenses by utilizing the smallest number of features possible. Moreover, similar to the H&R dataset, in cases where the pixel count is excessive, implementing a pre-processing technique is recommended to generate an initial reduction in features.

In the preceding hapter 4.3.1, it became evident that restricting the generation of only one rule per class can result in significant algorithmic errors. During the initial stages of the research, we attempted to force Rulex to produce an increased

number of rules beyond those it would naturally generate. Notably, when Rulex generates rules, it frequently generates more than it displays, subsequently filtering only those rules that surpass a certain threshold of coverage on the provided data. By applying a fixed number of rules, the program initially displays these rules with minimal coverage before producing rules that are essentially just modifications of the previous ones. It is evident that if only one rule is generated for a class, even with additional rules enforced, there will be no significant deviation from the initial rule. The methodology utilised in the initial stages resulted in a less accurate algorithm compared to the current version. This was due to the application of enforced rules which caused classification errors to occur. The issue of classes containing only one rule remains unresolved and can only be tackled by using larger datasets, which will enable Rulex to generate multiple rules.

Finally, the tests always used an operational dataset consisting entirely of one class. Currently, datasets with more than one class and where no class constitutes more than 50% of the dataset are not correctly classified. For instance, if an operational dataset contained 100% ID images but 33% of those images belonged to 3 different ID classes, the algorithm would consider it an OOD dataset as none of the ID classes reaches 50%. A technique must therefore be developed to identify such cases and ensure the algorithm remains operational.

Appendix A

Appendix

A.1 Code example

In this section, an example of code using Python and Matlab will be presented, analysed, and explained. The example will refer to the usage of the CIFAR-10 dataset. In this case, after reducing the number of features of the images from 1024 RGB pixels (3072 total pixels) to 10 using UMAP, rules will be generated using Rulex. These rules will then be transformed into functions usable in Python through a dedicated script. Subsequently, training and operational rule hit tables will be created, and their similarity will be evaluated. Out of the 10 classes in CIFAR-10, for simplicity, the first 9 (automobiles, airplanes, birds, cats, deer, dogs, frogs, horses, and ships) will be considered in-distribution, and the last one (trucks) will be considered out-of-distribution.

To avoid making the section overly lengthy and more accessible, library imports will not be included, and only essential code will be presented.

A.1.1 UMAP code

The code for reducing the number of features using the UMAP library is as follows:

```
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
assert x_train.shape == (50000, 32, 32, 3)
assert x_test.shape == (10000, 32, 32, 3)
assert y_train.shape == (50000, 1)
assert y_test.shape == (10000, 1)
x_train = np.reshape(x_train,(50000,3072))
x_train = x_train.astype('float32')
x_train /= 255
reducer_10D = umap.UMAP(n_components = 10, random_state = 42)
embedding_10D = reducer_10D.fit_transform(x_train)
CVS1 = pd.DataFrame(embedding_10D)
CVS1["Target"] = y_train
CVS1.to_csv(r'C:\Users\...\CIFAR10D.csv', index=False)
```

After downloading the entire CIFAR-10 dataset from the Keras library, the portion to be used (x_train) is reshaped from its original shape (5000, 32, 32, 3) to

the shape (5000, 3072). Then, the values of its features are normalized by dividing them by 255 (which corresponds to the number of colour levels for each pixel).

At this point, using UMAP, the features are reduced to 10 while setting a random_state for reproducibility of the reduction. Of course, reducing to a higher number of features would have worked equally well, while a lower number might lead to more confused and less isolated clusters. Since we cannot visualize more than 3 dimensions graphically, we will rely on the coverage that Rulex will provide us later on the data to understand whether the chosen number of features was sufficient to enable effective classification.

Finally, the newly reduced dataset is stored in a CSV file.

A.1.2 Training and operational datasets creation

For the creation of the rule hit tables, as already explained, we will need to have a training dataset (a dataset containing all images of the classes we consider in-distribution), which will be used to generate the rules, and an operational dataset (a dataset containing the images that our method should consider out-of-distribution in this specific example). Additionally, to create the tables, we will also need the datasets for each class. By taking all the data from the same dataset, we will create a dataset for each class (including the one we will use as OOD) and then create a unique dataset with all classes except the OOD class.

```
name_list = ["Airplane", "Automobile", "Bird", "Cat", "Deer", "Dog", "Frog"
    , "Horse", "Ship", "Truck"]
for j in range(0,10):
    data_train = pd.read_csv("CIFAR10D.csv")
    data = pd.DataFrame(data_train)
    OUTdata=pd.DataFrame()
    for i in range(len(data)):
        if data.Target[i] == j :
            OUTdata = OUTdata.append(data.iloc[i], ignore_index = True)
    name = '{}_Dataset.xlsx'.format(name_list[j])
    print(name)
    writer = pd.ExcelWriter(name, engine='xlsxwriter')
    OUTdata.to_excel(writer, index = False)
    writer.close()
INdata=pd.DataFrame()
for i in range(len(data)):
    if data.Target[i] != 9:
        INdata = INdata.append(data.iloc[i], ignore_index = True)
print('No_Truck_Dataset.xlsx')
writer = pd.ExcelWriter('No_Truck_Dataset.xlsx', engine='xlsxwriter')
INdata.to_excel(writer, index = False)
writer.close()
```

The code just shown will create a dataset for each class (including the OOD class) and then a combined dataset (excluding the OOD class) and save them all in separate files.

A.1.3 From Rulex C-like rules to Python rules

Starting from the dataset containing images of all classes (except the OOD class), Rulex creates a C-like formatted file containing the rules explaining the characteristics of each class. Since the file is created in a C-like format and the next steps of the algorithm are written in Python, the rules need to be transformed into a usable form. The following code has been written for this purpose:

```
import os
ruleset_C="No_forzature_rules.txt" #INSERIRE NOME FILE
i=0
newlines=[]
rules = 0
with open(ruleset_C,"r") as inf:
    lines = inf.readlines()
    lines = [e.replace("\n","") for e in lines]
    for l in lines:
        if l.startswith("#include"):
            continue
        elif l.startswith("const char *"):
            newl=l.replace("const char *ApplyRules","def ApplyRules"+str(i)
                ).replace("float ","").replace("{",": \n    i=0")
            newlines.append(newl+"\n")
            input_func = l.replace("const char *ApplyRules", "").replace("
                float ","").replace("{","")
        elif l.startswith("  if"):
            if not ("return \""+str(i)+"\"" in l):
                print("Class "+str(i)+", "+str(rules)+" rules")
                rules=1
                i+=1
                newl=l.replace("if","  return count_array \ndef ApplyRules"
                    +str(i)+""+str(input_func)+":\n    i=0\n    if").
                    replace("&&","and").replace("))",")):").replace(";","")
                    .replace("return \""+str(i)+"\"","\n        count_array
                    [i]=count_array[i]+1 \n        i+=1 \n    else: \n
                       i+=1")
            else:
                rules+=1
                newl=l.replace("if","  if").replace("&&","and").replace("))
                    ",")):").replace(";","").replace("return \""+str(i)+"\"
                    ","\n        count_array[i]=count_array[i]+1 \n
                    i+=1 \n    else: \n        i+=1")
            newlines.append(newl+"\n")
        elif l.startswith("}"):
            print("Class "+str(i)+", "+str(rules)+" rules")
            newl=l.replace("}", "    return count_array")
            newlines.append(newl)
        else:
```

```
            continue

fout = open(ruleset_C[:-2]+"_converted.txt","a")
fout.writelines(newlines)
fout.close()
```

The code requires as input only the name of the file created by Rulex. It transforms the format in which the rules were written by Rulex into the form of Python functions that will just need to be copied and pasted. Then, the code, prints on the screen the number of rules per class. The highest number of generated rules will be useful later in the code. Once the code is executed, the rules will have the following format:

```
def ApplyRules0(umap_0, umap_1, umap_2, umap_3, umap_4, umap_5, umap_6,
    umap_7, umap_8, umap_9) :
    i=0
    if ((umap_2 <= 7.526021) and (umap_3 > 3.257704 and umap_3 <= 5.543810)
         and (umap_8 <= 2.269972)):
        count_array[i]=count_array[i]+1
        i+=1
    else:
        i+=1
    if ((umap_2 <= 7.526021) and (umap_3 > 3.180035 and umap_3 <= 5.556519)
         and (umap_8 <= 2.269213)):
        count_array[i]=count_array[i]+1
        i+=1
    else:
        i+=1
    if ((umap_2 <= 7.526021) and (umap_3 > 3.184740 and umap_3 <= 5.561359)
         and (umap_8 <= 2.269213)):
        count_array[i]=count_array[i]+1
        i+=1
    else:
        i+=1
    if ((umap_2 <= 7.526021) and (umap_3 > 3.206564 and umap_3 <= 5.562153)
         and (umap_8 <= 2.269213)):
        count_array[i]=count_array[i]+1
        i+=1
    else:
        i+=1
    if ((umap_2 <= 7.544176) and (umap_3 <= 5.243007) and (umap_8 <=
        2.269213)):
        count_array[i]=count_array[i]+1
        i+=1
    else:
        ...
        return count_array
```

To avoid needlessly extending the section, only a small part of a single function has been presented. However, since the dataset consists of 9 classes that we have considered in-distribution, 8 more functions similar to the one above will be created, each with its specific rules. The purpose of these functions is simply to act as counters to determine within a split how many times each rule is satisfied by the images in that split.

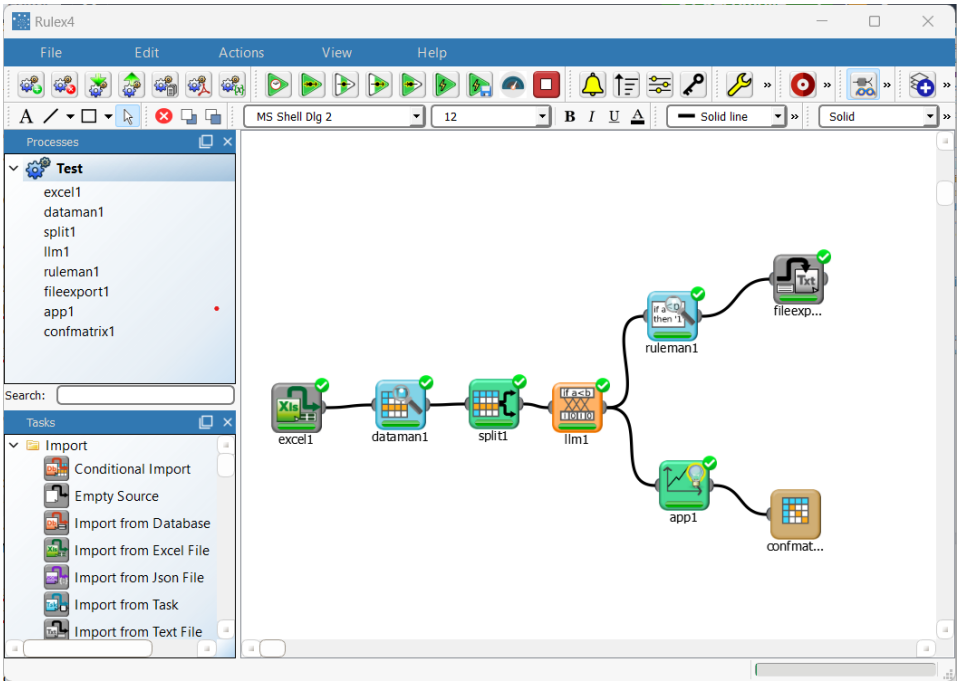


Figure A.1: Rulex processes scheme

A.1.4 Rule hits tables building

Once the functioning of the *ApplyRules* functions is understood, let's now see how rule hits tables are actually created. The following code generates the tables for the training data:

```
for z in range(0,9):
    nome1 = '{}_Dataset.xlsx'.format(name_list[z])
    data_1 = pd.read_excel(nome1)
    #Crea i subset
    lista_set_training=[];
    set_size = 300;
    contatore=0
    while contatore<20:
        lista_set_training.append(data_1.sample(n=set_size,replace=False))
        lista_set_training[contatore]=lista_set_training[contatore].
            reset_index(drop=True)
        contatore=contatore+1;
    lista_array_conteggi=[];
```

```
    for j in np.arange(contatore):
        count_array=np.zeros(26)
        for i in range(set_size):
            if z == 0:
                array_conteggio=(ApplyRules0(lista_set_training[j]["umap_0"
                    ][i], lista_set_training[j]["umap_1"][i],
                    lista_set_training[j]["umap_2"][i],lista_set_training[j
                    ]["umap_3"][i],lista_set_training[j]["umap_4"][i],
                    lista_set_training[j]["umap_5"][i],lista_set_training[j
                    ]["umap_6"][i],lista_set_training[j]["umap_7"][i],
                    lista_set_training[j]["umap_8"][i],lista_set_training[j
                    ]["umap_9"][i]))
            elif z == 1:
                array_conteggio=(ApplyRules1(lista_set_training[j]["umap_0"
                    ][i], lista_set_training[j]["umap_1"][i],
                    lista_set_training[j]["umap_2"][i],lista_set_training[j
                    ]["umap_3"][i],lista_set_training[j]["umap_4"][i],
                    lista_set_training[j]["umap_5"][i],lista_set_training[j
                    ]["umap_6"][i],lista_set_training[j]["umap_7"][i],
                    lista_set_training[j]["umap_8"][i],lista_set_training[j
                    ]["umap_9"][i]))
            elif z == 2:
                array_conteggio=(ApplyRules2(lista_set_training[j]["umap_0"
                    ][i], lista_set_training[j]["umap_1"][i],
                    lista_set_training[j]["umap_2"][i],lista_set_training[j
                    ]["umap_3"][i],lista_set_training[j]["umap_4"][i],
                    lista_set_training[j]["umap_5"][i],lista_set_training[j
                    ]["umap_6"][i],lista_set_training[j]["umap_7"][i],
                    lista_set_training[j]["umap_8"][i],lista_set_training[j
                    ]["umap_9"][i]))
            elif z == 3:
                array_conteggio=(ApplyRules3(lista_set_training[j]["umap_0"
                    ][i], lista_set_training[j]["umap_1"][i],
                    lista_set_training[j]["umap_2"][i],lista_set_training[j
                    ]["umap_3"][i],lista_set_training[j]["umap_4"][i],
                    lista_set_training[j]["umap_5"][i],lista_set_training[j
                    ]["umap_6"][i],lista_set_training[j]["umap_7"][i],
                    lista_set_training[j]["umap_8"][i],lista_set_training[j
                    ]["umap_9"][i]))
            elif z == 4:
                array_conteggio=(ApplyRules4(lista_set_training[j]["umap_0"
                    ][i], lista_set_training[j]["umap_1"][i],
                    lista_set_training[j]["umap_2"][i],lista_set_training[j
                    ]["umap_3"][i],lista_set_training[j]["umap_4"][i],
                    lista_set_training[j]["umap_5"][i],lista_set_training[j
                    ]["umap_6"][i],lista_set_training[j]["umap_7"][i],
                    lista_set_training[j]["umap_8"][i],lista_set_training[j
```

```
                    ]["umap_9"][i]))
            elif z == 5:
                array_conteggio=(ApplyRules5(lista_set_training[j]["umap_0"
                    ][i], lista_set_training[j]["umap_1"][i],
                    lista_set_training[j]["umap_2"][i],lista_set_training[j
                    ]["umap_3"][i],lista_set_training[j]["umap_4"][i],
                    lista_set_training[j]["umap_5"][i],lista_set_training[j
                    ]["umap_6"][i],lista_set_training[j]["umap_7"][i],
                    lista_set_training[j]["umap_8"][i],lista_set_training[j
                    ]["umap_9"][i]))
            elif z == 6:
                array_conteggio=(ApplyRules6(lista_set_training[j]["umap_0"
                    ][i], lista_set_training[j]["umap_1"][i],
                    lista_set_training[j]["umap_2"][i],lista_set_training[j
                    ]["umap_3"][i],lista_set_training[j]["umap_4"][i],
                    lista_set_training[j]["umap_5"][i],lista_set_training[j
                    ]["umap_6"][i],lista_set_training[j]["umap_7"][i],
                    lista_set_training[j]["umap_8"][i],lista_set_training[j
                    ]["umap_9"][i]))
            elif z == 7:
                array_conteggio=(ApplyRules7(lista_set_training[j]["umap_0"
                    ][i], lista_set_training[j]["umap_1"][i],
                    lista_set_training[j]["umap_2"][i],lista_set_training[j
                    ]["umap_3"][i],lista_set_training[j]["umap_4"][i],
                    lista_set_training[j]["umap_5"][i],lista_set_training[j
                    ]["umap_6"][i],lista_set_training[j]["umap_7"][i],
                    lista_set_training[j]["umap_8"][i],lista_set_training[j
                    ]["umap_9"][i]))
            elif z == 8:
                array_conteggio=(ApplyRules8(lista_set_training[j]["umap_0"
                    ][i], lista_set_training[j]["umap_1"][i],
                    lista_set_training[j]["umap_2"][i],lista_set_training[j
                    ]["umap_3"][i],lista_set_training[j]["umap_4"][i],
                    lista_set_training[j]["umap_5"][i],lista_set_training[j
                    ]["umap_6"][i],lista_set_training[j]["umap_7"][i],
                    lista_set_training[j]["umap_8"][i],lista_set_training[j
                    ]["umap_9"][i]))
        lista_array_conteggi.append((array_conteggio/set_size));
    df_training=pd.DataFrame()
    for i in np.arange(20):
        df_training['ID'+str(i)]=lista_array_conteggi[i]
    df_training.to_excel('{}_IN.xlsx'.format(name_list[z]), index=False)
```

A *for* loop iterates through the datasets of individual classes. For each class, splits are created (the number of splits is defined by the *contatore* variable) consisting of n elements (*set_size*). For each created split, another 'for' loop iterates through the individual images, creating a matrix of the size of the class with the most rules

(even if a class were to have 10 rules and the matrix were created with 15 rows, the last 5 rows would always be 0, both for the training and operational table, therefore not causing any issues), and for each image, the corresponding ApplyRules function for the image's class is called. Once the counts are completed, everything is normalized based on the *set_size*. After all splits are finished, the table is saved to a xlsx file, and the process proceeds with the next class.

After the creation of training tables is completed, the process continues with operational ones.

```
for z in range(0,9):
    data_1 = pd.read_excel('Truck_Dataset.xlsx')
    #Crea i subset
    lista_set_training=[];
    set_size = 300;
    contatore=0
    while contatore<20:
        lista_set_training.append(data_1.sample(n=set_size,replace=False))
        lista_set_training[contatore]=lista_set_training[contatore].
            reset_index(drop=True)
        contatore=contatore+1;
    lista_array_conteggi=[];
    for j in np.arange(contatore):
        count_array=np.zeros(26);
        for i in range(set_size):
            if z == 0:
                array_conteggio=(ApplyRules0(lista_set_training[j]["umap_0"
                    ][i], lista_set_training[j]["umap_1"][i],
                    lista_set_training[j]["umap_2"][i],lista_set_training[j
                    ]["umap_3"][i],lista_set_training[j]["umap_4"][i],
                    lista_set_training[j]["umap_5"][i],lista_set_training[j
                    ]["umap_6"][i],lista_set_training[j]["umap_7"][i],
                    lista_set_training[j]["umap_8"][i],lista_set_training[j
                    ]["umap_9"][i]))
            elif z == 1:
                array_conteggio=(ApplyRules1(lista_set_training[j]["umap_0"
                    ][i], lista_set_training[j]["umap_1"][i],
                    lista_set_training[j]["umap_2"][i],lista_set_training[j
                    ]["umap_3"][i],lista_set_training[j]["umap_4"][i],
                    lista_set_training[j]["umap_5"][i],lista_set_training[j
                    ]["umap_6"][i],lista_set_training[j]["umap_7"][i],
                    lista_set_training[j]["umap_8"][i],lista_set_training[j
                    ]["umap_9"][i]))
            elif z == 2:
                array_conteggio=(ApplyRules2(lista_set_training[j]["umap_0"
                    ][i], lista_set_training[j]["umap_1"][i],
                    lista_set_training[j]["umap_2"][i],lista_set_training[j
                    ]["umap_3"][i],lista_set_training[j]["umap_4"][i],
                    lista_set_training[j]["umap_5"][i],lista_set_training[j
```

```
                ]["umap_6"][i],lista_set_training[j]["umap_7"][i],
                lista_set_training[j]["umap_8"][i],lista_set_training[j
                ]["umap_9"][i]))
21          elif z == 3:
22              array_conteggio=(ApplyRules3(lista_set_training[j]["umap_0"
                    ][i], lista_set_training[j]["umap_1"][i],
                    lista_set_training[j]["umap_2"][i],lista_set_training[j
                    ]["umap_3"][i],lista_set_training[j]["umap_4"][i],
                    lista_set_training[j]["umap_5"][i],lista_set_training[j
                    ]["umap_6"][i],lista_set_training[j]["umap_7"][i],
                    lista_set_training[j]["umap_8"][i],lista_set_training[j
                    ]["umap_9"][i]))
23          elif z == 4:
24              array_conteggio=(ApplyRules4(lista_set_training[j]["umap_0"
                    ][i], lista_set_training[j]["umap_1"][i],
                    lista_set_training[j]["umap_2"][i],lista_set_training[j
                    ]["umap_3"][i],lista_set_training[j]["umap_4"][i],
                    lista_set_training[j]["umap_5"][i],lista_set_training[j
                    ]["umap_6"][i],lista_set_training[j]["umap_7"][i],
                    lista_set_training[j]["umap_8"][i],lista_set_training[j
                    ]["umap_9"][i]))
25          elif z == 5:
26              array_conteggio=(ApplyRules5(lista_set_training[j]["umap_0"
                    ][i], lista_set_training[j]["umap_1"][i],
                    lista_set_training[j]["umap_2"][i],lista_set_training[j
                    ]["umap_3"][i],lista_set_training[j]["umap_4"][i],
                    lista_set_training[j]["umap_5"][i],lista_set_training[j
                    ]["umap_6"][i],lista_set_training[j]["umap_7"][i],
                    lista_set_training[j]["umap_8"][i],lista_set_training[j
                    ]["umap_9"][i]))
27          elif z == 6:
28              array_conteggio=(ApplyRules6(lista_set_training[j]["umap_0"
                    ][i], lista_set_training[j]["umap_1"][i],
                    lista_set_training[j]["umap_2"][i],lista_set_training[j
                    ]["umap_3"][i],lista_set_training[j]["umap_4"][i],
                    lista_set_training[j]["umap_5"][i],lista_set_training[j
                    ]["umap_6"][i],lista_set_training[j]["umap_7"][i],
                    lista_set_training[j]["umap_8"][i],lista_set_training[j
                    ]["umap_9"][i]))
29          elif z == 7:
30              array_conteggio=(ApplyRules7(lista_set_training[j]["umap_0"
                    ][i], lista_set_training[j]["umap_1"][i],
                    lista_set_training[j]["umap_2"][i],lista_set_training[j
                    ]["umap_3"][i],lista_set_training[j]["umap_4"][i],
                    lista_set_training[j]["umap_5"][i],lista_set_training[j
                    ]["umap_6"][i],lista_set_training[j]["umap_7"][i],
                    lista_set_training[j]["umap_8"][i],lista_set_training[j
```

```
                    ]["umap_9"][i]))
            elif z == 8:
                array_conteggio=(ApplyRules8(lista_set_training[j]["umap_0"
                    ][i], lista_set_training[j]["umap_1"][i],
                    lista_set_training[j]["umap_2"][i],lista_set_training[j
                    ]["umap_3"][i],lista_set_training[j]["umap_4"][i],
                    lista_set_training[j]["umap_5"][i],lista_set_training[j
                    ]["umap_6"][i],lista_set_training[j]["umap_7"][i],
                    lista_set_training[j]["umap_8"][i],lista_set_training[j
                    ]["umap_9"][i]))
        lista_array_conteggi.append((array_conteggio/set_size));

    df_training=pd.DataFrame()
    for i in np.arange(20):
        df_training['ID'+str(i)]=lista_array_conteggi[i]

    df_training.to_excel('Truck_{}_OUT.xlsx'.format(name_list[z]), index=
        False)
```

The procedure is very similar: In this case, as well, splits are created just like in the previous code. However, in this case, the initial *for* loop serves the purpose of comparing the operational dataset with the rules of each class. Unlike before, we won't have a dataset for each class to compare with their corresponding rules. Instead, the operational dataset will be compared with the rules of each class to determine if it belongs to any of them. Similarly, once the counts are completed, everything is normalized based on the number of elements in the *set_size*. Finally, all the data is written to a xlsx file.

A.1.5 ODD

The last code provided is for the ODD.

```
filename = "Results_L1_Norm_noTruck.txt";
name_list = ["Airplane", "Automobile", "Bird", "Cat" , "Deer", "Dog", "Frog
    ", "Horse", "Ship", "Truck"];
for z = 0:8
    table = "_IN.xlsx";
    newtable = insertBefore(table,'_IN.xlsx',name_list(z+1));
    table1 = "_OUT.xlsx";
    newtable1 = insertBefore(table1,'_OUT.xlsx',name_list(z+1));
    data=readtable(newtable);
    data=data{:,:};
    data=array2table(data);
    data1=readtable(newtable1);
    data1=data1{:,:};
    data1=array2table(data1);

    n_h=size(data,2);
```

```
    ix34=1:n_h;
    iy34=1:n_h;
    for i=1:length(ix34)
        hitx=data{:,ix34(i)};
        for j=i:1:n_h
             hity=data{:,iy34(j)};
             mi34(i,j)=norm(abs(hity-hitx),1);
         end
    end
    minimo=min(mi34(:));
    massimo=max(mi34(:));
    ------------------------------------------------------------
    ix=ix34;
    somma1=0;
    somma2=0;
    for k=1:size(data1,2)
        somma1=0;
        for i=1:length(ix)
                hitx=data{:,ix(i)};
                hity=data1{:,k};
                mi(i)=norm(abs(hity-hitx),1);
                if  mi(i)<minimo ||   mi(i)>massimo
                    somma1=somma1+1;
                end
                MImatrix(i,k)=mi(i);
        end
        if somma1>fix(size(data,2)/2)
            somma2=somma2+1;
        end
    end
    ------------------------------------------------------------
    if somma2<fix(size(data,2)/2)
        result = "In-Distribution"+name_list(z+1)
        return;
    else
        result = "Out-of-Distribution"+name_list(z+1)
    end
    line = name_list(z+1)+"_IN VS "+name_list(z+1)+"_OUT:";
    writelines(line,filename,WriteMode="append")
    line=name_list(z+1)+": out somma1 = "+somma1+", somma2 = "+somma2;
    writelines(line,filename,WriteMode="append")
end
```

As previously explained in the preceding chapter (3.3), the ODD algorithm can be divided into three distinct parts (marked with $---$). In the first part, for each class, only the training table will be utilized: a nested double *for* loop will iterate through the columns of the table and apply a norm (in this example, the L1

norm, but it could also be the L2 norm or a combination of multiple methods) to all possible split combinations, saving the results. Among these results, the maximum and minimum values will be selected, which will be used to determine whether the subsequently analysed data is in-distribution or not. The second part of the code involves both the training and operational tables. In this case, the nested double *for* loop will ensure that the norm is applied to every combination of columns between the two tables. If the calculated difference for each split exceeds the range defined by the previously computed minimum and maximum values, the data will be considered OOD, and the variable $sum1$ will be incremented by one. At the end of the inner *for* loop, if more than 50% of the splits are determined to be OOD, the first split of the operational table will be considered OOD. At the conclusion of all the others *for* cycles, if more than 50% of the splits are deemed OOD, the table will be considered OOD. The third and final part's sole purpose is to terminate the program in case the table is found to be in-distribution for the currently analysed class, or otherwise confirm that the analysed table is OOD concerning that class and continue the loop for subsequent classes if present.

There is a slight modification that could be made to the code. As it's currently written, the calculated minimum value in the first part of the code is redundant: since the norm is computed between every split combination, it will also be calculated when comparing a split to itself, resulting in 0. Since norms cannot produce negative values, it's evident that the condition where a distance would be smaller than the minimum value during the comparison of training and operational tables in the second part is impossible.

One potential modification would be to avoid applying the norm between a split and itself. However, this modification could lead to false positives if an operational table is used that is based on the same dataset used to create the training table. Nonetheless, it has been decided to retain it, as it might prove useful if a different method for calculating distance other than norms were to be used.

Bibliography

[1] Kaiming He et al. “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification”. In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.

[2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25 (2012).

[3] Nick Drummond and Rob Shearer. “The open world assumption”. In: *eSI Workshop: The Closed World of Databases meets the Open World of the Semantic Web*. Vol. 15. 2006, p. 1.

[4] Jingkang Yang et al. “Generalized out-of-distribution detection: A survey”. In: *arXiv preprint arXiv:2110.11334* (2021).

[5] *EASA Concept Paper First usable guidance for Level 1 machine learning applications - Proposed Issue 01.pdf | EASA — easa.europa.eu*. https://www.easa.europa.eu/en/easa-concept-paper-first-usable-guidance-level-1-machine-learning-applications-proposed-issue-01pdf. [Accessed 05-08-2023].

[6] Giacomo De Bernardi et al. “Rule-based out-of-distribution detection”. In: *arXiv preprint arXiv:2303.01860* (2023).

[7] Agnieszka Mikołajczyk and Michał Grochowski. “Data augmentation for improving deep learning in image classification problem”. In: *2018 international interdisciplinary PhD workshop (IIPhDW)*. IEEE. 2018, pp. 117–122.

[8] Kimin Lee et al. “A simple unified framework for detecting out-of-distribution samples and adversarial attacks”. In: *Advances in neural information processing systems* 31 (2018).

[9] Julian Bitterwolf et al. “Breaking down out-of-distribution detection: Many methods based on ood training data estimate a combination of the same core quantities”. In: *International Conference on Machine Learning*. PMLR. 2022, pp. 2041–2074.

[10] Yiyou Sun et al. “Out-of-distribution detection with deep nearest neighbors”. In: *International Conference on Machine Learning*. PMLR. 2022, pp. 20827–20840.

[11] Anh Nguyen, Jason Yosinski, and Jeff Clune. “Deep neural networks are easily fooled: High confidence predictions for unrecognizable images”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 427–436.

[12] Abhijit Bendale and Terrance Boult. "Towards open world recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1893–1902.

[13] Dan Hendrycks and Kevin Gimpel. "A baseline for detecting misclassified and out-of-distribution examples in neural networks". In: *arXiv preprint arXiv:1610.02136* (2016).

[14] Shiyu Liang, Yixuan Li, and Rayadurgam Srikant. "Enhancing the reliability of out-of-distribution image detection in neural networks". In: *arXiv preprint arXiv:1706.02690* (2017).

[15] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. "Simple and scalable predictive uncertainty estimation using deep ensembles". In: *Advances in neural information processing systems* 30 (2017).

[16] Weitang Liu et al. "Energy-based out-of-distribution detection". In: *Advances in neural information processing systems* 33 (2020), pp. 21464–21475.

[17] Jimmy Lin and Xueguang Ma. "A few brief notes on deepimpact, coil, and a conceptual framework for information retrieval techniques". In: *arXiv preprint arXiv:2106.14807* (2021).

[18] Haoran Wang et al. "Can multi-label classification networks know what they don't know?" In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 29074–29087.

[19] Peyman Morteza and Yixuan Li. "Provable guarantees for understanding out-of-distribution detection". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 36. 7. 2022, pp. 7831–7840.

[20] Yiyou Sun, Chuan Guo, and Yixuan Li. "React: Out-of-distribution detection with rectified activations". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 144–157.

[21] Rui Huang, Andrew Geng, and Yixuan Li. "On the importance of gradients for detecting distributional shifts in the wild". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 677–689.

[22] Haoqi Wang et al. "Vim: Out-of-distribution with virtual-logit matching". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2022, pp. 4921–4930.

[23] Rui Huang and Yixuan Li. "Mos: Towards scaling out-of-distribution detection for large semantic space". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, pp. 8710–8719.

[24] Kimin Lee et al. "Training confidence-calibrated classifiers for detecting out-of-distribution samples". In: *arXiv preprint arXiv:1711.09325* (2017).

[25] Taewon Jeong and Heeyoung Kim. "OOD-MAML: Meta-learning for few-shot out-of-distribution detection and classification". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 3907–3916.

[26] Joost Van Amersfoort et al. "Uncertainty estimation using a single deep deterministic neural network". In: *International conference on machine learning*. PMLR. 2020, pp. 9690–9700.

[27] Jingkang Yang et al. “Semantically coherent out-of-distribution detection”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2021, pp. 8301–8309.

[28] Jiefeng Chen et al. “Atom: Robustifying out-of-distribution detection using outlier mining”. In: *Machine Learning and Knowledge Discovery in Databases. Research Track: European Conference, ECML PKDD 2021, Bilbao, Spain, September 13–17, 2021, Proceedings, Part III 21*. Springer. 2021, pp. 430–445.

[29] Hongxin Wei et al. “Mitigating neural network overconfidence with logit normalization”. In: *International Conference on Machine Learning*. PMLR. 2022, pp. 23631–23644.

[30] Yifei Ming, Ying Fan, and Yixuan Li. “Poem: Out-of-distribution detection with posterior sampling”. In: *International Conference on Machine Learning*. PMLR. 2022, pp. 15650–15665.

[31] Julian Katz-Samuels et al. “Training ood detectors in their natural habitats”. In: *International Conference on Machine Learning*. PMLR. 2022, pp. 10848–10865.

[32] Petra Bevandić et al. “Discriminative out-of-distribution detection for semantic segmentation”. In: *arXiv preprint arXiv:1808.07703* (2018).

[33] Andrey Malinin and Mark Gales. “Predictive uncertainty estimation via prior networks”. In: *Advances in neural information processing systems* 31 (2018).

[34] Dan Hendrycks, Mantas Mazeika, and Thomas Dietterich. “Deep anomaly detection with outlier exposure”. In: *arXiv preprint arXiv:1812.04606* (2018).

[35] Yonatan Geifman and Ran El-Yaniv. “Selectivenet: A deep neural network with an integrated reject option”. In: *International conference on machine learning*. PMLR. 2019, pp. 2151–2159.

[36] Matthias Hein, Maksym Andriushchenko, and Julian Bitterwolf. “Why relu networks yield high-confidence predictions far away from the training data and how to mitigate the problem”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 41–50.

[37] Alexander Meinke and Matthias Hein. “Towards neural networks that provably know when they don’t know”. In: *arXiv preprint arXiv:1909.12180* (2019).

[38] Sina Mohseni et al. “Self-supervised learning for generalizable out-of-distribution detection”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 04. 2020, pp. 5216–5223.

[39] Xuefeng Du et al. *Unknown-Aware Object Detection: Learning What You Don’t Know from Videos in the Wild*. 2022. arXiv: 2203.03800 [cs.CV].

[40] Xuefeng Du et al. *VOS: Learning What You Don’t Know by Virtual Outlier Synthesis*. 2022. arXiv: 2202.01197 [cs.LG].

[41] Jihoon Tack et al. “Csi: Novelty detection via contrastive learning on distributionally shifted instances”. In: *Advances in neural information processing systems* 33 (2020), pp. 11839–11852.

[42] Jim Winkens et al. “Contrastive training for improved out-of-distribution detection”. In: *arXiv preprint arXiv:2007.05566* (2020).

[43] Vikash Sehwag, Mung Chiang, and Prateek Mittal. “Ssd: A unified framework for self-supervised outlier detection”. In: *arXiv preprint arXiv:2103.12051* (2021).

[44] Ting Chen et al. “A simple framework for contrastive learning of visual representations”. In: *International conference on machine learning*. PMLR. 2020, pp. 1597–1607.

[45] Prannay Khosla et al. “Supervised contrastive learning”. In: *Advances in neural information processing systems* 33 (2020), pp. 18661–18673.

[46] Yifei Ming et al. “How to Exploit Hyperspherical Embeddings for Out-of-Distribution Detection?” In: *arXiv preprint arXiv:2203.04450* (2022).

[47] Jing Tian, Michael H Azarian, and Michael Pecht. “Anomaly detection using self-organizing maps-based k-nearest neighbor algorithm”. In: *PHM society European conference*. Vol. 2. 1. 2014.

[48] Puning Zhao and Lifeng Lai. “Analysis of knn density estimation”. In: *IEEE Transactions on Information Theory* 68.12 (2022), pp. 7971–7995.

[49] Liron Bergman, Niv Cohen, and Yedid Hoshen. “Deep nearest neighbor anomaly detection”. In: *arXiv preprint arXiv:2002.10445* (2020).

[50] Taurus T Dang, Henry YT Ngan, and Wei Liu. “Distance-based k-nearest neighbors outlier detection method in large-scale traffic data”. In: *2015 IEEE International Conference on Digital Signal Processing (DSP)*. IEEE. 2015, pp. 507–510.

[51] Xiaoyi Gu, Leman Akoglu, and Alessandro Rinaldo. “Statistical analysis of nearest neighbor methods for anomaly detection”. In: *Advances in Neural Information Processing Systems* 32 (2019).

[52] Catarina Pires et al. “Towards knowledge uncertainty estimation for open set recognition”. In: *Machine Learning and Knowledge Extraction* 2.4 (2020), pp. 505–532.

[53] Matteo Guarrera et al. “Class-wise thresholding for robust out-of-distribution detection”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022, pp. 2837–2846.

[54] Leland McInnes, John Healy, and James Melville. “Umap: Uniform manifold approximation and projection for dimension reduction”. In: *arXiv preprint arXiv:1802.03426* (2018).

[55] Laurens Van der Maaten and Geoffrey Hinton. “Visualizing data using t-SNE.” In: *Journal of machine learning research* 9.11 (2008).

[56] Wei Dong, Charikar Moses, and Kai Li. “Efficient k-nearest neighbor graph construction for generic similarity measures”. In: *Proceedings of the 20th international conference on World wide web*. 2011, pp. 577–586.

[57] Nikhil Ketkar and Nikhil Ketkar. “Stochastic gradient descent”. In: *Deep learning with Python: A hands-on introduction* (2017), pp. 113–132.

[58] Quan Zheng and David B Skillicorn. “Spectral embedding of signed networks”. In: *Proceedings of the 2015 SIAM international conference on data mining*. SIAM. 2015, pp. 55–63.

[59] Guoliang Xu. "Discrete Laplace–Beltrami operators and their convergence". In: *Computer aided geometric design* 21.8 (2004), pp. 767–784.

[60] F. Alimoglu. "Combining Multiple Classifiers for Pen-Based Handwritten Digit Recognition". In: (1996).

[61] F. Alimoglu and E. Alpaydin. "Methods of Combining Multiple Classifiers Based on Different Representations for Pen-based Handwriting Recognition". In: (1996).

[62] Marco Muselli. "Switching neural networks: A new connectionist model for classification". In: *Italian Workshop on Neural Nets*. Springer. 2005, pp. 23–30.

[63] Stefano Parodi et al. "Identifying environmental and social factors predisposing to pathological gambling combining standard logistic regression and logic learning machine". In: *Journal of Gambling Studies* 33 (2017), pp. 1121–1137.

[64] Stefano Parodi et al. "Differential diagnosis of pleural mesothelioma using Logic Learning Machine". In: *BMC bioinformatics* 16 (2015), pp. 1–10.

[65] Stefano Parodi et al. "Logic Learning Machine and standard supervised methods for Hodgkin's lymphoma prognosis using gene expression data and clinical variables". In: *Health Informatics Journal* 24.1 (2018), pp. 54–65.

[66] Marco Muselli. *Switching Neural Networks: A New Connectionist Model for Classification*. Jan. 2005.

[67] Li Deng. "The mnist database of handwritten digit images for machine learning research [best of the web]". In: *IEEE signal processing magazine* 29.6 (2012), pp. 141–142.

[68] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[69] Francois Chollet et al. *Keras*. 2015. URL: https://github.com/fchollet/keras.

[70] Dario Mantegazza et al. "An Outlier Exposure Approach to Improve Visual Anomaly Detection Performance for Mobile Robots." In: *IEEE Robotics and Automation Letters* 7.4 (2022), pp. 11354–11361. DOI: 10.1109/LRA.2022.3192794.

[71] Greg Brockman et al. "Openai gym". In: *arXiv preprint arXiv:1606.01540* (2016).

Acknowledgements

Giunto alla conclusione di questo lungo percorso di studi, vorrei porgere i miei più sentiti ringraziamenti, in primo luogo, alle persone che mi hanno accompagnato durante il tirocinio ed infine nella stesura della mia tesi, permettendomi di scoprire questo campo davvero interessante di ricerca, a cui ho avuto modo di appassionarmi, nonostante il breve periodo di tirocinio.

In particolare vorrei ringraziare i dottorandi Giacomo De Bernardi e Sara Narteni, miei correlatori, per la loro disponibilità, per avermi sempre coinvolto nello svolgimento del progetto e per avermi fornito utili consigli, chiarimenti e supporto nella stesura del mio elaborato, nonchè per la fiducia che hanno riposto nei miei confronti.

Rivolgo poi un caloroso e doveroso ringraziamento ai miei relatori: Maurizio Mongelli, per avermi guidato durante questo percorso, facendomi sentire davvero parte integrante del progetto, e Agostino Bruzzone per aver gentilmente approvato la mia tesi e per avermi concesso di entrare a far parte del team del CNR di Genova. Per me è stato un piacere ed una grande soddisfazione.

Colgo inoltre l'occasione per ringraziare anche le persone che mi sono state vicino con affetto durante tutto il mio percorso di studi, soprattutto i miei genitori, per aver sempre creduto in me, mia sorella, la nonne e tutti i miei zii e cugini. Senza di voi il periodo universitario sarebbe stato molto più duro.

Infine, ringrazio la mia compagna, che da ormai un anno si prede cura di me con amore e tanta voglia di cucinare.