Manuscript (Marked Version)

AutoOC: Automated Multi-objective Design of Deep Autoencoders and One-Class Classifiers using Grammatical Evolution

Luís Ferreira[a,*], Paulo Cortez[a]

[a] *ALGORITMI/LASI, Department of Information Systems, University of Minho, Guimaraes, Portugal*

Abstract

One-Class Classification (OCC) corresponds to a subclass of unsupervised Machine Learning (ML) that is valuable when labeled data is non-existent. In this paper, we present AutoOC, a computationally efficient Grammatical Evolution (GE) approach that automatically searches for OCC models. AutoOC assumes a multi-objective optimization, aiming to increase the OCC predictive performance while reducing the ML training time. AutoOC also includes two execution speedup mechanisms, a periodic training sampling, and a multi-core fitness evaluation. In particular, we study two AutoOC variants: a pure Neuroevolution (NE) setup that optimizes two types of deep learning models, namely dense Autoencoder (AE) and Variational Autoencoder (VAE); and a general Automated Machine Learning (AutoML) ALL setup that considers five distinct OCC base learners, specifically Isolation Forest (IF), Local Outlier Factor (LOF), One-Class SVM (OC-SVM), AE and VAE. Several experiments were conducted, using eight public OpenML datasets and two validation scenarios (unsupervised and supervised). The results show that AutoOC requires a reasonable amount of execution time and tends to obtain lightweight OCC models. Moreover, AutoOC provides quality predictive results, outperforming a baseline IF for all analyzed datasets and surpassing the best supervised OpenML human modeling for two datasets.

*Corresponding Author

Email addresses: luis.ferreira@dsi.uminho.pt (Luís Ferreira), pcortez@dsi.uminho.pt (Paulo Cortez)

Keywords: Automated Machine Learning, Deep Autoencoders, Grammatical Evolution, Multi-objective Optimization, One-Class Classification.

1. Introduction

In recent years, NE has gained increasing attention as an interesting approach to optimize Artificial Neural Network (ANN) models [1]. By adopting an Evolutionary Computation (EC) method as the main search engine, NE automates the design of ANNs (e.g., hyperparameters, structure, weights), often finding good solutions in complex and high-dimensional neural modeling spaces while using a reasonable amount of computational resources. Indeed, NE has been successfully applied to a variety of tasks, including [2, 3, 4]: reinforcement learning, unsupervised learning, optimization, time series forecasting, supervised learning, and deep learning Neural Architecture Search (NAS).

With the worldwide growth of Machine Learning (ML) applications, there has been a growing interest in the usage of Automated Machine Learning (AutoML) tools [5]. AutoML alleviates the modeling effort of non-ML experts by automating the search for the best ML algorithm and its hyperparameters. Several recently proposed AutoML tools are based on NE approaches (e.g., [6, 7]). However, the vast majority of AutoML tools target a supervised learning (e.g., classification, regression) and do not handle an OCC.

Also known as unary classification, OCC can be viewed as a subclass of unsupervised learning, where the Machine Learning (ML) model only learns using training examples from a single class [8, 9]. This type of learning is valuable in diverse real-world scenarios where labeled data is non-existent, infeasible, or difficult (e.g., requiring a costly and slow manual class assignment), such as fraud detection [10], cybersecurity [11], predictive maintenance [12] or industrial quality assessment [13].

This work presents a novel application of NE to the field of OCC (as shown in Section 2) and contributes to the growing body of research on the use of GE for optimizing ML models. In particular, we present AutoOC, an AutoML method for OCC that is based on a Grammatical Evolution (GE). GE has been shown to be effective at optimizing the hyperparameters of ML models [14]. AutoOC performs a multi-objective optimization, using the

Non-dominated Sorting Genetic Algorithm II (NSGA-II) algorithm to maximize the predictive performance of the OCC learners while minimizing their training time. The goal is to generate lightweight ML models, an important aspect when working with real-world Big Data that are common in OCC tasks. Furthermore, AutoOC adopts two computationally efficient mechanisms to speed up the overall execution time [15]: a continuous sampling of training data and a parallel fitness evaluation by adopting multi-core processors. Moreover, the adopted grammar allows a flexible definition of which OCC learners are optimized. In this work, we particularly explore two AutoML grammar variants:

- NE - a pure evolutionary Neural Architecture Search (NAS) approach that searches for the best model using two types of deep AEs, standard dense AE and VAE; and
- ALL - a more general AutoML that selects the best of five OCC learners, namely IF, LOF, OC-SVM, AE, and VAE.

Several computational experiments are held to evaluate the effectiveness of the two AutoOC variants, using eight public datasets and two distinct validation modes (unsupervised and supervised). The results are compared with a baseline IF and also with the best public supervised learning results from the OpenML platform [16].

The paper is organized as follows. Section 2 presents the related work. Then, Section 3 describes the problem formulation of the OCC optimization task. Next, Section 4 describes the proposed AutoOC method. Then, Section 5 presents the experimental results, including the datasets used, the experimental setup and the obtained results. Finally, Section 6 presents the main conclusions and discusses future work directions.

2. Related work

The related work can be grouped into three categories: 1 – the application of EC methods to perform an AutoML optimization; 2 – research works that assume a multi-objective AutoML; and 3 studies that specifically target multi-objective OCC. Table 1 summarizes the state-of-the-art works by using these columns: **Year** – the publication year; **Ref.** – the publication reference; **Cat.** – the study category (1, 2 or 3); **BL** – the number of distinct Base Learners (BL) or ML algorithms; **Dat.** – the number of analyzed datasets;

AutoML – if the study performs an AutoML; **NAS** – if the study targets a NAS; **OCC** – if the study performs OCC; **EC** – the type of EC algorithm used to search for the best ML design; and **MO** – if the study considers a Multi-Objective (MO) optimization (more than one objective). The related works are quite recent, with 19 studies published since 2016, including 4 works published in 2021 and 5 in 2022. In this section, the related work analysis is split into two parts. First, we analyze the first 18 rows of Table 1, which are related to research works performed by other authors. Then, we detail the differences between our previous work [12] and this paper, since they share some similarities (as shown by the last two rows of Table 1).

Table 1: Summary of the related work.

Year	Ref.	Cat.	BL	Dat.	AutoML	NAS	OCC	EC	MO
2016	[17]	2	5	1	✓				✓
2017	[18]	1	20	10	✓			GE	
2018	[19]	3	1	1			✓	Genetic Algorithms (GA)	✓
2018	[20]	3	5	4			✓		✓
2019	[21]	1	5	1	✓			GE	
2019	[22]	1	11	10	✓			GA	
2019	[6]	1	1	2	✓	✓		GE	✓
2019	[23]	2	4	2	✓			GA	✓
2020	[24]	1	22	10	✓			GE	
2020	[25]	3	1	1			✓	GA	✓
2021	[26]	1	3	1	✓			GE	
2021	[27]	1	8	50	✓			GE	
2021	[28]	2	1	3	✓				✓
2021	[29]	2	1	2	✓				✓
2022	[30]	1	11	20	✓			GE	
2022	[7]	1	1	1	✓	✓		GE	
2022	[31]	2	1	1	✓				✓
2022	[32]	2	20	-	✓				✓
2022	[12]	1,2,3	3	1	✓	✓	✓	GE	$*$
2023	This Work	1,2,3	5	8	✓	✓	✓	GE	✓

$*$ – only partially studied.

In terms of study categories, from the analyzed first 18 works of Table 1, nine are related to the first category (EC to guide the optimization of the AutoML), six belong to the second category (multi-objective AutoML), and three works are from the third category (multi-objective OCC). From the

first category, most works use EC to perform an hyperparameter tuning of the base learners or to build ML pipelines. Apart from this work, only two other studies apply a NAS optimization, thus approaching a pure NE. All the works from category 1 only target supervised learning algorithms and do not consider an OCC. From category 2, most works consider two optimization objectives, with two exceptions ([17, 28]). Regarding the third category, three works use OCC in a multi-objective manner, with two of them using GA to perform a multi-objective optimization.

In contrast with our research, the majority of the analyzed 18 related works approach supervised learning ML tasks. Only two studies employ an EC to optimize OCC models [19, 25]. Thus, this paper is the only work that assumes a NE to evolve ANNs (performing a NAS). It also optimizes up to five base One-Class (OC) classifiers, while [19] and [25] only optimize one ML algorithm. Moreover, we adopt a GE as the search engine, which is adopted by most of the related works but not by the two OCC optimization studies [19, 25], which use a GA. Finally, the two OCC related works only adopt one dataset, while our work explores eight datasets.

Our previous work [12] was exclusively focused on the predictive maintenance application domain and it only analyzed one dataset. Moreover, the proposed grammar in [12] included just three base learners (IF, OC-SVM and AE), while we also test in this work LOF and VAE. Furthermore, in [12] the experimental results were mostly focused on a single objective. Finally, the proposed AutoOC method is more computationally efficient than our previous work, since it makes use of two acceleration mechanisms (Section 4.1).

3. Problem formulation

In this work, we address the Combined Algorithm Selection and Hyperparameter (CASH) problem for OCC ML tasks. The CASH problem was first proposed in [33] and defines the problem of, given a search space of ML algorithms and its associated hyperparameters, selecting the best algorithm and fine-tuning its hyperparameters by using an optimization method (e.g., Bayesian optimization). The original proposal of the CASH focused on a supervised learning task, in particular classification algorithms. Similarly, most of the recent research works that approach the CASH problem are focused on a supervised learning (as described in Section 2).

Let $\mathcal{D}_{\text{train}} = \{\mathbf{x}_1, ..., \mathbf{x}_n\}$ denote a training dataset with n unlabeled (the normal) examples, where $\mathbf{x}_i$ denotes a vector with several input attribute val-

ues. There is also a disjoint validation set $\mathcal{D}_{\text{valid}}$ with a length of m examples and that can assume two variants: unsupervised, $\mathcal{D}_{\text{valid}_U} = \{\mathbf{x}_{n+1}, ..., \mathbf{x}_{n+m}\}$; or supervised, $\mathcal{D}_{\text{valid}_S} = \{(\mathbf{x}_{n+1}, y_{n+1}), ..., (\mathbf{x}_{n+m}, y_{n+m})\}$, where y_i denotes a binary labeled output class (e.g., $y_i \in \{0, 1\}$). Let $\mathcal{A} = \{A^1, ..., A^k\}$ define a finite set of k OCC algorithms and $\Lambda = \{\Lambda^1, ..., \Lambda^k\}$ the respective hyperparameter search spaces. The CASH search space is defined by $\mathcal{S} = A^i_\lambda$, where A^i_λ denotes the usage of algorithm A^i with the hyperparameter values $\lambda \in \Lambda^i$ and $i \in \{1, ..., k\}$. A particular A^i_λ OCC algorithm is trained using the unlabeled training examples, namely the $\mathcal{D}_{\text{train}}$ dataset, generating the learning model $\mathcal{M}^i_\lambda$.

In this work, we assume multi-objective OCC CASH task, where an $\mathcal{O}$ optimization algorithm searches for the best A^*_λ combination that satisfies:

$$A^*_\lambda \in \underset{A^j \in \mathcal{A}, \lambda \in \Lambda^j}{\arg\min} (\mathcal{L}_1(A^i_\lambda, \mathcal{D}_{\text{valid}}), \mathcal{L}_2(A^i_\lambda, \mathcal{D}_{\text{train}})) \tag{1}$$

where $\mathcal{L}_1$ denotes a generalization error measured using the validation set and $\mathcal{L}_2$ represents the computational effort required to train the learning model ($\mathcal{M}^i_\lambda$).

4. Proposed method: AutoOC

In this paper, we propose the AutoOC method to solve the multi-objective CASH problem for OCC ML tasks. The algorithm search space $\mathcal{A}$ is composed of a maximum of five ($k = 5$) OCC learners, namely A = {IF,LOF,OC-SVM,AE,VAE} (see Section 4.3). The search for the best OCC algorithm and hyperparameters ($\mathcal{O}$) is performed by a computationally efficient multi-objective that uses a GE and the NSGA-II algorithm that returns a set of best Pareto solutions $\mathcal{B} = \{A^1_\lambda, ..., A^p_\lambda\}$, where each A^i_λ combination is a non-dominated solution in terms of the $\mathcal{L}_1$ and $\mathcal{L}_2$ minimization objectives. As for the hyperparameter search spaces (Λ), they are defined by the adopted GE grammar (as detailed in Section 4.4).

GE is a biologically inspired evolutionary algorithm for generating computer programs. The algorithm was proposed by O'Neill and Ryan in 2001 [34] and has been widely used in both optimization and ML tasks. GE can handle complex optimization problems with a large number of objectives and constraints. It can also handle continuous and discrete optimization problems, as well as problems with mixed variables. Indeed, GE has been shown to be effective in finding high-quality solutions in a relatively short time,

compared to other optimization methods [35]. In GE, a set of programs is represented as strings of characters, known as chromosomes. The chromosomes are encoded using a formal grammar, which defines the syntax and structure of the programs. The grammar is used to parse the chromosomes and generate the corresponding programs, which are then evaluated using a fitness function. The fitness function measures the quality of the programs and is used to guide the evolution process toward better solutions.

There are two main reasons that make GE a suitable choice for our AutoML OCC search. Firstly, it can handle variable-length solution representations, which is useful when handling different types of OCC algorithms, where each algorithm contains its own hyperparameters. Secondly, and in contrast with other variable-length EC methods, such as Genetic Programming or Gene Expression Programming, it allows an easy customization of the OCC search space, since it is defined by a human-readable grammar. The AutoOC grammar employs up to $k = 5$ OCC methods and directly generates Python code. If needed, the grammar can be adapted to include any combination of the five base learners, additional hyperparameters, or even new OCC algorithms.

AutoOC assumes a multi-objective optimization by adopting the popular NSGA-II algorithm that was proposed in 2002 [36]. The algorithm is based on the concept of non-dominance, which means that a solution is considered superior to another solution if it is not worse than the other solution in any objective and strictly better in at least one objective. The goal of NSGA-II is to find a set of non-dominated solutions, known as the Pareto front, which represents the trade-off between the different objectives. NSGA-II includes a crowding distance measure, which is used to preserve diversity among the solutions and avoid premature convergence. The algorithm has been widely used in various fields, including engineering, economics, and biology, and has shown promising results in a variety of multi-objective optimization problems [37].

In this study, we implemented a Pareto optimization approach to simultaneously minimize two objectives: generalization discrimination error ($\mathcal{L}_1$) and training time ($\mathcal{L}_2$). The resulting Pareto front contains a set of non-dominated solutions, each representing a trade-off between the two objectives. The rationale of this multi-objective approach is to allow for the selection of lightweight OCC models, even if they are associated with a slightly lower performance. Indeed, reducing the computational training time is particularly valuable within the OCC domain, since most of the analyzed

datasets are unlabeled and thus often rather large.

4.1. Acceleration mechanisms and objective functions

As explained in Section 3, the training data $\mathcal{D}_{\text{train}}$ is composed only of data from one class ("normal" data). OCC typically involves a large set of unlabeled data, thus performing an evolutionary optimization in this domain is a computationally demanding task. In order to speed up the GE execution time, AutoOC adopts two recently proposed computationally efficient mechanisms [15].

Firstly, AutoOC uses a periodic sampling mechanism, where each g generation of the GE optimization uses the random sample $\mathcal{D}^g_{train}$ that includes $s < n$ examples from the entire training dataset. For an example dataset with n =10,000 records and a sample size of s=2,500, each generation of the GE optimization will use s=2,500 randomly sampled records to train the OCC models. The sampling is applied to the entire dataset at the beginning of each generation and it is performed with replacement (similarly to the bagging ML ensemble method), meaning that a specific record can be chosen more than once. The reason for this approach is related to an acceleration of the total optimization time, since training the models on a small sample of a dataset will be faster than training all the individuals on an entire dataset, especially if the dataset has a huge number of records (e.g., millions of records). On the other hand, the fact that each generation uses a different set of examples will allow the optimization to avoid overfitting the training set since the training data is always different. Secondly, each A^i_λ solution is trained in a parallel manner, where a $\mathcal{M}^i_\lambda$ model is obtained by applying the A^i_λ algorithm to the $\mathcal{D}^g_{train}$ dataset. This means that, for each generation, more than one individual can be trained at the same time using different cores (processors). In practice, when the used machine has more cores than the population size, it is possible to train all the individuals at the same time. For each trained $\mathcal{M}^i_\lambda$ model, AutoOC stores the value of $\mathcal{L}_2(A^i_\lambda, \mathcal{D}^g_{train})$, which corresponds to the time elapsed to obtain $\mathcal{M}^i_\lambda$ when using a single core, in seconds. This $\mathcal{L}_2$ value corresponds to the second objective function, which guides the $\mathcal{O}$ search in terms of minimizing the OCC training computational effort.

AutoOC is primarily designed for anomaly detection tasks, where most examples are "normal" records. While the training only uses normal examples, the OCC predictive performance validation can be performed using two distinct setups [12]: unsupervised validation, where the model performance is

evaluated using only $\mathcal{D}_{\text{valid}_U}$ unlabeled data (e.g., through an anomaly score), or supervised validation, where there is access to a (often smaller) $\mathcal{D}_{\text{valid}_S}$ labeled validation set to assess the model performance by using supervised learning metrics, such as the popular Area Under the Curve (AUC) of the Receiver Operating Characteristic (ROC) curve classification measure [38].

All OCC models produce an anomaly score (S_j) for a particular $\mathbf{x}_j$ data example. The $\mathcal{M}_\lambda^i$ validation or test anomaly scores are first normalized within the $S_i \in [0, 1]$ range by applying a min-max normalization using the training data. Two relevant performance measures adopted in this work are the average anomaly score ($\overline{S}$) and AUC:

$$\begin{array}{rcl} \overline{S} & = & \frac{1}{l}\sum_{j=1}^{l} S_j \\ AUC & = & \int_0^1 ROC dTh \end{array} \tag{2}$$

where l denotes the length of the predicted data (e.g., $l = m$ for a validation set) and $Th \in [0, 1]$ is a threshold decision value, allowing to interpret the predicted anomaly class as positive if $S_i > Th$. The ROC curve plots the False Positive Rate (FPR) versus the True Positive Rate (TPR) for all threshold values. AUC is a popular binary classification measure of performance, providing two main advantages [39]. Firstly, quality values are not influenced by the presence of unbalanced data, which occurs in OCC tasks. Secondly, the AUC values can be easily interpreted as follows: 50% – performance of a random classifier; 60% - reasonable; 70% - good; 80% - very good; 90% - excellent; and 100% - perfect.

In this work, we explore the two OCC validation modes (supervised and unsupervised), which lead to two distinct fitness functions that measure OCC generalization error performance ($\mathcal{L}_1$, the first objective function). For the supervised validation, the generalization error performance (to be minimized), is defined as $\mathcal{L}_1(A_\lambda^i, \mathcal{D}_{\text{valid}_S}) = 1-\text{AUC}$. The lower the $\mathcal{L}_1$ value, the better will be the OCC AUC predictive performance. Under the unsupervised validation mode, labeled data (i.e., abnormal examples) is not available, making the computation of the AUC infeasible. Therefore, to select the best ML models, the average anomaly score ($\overline{S}$) is used as a proxy for the 1-AUC computation: $\mathcal{L}_1(A_\lambda^i, \mathcal{D}_{\text{valid}_U}) = \overline{S}$. The idea is that if a model produces a low anomaly score when trained on a large set of normal data, it should be capable of generating high anomaly scores for abnormal data, resulting in a satisfactory ROC curve. Nevertheless, it is important to note that to accurately benchmark the unsupervised validation scenario, labeled data was used

in the test set, allowing the computation of ROC curves and AUC measures, which were then compared to those obtained using the supervised validation scenario. Table 2 summarizes the type of data used for each validation setup.

Table 2: Validation modes for AutoOC.

Validation Mode	**Training Set**	**Validation Set**	**Test Set**
Supervised	Unlabeled Data	Labeled Data	Labeled Data
Unsupervised	Unlabeled Data	Unlabeled Data	Labeled Data

4.2. Pseudo-code

The pseudo-code for our proposed AutoOC is illustrated in Algorithm 1. There are four main inputs, the training and validation sets ($\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{valid}}$, the sampling size ($s$), the maximum number of generations (G) and the population size (N_P). The search algorithm ($\mathcal{O}$) combines a GE with a multi-objective NSGA-II optimization. The GE elements are used to generate the initial population and breed new individuals (through crossover and mutation operators). As for the NSGA-II procedures, they enforce a simultaneous multi-objective search in terms of selecting interesting new population individuals and the best set of Pareto solutions. Moreover, the two AutoOC acceleration mechanisms are implemented in lines 6 (periodic random sampling) and 7 (parallel execution of the training algorithm and computation of its validation measures). After G generations (the termination criteria), the search returns the best searched Pareto front of solutions ($\mathcal{B}$).

4.3. Base learners

AutoOC uses up to five popular OCC Learning algorithms: AEs, IF, LOF, OC-SVM, and VAEs. The AEs and VAEs were implemented through the Keras module of TensorFlow library [40], while IF, LOF, and OC-SVM used the Scikit-Learn framework [41]. Table 3 summarizes the five adopted base learners in terms of: the name of the (**Algorithm**), the base (**Framework**), used (**Version**), and (**API**) documentation reference.

LOF is a density-based anomaly detection algorithm that is used to identify instances in a dataset that are significantly different from the majority of the instances. It works by calculating an anomaly score S_i for each instance i, which reflects the degree to which it is isolated from the rest of the

Algorithm 1 AutoOC pseudo-code.

Inputs: $\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{valid}}, \mathcal{S}, s, G, N_P$ ▷ Training and validation sets, search space, training sample size, maximum number of generations and population size
$\mathcal{B} \leftarrow \emptyset$ ▷ Initialize $\mathcal{B}$
$\mathcal{P}^0 \leftarrow create(\mathcal{S}, N_P)$ ▷ Initial GE population with $\{A_\lambda^1, ..., A_\lambda^{N_P}\}$ solutions
$g \leftarrow 0$
while $g < G$ **do** ▷ Cycle up to G generations
  $\mathcal{D}_{\text{train}}^g \leftarrow sample(\mathcal{D}_{\text{train}}, s)$ ▷ Random sample of size s
  $F^g \leftarrow evaluate(\mathcal{P}^g, \mathcal{D}_{\text{train}}^g, \mathcal{D}_{\text{valid}})$ ▷ Fitness values ($\mathcal{L}_1$ and $\mathcal{L}_2$) for $\mathcal{P}^g$
  $\mathcal{P}^g \leftarrow \mathcal{P}^g \cup \mathcal{B}$ ▷ Add $\mathcal{B}$ to current population
  $\mathcal{B} \leftarrow \emptyset$ ▷ Reinitialize $\mathcal{B}$
  if $A_\lambda^i \in \mathcal{P}^g$ is a NSGA-II interesting (e.g., non-dominated) solution **then** ▷ Apply NSGA-II
    $\mathcal{B} \leftarrow \mathcal{B} \cup A_\lambda^i$ ▷ Add A_λ^i to the set of best Pareto solutions
  end if
  $\mathcal{P}^{g+1} \leftarrow$ evolve$(\mathcal{P}^g)$ ▷ Apply GE crossover and mutation operators and NSGA-II selection
  $g \leftarrow g + 1$ ▷ Increment g
end while

return $\mathcal{B}$ ▷ Return best solutions (Pareto front of OCC models)

Table 3: Characteristics of the base learners used by AutoOC.

Algorithm	**Framework**	**Version**	**API**
Local Outlier Factor (LOF)	Scikit-Learn	1.2.0	[42]
Isolation Forest (IF)	Scikit-Learn	1.2.0	[43]
One-Class SVM (OC-SVM)	Scikit-Learn	1.2.0	[44]
Autoencoder (AE)	TensorFlow	2.6.0	[45]
Variational Autoencoder (VAE)	TensorFlow	2.6.0	[46]

examples in the dataset. LOF is particularly useful for detecting anomalies in high-dimensional datasets, as it is able to capture complex patterns in the data [47]. High LOF scores are considered to be outliers, as they are located in areas of the feature space that are less densely populated. Thus, in this work, we use the LOF S_i score as the anomaly degree measure.

IF is an OCC algorithm that is used for detecting anomalous data points in a dataset [48]. IF is particularly useful for identifying outliers in large, high-dimensional datasets. The algorithm works by creating a forest of decision trees, where each tree is trained to isolate a single instance in the dataset. IF is based on the idea that anomalous data points are more difficult to isolate and will therefore have shorter paths in the decision tree. The algorithm calculates an anomaly score for each data point (S_i), which is based on the length of the path to the isolated data point in the decision tree. Data points with higher S_i values are more likely to be anomalous and thus this measure is used as the anomaly score.

OC-SVM is a ML algorithm that is used to identify anomalies in a dataset. It is an extension of the Support Vector Machine algorithm that is designed to work with unlabeled data [49]. OC-SVM is particularly useful for detecting rare or unusual events and is often used in fraud detection, intrusion detection, and other applications where the goal is to identify instances that are significantly different from the norm. OC-SVM works by finding a hyperplane in the feature space that maximally separates the normal instances from the origin, and then classify any new instances as normal or anomalous based on which side of the hyperplane they fall on. In this work, the anomalous class probability of OC-SVM is directly used as the anomaly score (S_i).

AEs are a type of ANN that are trained to reconstruct their input data by learning a compressed representation (or encoding) of the input data and then using this encoding to reconstruct the original data. AEs can be used in a variety of tasks, including dimensionality reduction, anomaly detection, and data generation [50]. Following the success of Deep Learning, there has been a growing usage of AEs to perform OCC [39]. Within this context, AEs can be trained on normal data and attempt to produce outputs that are similar to the inputs. For each input instance, there is a reconstruction error, and higher reconstruction errors are associated with a higher probability of being an anomaly [51]. In this work, the popular Mean Absolute Error (MAE) measure [52] is used to compute the reconstruction error for an instance i, which corresponds to the adopted anomaly score (S_i).

VAEs differ from traditional AEs in that they are trained to learn a distribution over the input data, rather than simply reconstructing the input data [53]. VAEs are composed of two parts: an encoder that maps the input data to a latent representation, and a decoder that maps the latent representation back to the original data space. The encoder and decoder are trained to optimize an objective function that encourages the generated data to be similar to the original data, while also encouraging the latent representation to be smooth and continuous. This allows VAEs to generate new data points that are similar to the original data, whereas traditional AEs are only able to reconstruct the input data. VAEs can be used for anomaly detection since anomalies are expected to have different distributions when compared with the normal training examples [54]. To compute the anomaly score (S_i) for an instance i, the same MAE measure is computed by comparing the VAE prediction with the input data.

4.4. AutoOC grammar

GE uses a mapping process to generate programs from a genome encoded using a formal grammar, typically in Backus-Naur Form (BNF) notation. This notation consists of terminals, which represent items that can appear in the language, and non-terminals, which are variables that include one or more terminals. In this study, we used an open-source implementation of GE in Python (PonyGE2 [55]) to develop AutoOC. PonyGE2 allows for the use of Python BNF (PyBNF), which enables the inclusion of Python code in the production rules. To build AutoOC, a PyBNF grammar was developed to tune the hyperparameters of the OCC algorithms described in Section 4.3.

The use of PyBNF allowed for the generation of Python code snippets that enabled GE to produce various types of ML models. For example, the IF, LOF, and OC-SVM grammars were implemented by creating the corresponding Scikit-Learn class and adding the hyperparameters as terminals and non-terminals. We note that the proposed grammar includes all IF, LOF, and OC-SVM hyperparameters that were available in the consulted Scikit-Learn documentation (see Table 3). The process was different for the AEs and VAEs, as the TensorFlow API requires the definition of a variable number of layers. To address this, the grammar was designed to generate only the encoder: first, an input layer with the same number of nodes as the number of attributes in the dataset is generated, followed by a variable number of hidden layers. Given that in a typical AE or VAE the subsequent encoder layers have fewer nodes than the previous layer, the layer nodes were

defined as a percentage (between 0% and 100%) of nodes in the previous layer rather than a fixed number. Two auxiliary functions, (`get_ae_from_encoder` and `get_vae_from_encoder`), were also defined to translate the generated phenotype into functional TensorFlow AEs and VAEs. The decoder, which is symmetrical to the encoder, was not included in the grammar. Besides the ANN structure, deep learning architectures include a large number of additional hyperparameters. In order to reduce the search space, using modeling knowledge from previous OCC works (e.g., [39, 13]) we fixed some choices, such as the usage of the MAE measure as the loss function for both AE and VAE and usage of Batch Normalization layers for AE. We also restricted the search space for some hyperparameters. For instance, only two optimizers are explored to adjust the AE weights (RMSprop and Adam). Moreover, while the analyzed TensorFlow version provides up to 16 activation functions, the proposed grammar only searches for the best of eight of these functions (e.g., ReLU). Nevertheless, in future works and if needed, the grammar can be easily adapted to include other deep learning hyperparameter choices.

The proposed grammar of AutoOC defines the OCC search space ($\mathcal{S}$) and is flexible enough to allow the usage of any combination of the five base learners (Section 4.3) or even include other OCC algorithms. In this work, we empirically study the effect of two AutoOC variants: NE – assuming only the deep AE and VAE base learners (thus $k = 2$), working as a pure NAS optimization; ALL – where all $k = 5$ base learners are used during the optimization, working as a more general AutoML OCC search. The developed PyBNF grammar for the "ALL" mode is shown in Fig. 1. The grammar for the "NE" mode follows a similar logic, using only the AEs and VAEs entries.

5. Experimental results

5.1. Datasets

A total of eight public domain datasets (Table 4) were retrieved from OpenML [16], an open platform for sharing datasets and ML experiments. As selection criteria, we opted to select binary classification tasks from distinct application domains (e.g., banking, telecommunications) and reflecting different numbers of instances (**Rows**), categorical (**Categorical Columns**) and numerical attributes (**Numerical Columns**), and output target class balancing (**Class Balancing**). We particularly selected datasets with a clear

```
<response> ::= <autoencoder> | <iforest> | <lof> | <ocsvm> | <vae>

<autoencoder> ::=  encoder = Sequential(){::}
                   encoder.add(Input(shape=(input_shape,), name="'input'")){::}
                   <hidden_layers>{::}
                   <latent_space>{::}
                   model = get_ae_from_encoder(encoder){::}
                   model.add(Dense(input_shape, activation=<activation>, name="'output'")){::}
                   model.compile(<optimizer>, "'mae'")
<hidden_layers> ::= <Dense>{::} | <Dense>{::}<Dense>{::} | <hidden_layers><Dense>{::} | <Dense>{::}<extra>{::}
<Dense> ::= encoder.add(Dense(units = <percentage>, activation = <activation>))
<activation> ::= "'relu'" | "'sigmoid'" | "'softmax'" | "'softplus'" | "'tanh'" | "'selu'" | "'elu'" | "'exponential'"
<latent_space> ::= encoder.add(Dense(units = <percentage>, activation = <activation>, name="'latent'"))
<extra> ::= encoder.add(Dropout(rate=0.<dropout_digit>)){::} | encoder.add(BatchNormalization()){::}
<dropout_digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<percentage> ::= 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100
<optimizer> ::= "'RMSprop'" | "'Adam'"


<iforest> ::= model = IsolationForest(n_estimators=<estimators>, contamination=<contamination>, bootstrap=<bootstrap>, n_jobs =-1)
<estimators> ::= <digit><estimators> | <digit>
<estimators_digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<contamination> ::= "'auto'" | 0.<contamination_digits>
<contamination_digits> ::= 1 | 2 | 3 | 4 | 5
<bootstrap> ::= "True" | "False"


<ocsvm> ::= model = OneClassSVM(kernel=<kernel>, degree=<degree>, gamma=<gamma>, shrinking=<shrinking>)
<kernel> ::= "'linear'" | "'poly'" | "'rbf'" | "'sigmoid'"
<degree> ::= <digit><degree_digit> | <digit>
<degree_digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<gamma> ::= "'scale'" | "'auto'"
<shrinking> ::= "True" | "False"


<lof> ::= model = LocalOutlierFactor(n_neighbors = <n_neighbors>, algorithm = <algorithm>, leaf_size = <leaf_size>,
                   metric = <metric>, contamination = <contamination>, novelty = "True", n_jobs =-1)
n_neighbors> ::= <single_digit> | <single_digit><digit> | <single_digit><digit><digit>
<algorithm> ::= "'ball_tree'" | "'kd_tree'" | "'brute'" | "'auto'"
<leaf_size> ::= <single_digit> | <single_digit><digit>
<metric> ::= "'minkowski'" | "'cityblock'" | "'chebyshev'" | "'euclidean'" | "'l1'" | "'l2'"


<vae> ::=     encoder = Sequential(){::}
                   encoder.add(Input(shape=(input_shape,), name="'input'")){::}
                   <hidden_layers>{::}
                   <z_mean>{::}
                   <z_log_var>{::}
                   <z>{::}
                   model = get_vae_from_encoder(encoder){::}
                   model.add(Dense(input_shape, activation=<activation>, name="'output'")){::}
                   model.compile(<optimizer>, "'mae'")
<z_mean> ::= encoder.add(Dense(units = <percentage>, activation = <activation>, name="'z_mean'"))
<z_log_var> ::= encoder.add(Dense(units = <percentage>, activation = <activation>, name="'z_log_var'"))
<z> ::= Lambda(sample, output_shape=(<percentage>,), name='z')
```

Figure 1: The adopted PyBNF grammar (for the full "ALL" search space representation mode).

distinction between the two classes, where the majority class could be considered as "normal" and the minority class as an "anomaly" state. Table 4 also details the name of the dateset (**Dataset**) and the unique OpenML identifier (**OpenML ID**).

Table 4: Description of the selected OpenML datasets.

Dataset	**OpenML ID***	**Rows**	**Categorical Columns**	**Numerical Columns**	**Class Balancing ("normal"/"anomaly")**
Bank Marketing	1461	45,211	16	9	88%/12%
Churn	40701	5,000	4	16	86%/14%
Credit Card	44235	284,807	0	30	99%/1%
EEG	1471	14,980	0	14	55%/45%
Mushroom	24	8,124	23	0	52%/48%
Nomao	1486	34,465	30	89	71%/29%
Phoneme	1489	5,404	0	5	70%/30%
Spambase	44	4,601	0	57	60%/40%

*The datasets can be retrieved by entering their OpenML unique identifier (ID) at the following URL: `http://www.openml.org/search?type=data&id=`ID.

Since AutoOC focuses on algorithm selection and hyperparameters, the datasets need to be preprocessed before feeding them into the OCC base learners. In order to achieve a fair comparison, the same fixed data preprocessing is applied to all datasets.

Since none of the base learners deals with data attributes of type String, we encoded all String attributes into numerical types. For categorical attributes with low cardinality (ten levels or fewer), we applied the popular one-hot encoding. For categorical columns with missing values, we replaced the missing values with zero, which is treated as a numeric code value for the "unknown" level. As for high cardinality categorical attributes, the one-hot transform produces a large number of binary inputs, which highly affects the computational performance (in terms of both memory and processing time). Thus, for these attributes, we employed instead the Inverse Document Frequency (IDF) technique, available in the Python CANE module [56], which converts a categorical column into a numerical column of positive values based on the frequency of each attribute level. IDF uses the function $f(x) = log(n/fx)$, where n is the length of x and $f(x)$ is the frequency of x. This technique has the advantage of generating just one numeric column for each attribute, thus reducing the ML computational effort. For the remaining

attributes of Integer and Float types, we applied a z-score standardization [52], which results in a new scale with a mean of zero and standard deviation of one. The missing values in numerical columns were also replaced with the mean value for that column (mean imputation).

5.2. Experimental setup

All experiments were run on an Intel Xeon 1.70 GHz server with 56 cores and 64 GB of RAM, without a GPU. When running AutoOC, we stored two types of time elapsed times (in seconds), the overall GE execution time and the training time required by the OCC algorithms. To assess the performance of AutoOC, we followed an approach based on the benchmark in [5]. We divided the datasets into 10 folds to obtain an external cross-validation, which is used to get test (unseen) data that allows measuring the predictive generalization performance of the selected OCC model. As for the training data, it is further split by applying an internal and random holdout split, where 75% of the data is used for fitting purposes ($\mathcal{D}^{g}_{\text{train}}$) and the remaining 25% is used for validation purposes ($\mathcal{D}_{\text{valid}}$).

To evaluate the predictions on the test set from the external 10-fold validation, we employed the AUC analysis of the ROC curve. The obtained results are aggregated by computing the median of the evaluation measures across the 10 external folds and their respective 95% confidence intervals based on the nonparametric Wilcoxon test [57], to determine the statistical significance of the experiments.

5.3. AutoOC results

For each dataset, we executed four AutoOC experiments, with two base learner configurations (NE and ALL) and both validation modes (supervised and unsupervised). Since it is unfeasible to evaluate every possible combination of the GE optimization parameters, we fixed some of these values using reasonable assumptions and some preliminary experiments performed using other OCC datasets. The summary of the different parameters used in the experimental evaluation is shown in Table 5. All experiments were executed with an initial random generated population of N_P =20 individuals and G =100 generations. Also, for the GE parameters of crossover and mutation, we adopted the default PonyGE2 values: Variable One-point crossover (selection of a different point on each parent genome for crossover to occur) with a crossover probability of 75%; and Int Flip Per Codon mutation (random mutation of every individual codon in the genome) with a mutation

Table 5: GE parameters used for the experiments.

Parameter	Used Values
Population Size (N_P)	20
Number of Generations (G)	100
Crossover	Variable One-point with 75% crossover probability (PonyGE2 default)
Mutation	Int Flip Per Codon with 100% mutation probability (PonyGE2 default)
Base Learners Setup	ALL (used algorithms: AE, IF, LOF, OC-SVM, VAE) NE (used algorithms: AE, VAE)
Optimization Type	Multi-objective (NSGA-II)
Predictive Objective	Minimize validation 1-AUC ($\mathcal{L}_1$ for supervised validation) Minimize validation average anomaly score $\overline{S}$ ($\mathcal{L}_1$ for unsupervised validation)
Efficiency Objective	Minimize training time ($\mathcal{L}_2$)
Validation Type	Supervised Unsupervised
Sampling size	s=2,500
Parallel Training	True

probability of 100%. Additionally, we applied both acceleration mechanisms described in Section 4.1, using a periodic random sampling, performed in each generation and applied to all the population individuals, of s=2,500 records and parallel training.

Table 6 presents the results obtained by AutoOC on the eight open-source datasets described in Section 5.1. The table shows the median test set results of the external 10 folds and the respective confidence intervals for the predictions (**Median AUC**) and the efficiency (**Median Training Time**, in seconds). It is worth noting that, since these experiments apply a multi-objective approach, each external fold generates more than one optimal model per fold (all that belong to the Pareto front). Thus, we divided the AUC and training time median results into three columns each. The predictions (**Pred.**) column only considers the individuals from the Pareto front with the best predictive objective score; the **Speed** column considers the Pareto front individuals with the least training time (efficiency objective, in seconds); the column **Pareto** considers all the individuals belonging to the Pareto front. Table 6 also shows the median time needed for the GE optimization (**Median GE Time**) with confidence intervals, the type of validation (**V**) that was used, and which base learner setup was considered (**BL**). For the best results of each dataset (AUC, training time, and GE time; values highlighted using a **boldface** font), we apply the nonparametric Wilcoxon test for measuring statistical significance.

Regarding the predictive performance, the ALL mode with supervised validation achieved the best median AUC on the test set for: seven of the

Table 6: AutoOC experimental results (best values for each measure in **bold**).

Dataset	BL	V*	Median AUC			Median Training Time			Median GE Time
			Pred.	Speed	Pareto	Pred.	Speed	Pareto	
Bank	ALL	S	**0.72**[a]±0.01	0.52±0.02	**0.63**[a]±0.01	**0.29**[c]±0.20	**0.01**[c]±0.00	**0.05**[a]±0.03	**740**[c]±57
Bank	ALL	U	0.62±0.03	0.55±0.04	0.58±0.01	0.41±0.02	0.01±0.00	0.11±0.03	758±34
Bank	NE	S	0.62±0.01	0.56±0.00	0.59±0.00	8.48±1.59	1.55±0.09	3.98±0.74	2,002±102
Bank	NE	U	0.65±0.02	**0.58**[b]±0.03	0.60±0.02	9.46±0.86	3.27±0.65	6.14±0.43	1,505±44
Churn	ALL	S	**0.75**[a]±0.01	**0.56**[b]±0.02	**0.65**[a]±0.01	**0.01**[a]±0.02	**0.01**[c]±0.00	**0.01**[a]±0.00	691±81
Churn	ALL	U	0.62±0.03	0.55±0.01	0.59±0.01	0.27±0.02	0.01±0.00	0.08±0.01	**645**[c]±38
Churn	NE	S	0.63±0.01	0.55±0.00	0.60±0.01	7.13±1.24	1.33±0.10	3.47±0.25	1,657±29
Churn	NE	U	0.53±0.01	0.52±0.02	0.53±0.00	7.50±0.82	2.12±0.39	3.92±0.22	1,389±24
Credit	ALL	S	0.92±0.00	0.80±0.08	0.88±0.01	**0.13**[a]±0.04	**0.01**[c]±0.00	**0.04**[a]±0.02	**949**[c]±125
Credit	ALL	U	0.97±0.09	0.84±0.01	0.89±0.04	0.48±0.04	0.01±0.00	0.15±0.02	1,191±448
Credit	NE	S	**0.98**[e]±0.00	**0.93**[a]±0.00	**0.95**[a]±0.00	7.25±0.81	1.19±0.08	3.56±0.54	2,211±202
Credit	NE	U	0.91±0.10	0.89±0.01	0.90±0.02	10.79±1.28	2.54±0.58	5.31±0.47	4,208±1430
EEG	ALL	S	**0.68**[a]±0.03	0.51±0.01	**0.59**[a]±0.02	**0.22**[c]±0.43	0.01±0.00	**0.05**[c]±0.15	**617**[c]±230
EEG	ALL	U	0.56±0.01	**0.52**[b]±0.02	0.53±0.03	0.27±0.12	**0.01**[c]±0.00	0.06±0.02	645±41
EEG	NE	S	0.52±0.02	0.49±0.02	0.51±0.01	5.53±2.45	1.36±0.12	2.82±0.81	3,798±965
EEG	NE	U	0.51±0.00	0.51±0.00	0.51±0.01	7.99±1.32	1.88±0.13	3.95±0.21	1,985±56
Mushroom	ALL	S	**1.00**[d]±0.00	0.62±0.13	0.81±0.05	0.27±0.06	**0.01**[c]±0.00	0.06±0.02	1,057±113
Mushroom	ALL	U	0.58±0.06	0.50±0.10	0.58±0.02	**0.24**[c]±0.04	0.01±0.00	**0.03**[a]±0.01	**998**[c]±90
Mushroom	NE	S	0.99±0.04	0.82±0.20	0.87±0.02	9.22±1.46	2.08±0.15	3.99±0.23	2,016±1508
Mushroom	NE	U	0.99±0.04	**0.83**[f]±0.02	**0.91**[a]±0.02	9.46±1.35	2.62±0.19	5.31±0.49	2,041±31
Nomao	ALL	S	**0.83**[a]±0.02	0.62±0.06	**0.73**[a]±0.02	**0.01**[a]±0.00	**0.01**[c]±0.00	**0.01**[a]±0.00	**885**[c]±115
Nomao	ALL	U	0.63±0.05	0.59±0.02	0.62±0.09	0.46±0.09	0.01±0.00	0.07±0.02	905±58
Nomao	NE	S	0.70±0.01	0.50±0.01	0.63±0.01	5.62±1.58	1.89±0.14	3.08±0.63	1,901±67
Nomao	NE	U	0.68±0.02	**0.67**[a]±0.02	0.68±0.03	7.48±0.48	3.07±0.37	5.29±0.27	2,370±266
Phoneme	ALL	S	**0.74**[a]±0.01	**0.57**[b]±0.06	**0.68**[a]±0.01	**0.01**[a]±0.01	**0.01**[c]±0.00	**0.01**[a]±0.00	654±84
Phoneme	ALL	U	0.63±0.02	0.53±0.03	0.60±0.02	0.22±0.05	0.01±0.00	0.07±0.02	**634**[c]±81
Phoneme	NE	S	0.62±0.01	0.52±0.01	0.58±0.01	13.98±5.47	1.27±0.02	4.24±0.78	2212±88
Phoneme	NE	U	0.56±0.01	0.51±0.02	0.53±0.02	6.73±1.63	1.38±0.11	3.26±0.39	2192±134
Spambase	ALL	S	**0.81**[a]±0.01	**0.62**[b]±0.07	**0.73**[a]±0.01	**0.12**[a]±0.13	0.01±0.00	**0.03**[a]±0.03	**608**[c]±111
Spambase	ALL	U	0.68±0.04	0.59±0.00	0.63±0.04	0.29±0.02	**0.01**[c]±0.00	0.07±0.01	618±51
Spambase	NE	S	0.62±0.01	0.50±0.01	0.57±0.00	9.76±0.92	1.49±0.06	4.47±0.28	1,979±40
Spambase	NE	U	0.73±0.01	0.60±0.04	0.65±0.01	13.15±1.43	3.29±0.57	7.28±0.55	2,009±51

* Validation mode: S - Supervised; U - Unsupervised.
[a]Statistically significant (p-value < 0.05) under a pairwise comparison when compared with all the other setups.
[b]Statistically significant (p-value < 0.05) under a pairwise comparison when compared with none of the other setups.
[c]Statistically significant (p-value < 0.05) under a pairwise comparison when compared with the setups: Supervised NE and Unsupervised NE.
[d]Statistically significant (p-value < 0.05) under a pairwise comparison when compared with the setups: Unsupervised ALL.
[e]Statistically significant (p-value < 0.05) under a pairwise comparison when compared with the setups: Supervised ALL.
[f]Statistically significant (p-value < 0.05) under a pairwise comparison when compared with the setups: Supervised ALL and Unsupervised ALL.

eight datasets when considering predictive power; three datasets when considering the training speed; and six datasets when considering the entire Pareto front. In these scenarios, the supervised ALL achieved a median of 11.0 AUC percentage points (pp) higher than the respective second-best configuration for predictive mode, 2.0 pp for speed mode, and 5.5 pp for the Pareto mode. An interesting result was obtained by the Credit Card dataset, achieving the best predictive results exclusively with NE approaches, namely with the supervised validation mode. This setup obtained, on median, 1.0 AUC pp higher than the second-best setup for predictive mode, 9.0 pp for speed mode, and 6.0 pp for the Pareto mode.

When considering the total GE execution time, the unsupervised ALL approach required a median value of 702 s across all datasets, followed by supervised ALL (716 s), supervised NE (2,009 s), and unsupervised NE (2,025 s). These results can be explained by the training time required by the deep ANNs (either a traditional AE or a VAE), which is higher when compared with the other base learners. In effect, both ALL setups (supervised and unsupervised) tend to produce lightweight OCC models, presenting median training time values always lower than one second, and most of the times being only 0.01 s. In contrast, the setups with the NE variant present median training times ranging from 1.19 s and 13.98 s, with a median value of 3.99 s. Nevertheless, the total GE execution time results back the proposed AutoOC as a computationally efficient tool to model large OCC datasets. For instance, for the largest dataset (Credit Card, with around 285,000 examples), and when adopting the supervised validation mode, the ALL and NE variants only require a median GE optimization time of 949 s (around 16 minutes) and 2,211 s (around 37 minutes). In Section 5.4, we further compare these Credit execution time results (using the sampling mechanism) with a GE that uses all training data (no sampling).

To further compare the obtained AutoOC results, we analyzed the Pareto fronts from the test set results. Given that each experiment is composed of ten test sets (one for each external fold), we aggregate the distinct Pareto fronts from each experiment. Inspired by the ROC curve vertical aggregation [38], we aggregate the results vertically. To facilitate the visual analysis, in all Pareto front graphs shown in this paper, we assume the -AUC minimization objective on the x-axis and the training time minimization objective on the y-axis. Thus, the ideal point corresponds to the bottom left corner of the Pareto graphs. For different values of -AUC, we estimate the Wilcoxon median training time and the respective 95% confidence intervals. The obtained

median curves are presented in Fig. 2. The figure shows that for the ALL

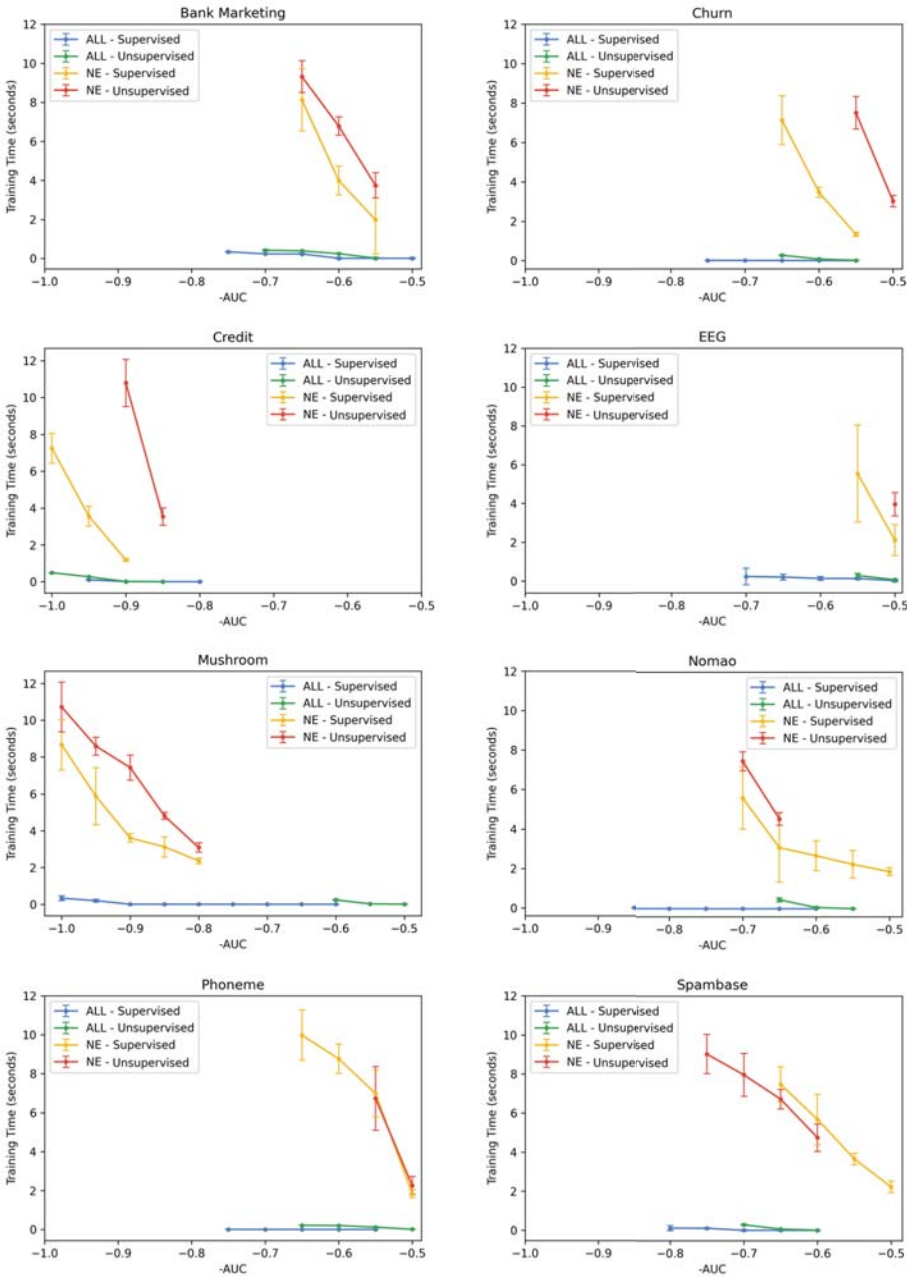


Figure 2: AutoOC experimental results (points denote the Wilcoxon median values and whiskers represent the respective 95% confidence intervals).

setups, the results are usually close in terms of training time, presenting differences that depend on the dataset but that tend to be small. As for the predictive performance, the supervised ALL tends to produce better AUC scores (e.g., Churn, EEG, Mushroom, Nomao, Phoneme, Spambase). As for the NE setups, the results usually present higher training times than the ALL setups. Moreover, the 95% confidence intervals usually do not overlap with the ALL setups, showing statistically significant differences.

5.4. Credit card dataset results

For more detailed results, we present in this section additional analyses of one of the datasets used in the experiments. We chose the Credit Card dataset to perform these analyses for two main reasons. Firstly, this dataset is a very accurate representation of a typical OCC learning scenario, since it has a large number of examples (284,807) and presents a huge unbalance between classes (with more than 99% of examples belonging to the "normal" class). Second, it is among the datasets that obtained the best experimental predictive results in Table 6.

The first Credit Card analysis is related to the hypervolume, assuming a reference point of (AUC=0; maximum training time = 15 s). As a demonstration, we selected the first fold for each of the four experiments performed on the Credit Card dataset to evaluate the hypervolume evolution across the GE generations. For each generation, we computed the median hypervolume value of the current Pareto-optimal front. Fig. 3 presents the evolution of the hypervolume measure (in percentage, y-axis) through the 100 generations of the GE optimization (x-axis). The figure includes two plots, one for each validation mode, for a better comparison since different predictive objectives are being considered in each validation mode (AUC for supervised mode and anomaly score for unsupervised mode). The figure shows a fast hypervolume growth in the first 10 generations, even though it continues to increase until the end of the optimization, but at a lower rate. Three of the four curves present a period without significant improvements in the hypervolume in the first half of the optimization process (until generation 50). It is also worth noting that the experiments that used the ALL mode achieved a better final hypervolume percentage than the respective NE experiment. This can be explained by the fact that, even though the ALL mode presented a lower predictive performance than NE, it was able to generate individuals with much lower training time.

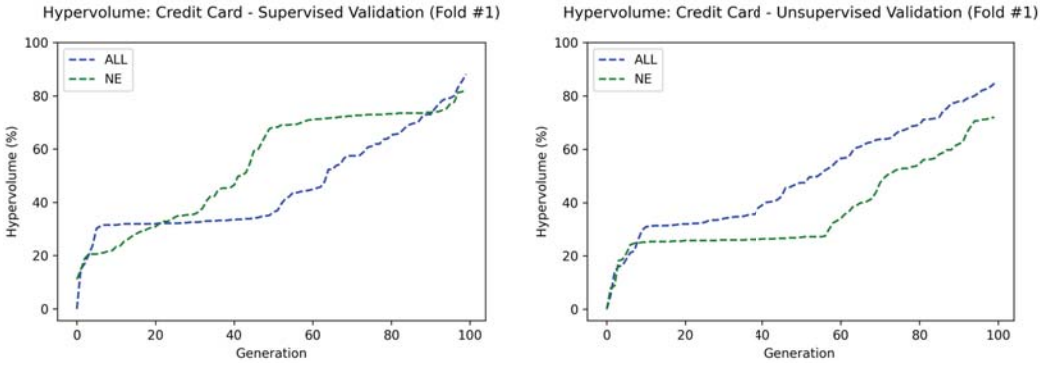


Figure 3: Hypervolume (y-axis, in %) generation evolution (x-axis) for one fold of the Credit Card experiments.

Table 7 provides an additional analysis of the Credit Card dataset experiments regarding the composition of the Pareto front. For each of the four setups (two base learner setups and validation modes), the table details the median number of individuals on the Pareto front by each type of base learner and in total. The table shows that the ALL setups presented a median num-

Table 7: Median number of individuals per base learner on the Pareto Front of the Credit Card dataset experiments.

Dataset	**Base Learner Setup**	**Validation Mode**	**Median Number of Individuals**					
			Pareto Front	**IF**	**LOF**	**OC-SVM**	**AE**	**VAE**
Credit Card	ALL	Supervised	6	4	3	1	1	0.5
	ALL	Unsupervised	6.5	3.5	1	1.5	1	1
	NE	Supervised	9	-	-	-	6	4.5
	NE	Unsupervised	12.5	-	-	-	6	7

ber of Pareto front individuals lower than the NE setup. For the ALL setup, the most common base learner was IF, followed by LOF, and OC-SVM. Both AEs and VAEs are represented in the Pareto curve with a median number of one or fewer individuals. Both NE setups present a larger median number of individuals on the Pareto front (9 and 12.5). Regarding the presence of the base learners, the division between AE and VAE is relatively balanced. As an example, Fig. 4 shows the Pareto front of one of the folds of each of the four Credit Card experiments, detailing the type of base learner from each

point, represented by the initials of the respective base learner.

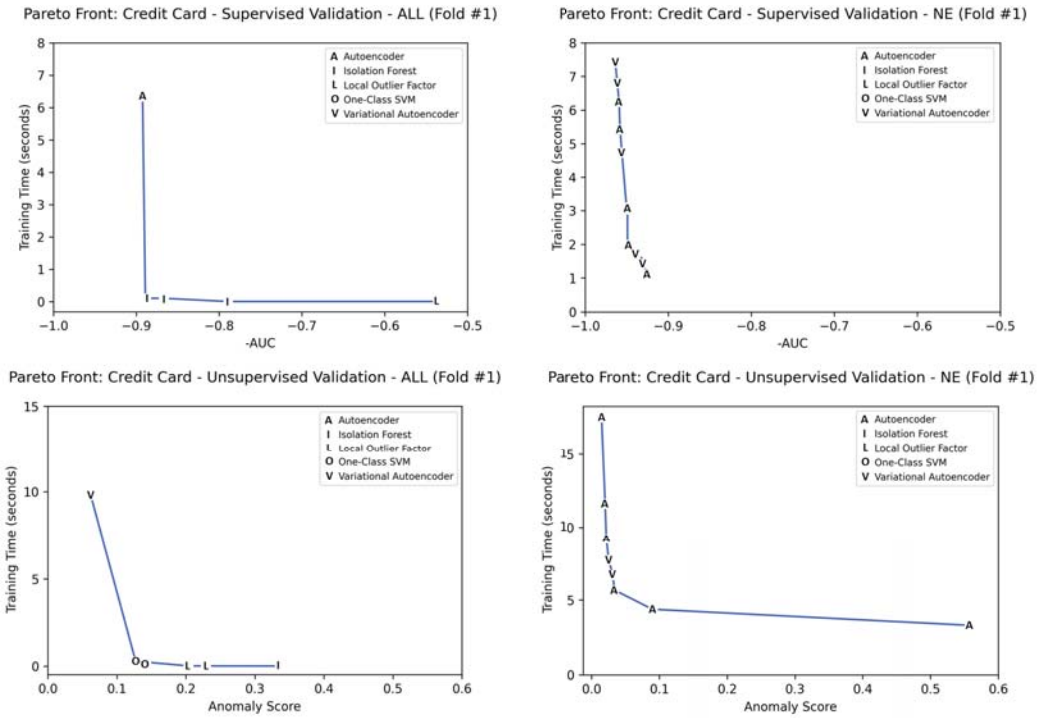


Figure 4: Pareto curves for one fold of the Credit Card experiments. Each Pareto front point is denoted by the initial of the respective base learner.

To study the effect of the periodic sampling mechanism, we replicated the AutoOC experiments for the Credit Card dataset using the full dataset (284,807 rows). Table 8 shows the results obtained on the Credit Card dataset using sampling with s=2,500 (also shown in Table 6) and using the full dataset (without sampling). Regarding the optimization time, the results clearly show that without sampling a substantially higher computational effort is required (the increase is between $\times 8$ and $\times 18$). Similarly, the OCC training time of the individuals was also much higher when using the full dataset. In some cases, it was 200 times higher than the respective experiment with sampling. As for the predictive results, there is only a rather small improvement when adopting the full dataset (e.g., around 3 pp for the Pareto individuals). Given that OCC tasks are often associated with Big Data and several real-world applications tend to require lightweight ML

Table 8: Comparison of AutoOC results for the Credit Card dataset using the sampling mechanism and the full dataset (best values for each measure in **bold**).

Sampling Mode	BL	V*	Median AUC			Median Training Time			Median GE Time
			Pred.	Speed	Pareto	Pred.	Speed	Pareto	
Sampling	All	S	0.92±0.00	0.80±0.08	0.88±0.01	**0.13**±0.04	**0.01**±0.00	**0.04**±0.02	**949**±125
Sampling	All	U	0.97±0.09	0.84±0.01	0.89±0.04	0.48±0.04	0.01±0.00	0.15±0.02	1,191±448
Sampling	NE	S	**0.98**±0.00	**0.93**±0.00	**0.95**±0.00	7.25±0.81	1.19±0.08	3.56±0.54	2,211±202
Sampling	NE	U	0.91±0.10	0.89±0.01	0.90±0.02	10.79±1.28	2.54±0.58	5.31±0.47	4,208±1430
No Sampling	All	S	0.95±0.01	0.87±0.02	0.89±0.02	**38.88**±3.12	2.04±0.41	**23.75**±1.50	17,274±1,026
No Sampling	All	U	0.98±0.01	0.89±0.03	0.91±0.01	45.41±5.95	**1.97**±0.35	27.28±3.78	**16,756**±970
No Sampling	NE	S	**0.99**±0.00	0.94±0.01	**0.96**±0.00	421.57±62.24	240.98±22.37	335.81±39.81	38,391±2,489
No Sampling	NE	U	0.98±0.01	**0.95**±0.00	0.96±0.01	393.70±44.92	256.01±15.70	355.65±35.43	36,431±980

* Validation mode: S - Supervised; U - Unsupervised.

models, the results from Table 8 do value the proposed sampling mechanism.

5.5. Comparison with a baseline method and a supervised gold standard

In a last empirical comparison, we contrast the best AutoOC results with a default (not tuned) IF (also trained as AutoOC with s=2,500 random samples) and the best public OpenML results. For each dataset, we show the best median AutoOC AUC score (column **Pred**. from Table 6), the median score obtained by the baseline IF, and the best result published in OpenML (including the AUC score, the used algorithm name, and the number of human ML attempts, described as "runs" in OpenML). It is worth mentioning that this comparison should be viewed with some caution. Firstly, we only compare the predictive performance AUC results and not other measures targeted by AutoOC, such as the total execution time or training time of the ML models. Secondly, AutoOC is fully automated and the best OpenML results were obtained after a large number of ML human expert modeling trials (ranging from 5,463 to 416,606). Thirdly, the OpenML results adopt a particular data preprocessing method and a supervised learning using the complete training datasets. Fourthly, we do not know the exact validation and testing procedures adopted by the OpenML modeling attempts. Thus, rather than assuming an ideal ML comparison, we use the best OpenML results as a "gold standard", denoting a proxy to the upper limit of the best empirical predictive results that can be achieved when using a human expert supervised learning modeling. The results are shown in Table 9.

When comparing the AutoOC results with the baseline IF, it is possible to verify that the best AutoOC results always achieved a median AUC

Table 9: Comparison of the best AutoOC results with a baseline IF and best OpenML public results.

Dataset	Best Results				
	AutoOC	IF	OpenML		
	Score	Score	Score	Algorithm	Runs
Bank Marketing	0.723	0.546	0.938	XGBoost	40,465
Churn	0.746	0.603	0.932	GBM*	5,463
Credit Card	0.976	0.883	0.942	GBM*	416,606
EEG	0.675	0.505	0.998	SVM*	97,277
Mushroom	1.000	0.782	0.998	SVM	12,556
Nomao	0.830	0.668	0.953	Decision Tree	32,749
Phoneme	0.743	0.510	0.971	AdaBoost*	113,799
Spambase	0.806	0.524	0.989	XGBoost	58,350

*Algorithm used in a pipeline (with one or more preprocessing steps). Algorithm acronyms: GBM – Gradient Boosting Machine; SVM – Support Vector Machine.

higher than the IF. The differences between the best AutoOC and IF results ranged from 9 pp and 28 pp, with a median difference of 17 pp. As for the comparison with the best public OpenML results, the supervised human modeling obtains a median overall AUC of 0.96, which is around 0.18 pp higher when compared with the AutoOC (median AUC of 0.78). While these results were expected, it should be highlighted that most best AutoOC results are of quality, obtaining a good discrimination (AUC>70%) for three datasets (Bank Marketing, Churn, and Phoneme), a very good predictive performance (AUC>80%) in two cases (Nomao and Spambase), and an excellent discrimination (AUC>90%) in two datasets (Credit Card and Mushroom). We particularly highlight the two excellent AUC results that even outperformed the best OpenML public results. Indeed, this corresponds to a high-quality AutoOC performance behavior, since the best OpenML supervised results were obtained after 12,556 (Mushroom) and 416,606 (Credit Card) human attempts.

6. Conclusions

In this work, we presented AutoOC, which consists of a computationally efficient Grammatical Evolution (GE) to automate the design of lightweight One-Class Classification (OCC) Machine Learning (ML) models. We particularly explore two AutoOC variants: a pure Neuroevolution (NE) that evolves two types of deep learning Autoencoders (AEs), standard dense AE and Variational Autoencoder (VAE); and a general Automated Machine Learning (AutoML) version termed ALL and that searches for the best of five OCC algorithms, namely Isolation Forest (IF), Local Outlier Factor (LOF), One-Class SVM (OC-SVM), AE and VAE. The proposed GE adopts an evolutionary multi-objective optimization approach, aiming to maximize the predictive performance of the OCC learners while minimizing their training time. Moreover, it includes two mechanisms to speed up the execution time, a periodic sampling of the training data and a fitness evaluation parallelization by using a multi-core processing. To the best of our knowledge, this is the first time that GE has been applied as a NE and AutoML for OCC tasks.

A large set of empirical experiments was held, considering eight public domain datasets retrieved from the OpenML platform, two GE variants (NE and ALL) and two validation scenarios (unsupervised and supervised). Overall, competitive results were achieved by the proposed AutoOC, which is capable of modeling large datasets using a reasonable amount of computational resources. For instance, for the largest analyzed dataset (Credit Card, which contains around 285 thousand examples) and supervised validation mode, the median execution time of AutoOC was around 16 minutes for the general ALL AutoML and around 37 minutes for the NE. Moreover, the optimized One-Class Classification (OCC) models require a reduced training time. For example, when assuming the sampled s=2,500 training examples and the best Pareto predictive performance results, the ALL setup requires a median training time that is lower than 1 s, while the NE variant optimizes AEs that need a median training time of 8.2 s. As for the predictive performance AutoOC results, quality Area Under the Curve (AUC) of the Receiver Operating Characteristic (ROC) curve values (e.g., >70%) were obtained for seven of the eight analyzed datasets. The AutoOC tool clearly outperformed a baseline IF and even managed to surpass the best public OpenML human modeling approach for two datasets (Credit and Mushroom).

In future work, we intend to explore the use of other NE algorithms, such as Genetic Programming, in the context of OCC. Additionally, we intend to

test our method on a wider range of datasets and add more OCC algorithms to the grammar to further validate its effectiveness.

Acknowledgments

We wish to thank the anonymous reviewers for their helpful comments.

References

[1] K. O. Stanley, D. B. D'Ambrosio, J. Gauci, A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks, Artificial Life 15 (2) (2009) 185–212. doi:10.1162/artl.2009.15.2.15202.

[2] D. Floreano, P. Dürr, C. Mattiussi, Neuroevolution: From Architectures to Learning, Evolutionary Intelligence 1 (1) (2008) 47–62. doi:10.1007/s12065-007-0002-4.

[3] P. Cortez, P. J. Pereira, R. Mendes, Multi-step Time Series Prediction Intervals Using Neuroevolution, Neural Computing and Applications 32 (13) (2020) 8939–8953. doi:10.1007/s00521-019-04387-3.

[4] D. Baymurzina, E. A. Golikov, M. S. Burtsev, A Review of Neural Architecture Search, Neurocomputing 474 (2022) 82–93. doi:10.1016/j.neucom.2021.12.014.

[5] L. Ferreira, A. L. Pilastri, C. M. Martins, P. M. Pires, P. Cortez, A Comparison of AutoML Tools for Machine Learning, Deep Learning and XGBoost, in: International Joint Conference on Neural Networks, IJCNN 2021, Shenzhen, China, July 18-22, 2021, IEEE, 2021, pp. 1–8. doi:10.1109/IJCNN52387.2021.9534091.

[6] T. Cetto, J. Byrne, X. Xu, D. Moloney, Size/Accuracy Trade-Off in Convolutional Neural Networks: An Evolutionary Approach, in: L. Oneto, N. Navarin, A. Sperduti, D. Anguita (Eds.), Recent Advances in Big Data and Deep Learning, Proceedings of the INNS Big Data and Deep Learning Conference INNSBDDL 2019, held at Sestri Levante, Genova, Italy 16-18 April 2019, Springer, 2019, pp. 17–26. doi:10.1007/978-3-030-16841-4_3.

[7] T. Z. Miranda, D. B. Sardinha, M. P. Basgalupp, R. Cerri, A New Grammatical Evolution Method for Generating Deep Convolutional Neural Networks with Novel Topologies, in: J. E. Fieldsend, M. Wagner (Eds.), GECCO '22: Genetic and Evolutionary Computation Conference, Companion Volume, Boston, Massachusetts, USA, July 9 - 13, 2022, ACM, 2022, pp. 663–666. doi:10.1145/3520304.3529025.

[8] M. M. Moya, D. R. Hush, Network Constraints and Multi-objective Optimization for One-Class Classification, Neural Networks 9 (3) (1996) 463–474. doi:10.1016/0893-6080(95)00120-4.

[9] P. Zola, P. Cortez, E. Brentari, Twitter Alloy Steel Disambiguation and User Relevance via One-Class and Two-Class News Titles Classifiers, Neural Computing and Applications 33 (4) (2021) 1245–1260. doi:10.1007/s00521-020-04991-8.

[10] N. Seliya, A. A. Zadeh, T. M. Khoshgoftaar, A Literature Review on One-Class Classification and its Potential Applications in Big Data, Journal of Big Data 8 (1) (2021) 122. doi:10.1186/s40537-021-00514-x.

[11] P. Arregoces, J. Vergara, S. A. Gutierrez, J. F. Botero, Network-based Intrusion Detection: A One-class Classification Approach, in: 2022 IEEE/IFIP Network Operations and Management Symposium, NOMS 2022, Budapest, Hungary, April 25-29, 2022, IEEE, 2022, pp. 1–6. doi:10.1109/NOMS54207.2022.9789927.

[12] L. Ferreira, A. Pilastri, F. Romano, P. Cortez, Using Supervised and One-Class Automated Machine Learning for Predictive Maintenance, Applied Soft Computing 131 (2022) 109820. doi:10.1016/j.asoc.2022.109820.

[13] D. Ribeiro, L. M. Matos, G. Moreira, A. L. Pilastri, P. Cortez, Isolation Forests and Deep Autoencoders for Industrial Screw Tightening Anomaly Detection, Computers 11 (4) (2022) 54. doi:10.3390/computers11040054.

[14] C. Ryan, M. O'Neill, J. Collins, Handbook of Grammatical Evolution, Vol. 1, Springer, 2018.

[15] P. J. Pereira, P. Cortez, R. Mendes, Multi-objective Grammatical Evolution of Decision Trees for Mobile Marketing User Conversion Prediction, Expert Systems with Applications 168 (2021) 114287. doi:10.1016/j.eswa.2020.114287.

[16] J. Vanschoren, J. N. van Rijn, B. Bischl, L. Torgo, OpenML: Networked Science in Machine Learning, ACM SIGKDD Explorations Newsletter 15 (2) (2013) 49–60. doi:10.1145/2641190.2641198.

[17] P. Balaprakash, A. Tiwari, S. M. Wild, L. Carrington, P. D. Hovland, AutoMOMML: Automatic Multi-objective Modeling with Machine Learning, in: J. M. Kunkel, P. Balaji, J. J. Dongarra (Eds.), High Performance Computing - 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings, Vol. 9697 of Lecture Notes in Computer Science, Springer, 2016, pp. 219–239. doi:10.1007/978-3-319-41321-1_12.

[18] A. G. C. de Sá, W. J. G. S. Pinto, L. O. V. B. Oliveira, G. L. Pappa, RECIPE: A Grammar-Based Framework for Automatically Evolving Classification Pipelines, in: J. McDermott, M. Castelli, L. Sekanina, E. Haasdijk, P. García-Sánchez (Eds.), Genetic Programming - 20th European Conference, EuroGP 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings, Vol. 10196 of Lecture Notes in Computer Science, 2017, pp. 246–261. doi:10.1007/978-3-319-55696-3_16.

[19] R. de Lima Thomaz, P. C. Carneiro, J. E. Bonin, T. A. A. Macedo, A. C. Patrocinio, A. B. Soares, Novel Mahalanobis-based Feature Selection Improves One-Class Classification of Early Hepatocellular Carcinoma, Medical & Biological Engineering & Computing 56 (5) (2018) 817–832. doi:10.1007/s11517-017-1736-5.

[20] Z. Chen, C. K. Yeo, B. Lee, C. T. Lau, Y. Jin, Evolutionary Multiobjective Optimization Based Ensemble Autoencoders for Image Outlier Detection, Neurocomputing 309 (2018) 192–200. doi:10.1016/j.neucom.2018.05.012.

[21] S. Estevez-Velarde, Y. Gutiérrez, A. Montoyo, Y. Almeida-Cruz, AutoML Strategy Based on Grammatical Evolution: A Case Study about Knowledge Discovery from Text, in: A. Korhonen, D. R. Traum,

L. Màrquez (Eds.), Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28-August 2, 2019, Volume 1: Long Papers, Association for Computational Linguistics, 2019, pp. 4356–4365. doi:10.18653/v1/p19-1428.

[22] C. H. N. L. Jr., H. J. C. Barbosa, Auto-CVE: a Coevolutionary Approach to Evolve Ensembles in Automated Machine Learning, in: A. Auger, T. Stützle (Eds.), Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2019, Prague, Czech Republic, July 13-17, 2019, ACM, 2019, pp. 392–400. doi:10.1145/3321707.3321844.

[23] S. Gardner, O. Golovidov, J. Griffin, P. Koch, W. Thompson, B. Wujek, Y. Xu, Constrained Multi-Objective Optimization for Automated Machine Learning, in: L. Singh, R. D. D. Veaux, G. Karypis, F. Bonchi, J. Hill (Eds.), 2019 IEEE International Conference on Data Science and Advanced Analytics, DSAA 2019, Washington, DC, USA, October 5-8, 2019, IEEE, 2019, pp. 364–373. doi:10.1109/DSAA.2019.00051.

[24] F. Assunção, N. Lourenço, B. Ribeiro, P. Machado, Evolution of Scikit-Learn Pipelines with Dynamic Structured Grammatical Evolution, in: P. A. Castillo, J. L. J. Laredo, F. F. de Vega (Eds.), Applications of Evolutionary Computation - 23rd European Conference, EvoApplications 2020, Held as Part of EvoStar 2020, Seville, Spain, April 15-17, 2020, Proceedings, Vol. 12104 of Lecture Notes in Computer Science, Springer, 2020, pp. 530–545. doi:10.1007/978-3-030-43722-0_34.

[25] L. A. Moctezuma, M. Molinas, Multi-objective Optimization for EEG Channel Selection and Accurate Intruder Detection in an EEG-based Subject Identification System, Scientific Reports 10 (1) (2020) 1–12. doi:10.1038/s41598-020-62712-6.

[26] S. Estevez-Velarde, Y. Gutiérrez, Y. Almeida-Cruz, A. Montoyo, General-purpose Hierarchical Optimisation of Machine Learning Pipelines with Grammatical Evolution, Information Sciences 543 (2021) 58–71. doi:10.1016/j.ins.2020.07.035.

[27] R. Marinescu, A. Kishimoto, P. Ram, A. Rawat, M. Wistuba, P. P. Palmes, A. Botea, Searching for Machine Learning Pipelines Using a

Context-Free Grammar, in: Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021, AAAI Press, 2021, pp. 8902–8911. doi:10.1609/aaai.v35i10.17077.

[28] S. Mahjoubi, R. Barhemat, P. Guo, W. Meng, Y. Bao, Prediction and Multi-objective Optimization of Mechanical, Economical, and Environmental Properties for Strain-hardening Cementitious Composites (SHCC) Based on Automated Machine Learning and Metaheuristic Algorithms, Journal of Cleaner Production 329 (2021) 129665. doi:10.1016/j.jclepro.2021.129665.

[29] S. Gardner, O. Golovidov, J. Griffin, P. Koch, R. Shi, B. Wujek, Y. Xu, Fair AutoML Through Multi-objective Optimization (2021).

[30] J. M. Moyano, S. Ventura, Auto-adaptive Grammar-Guided Genetic Programming Algorithm to Build Ensembles of Multi-Label Classifiers, Information Fusion 78 (2022) 1–19. doi:10.1016/j.inffus.2021.07.005.

[31] F. Pfisterer, Democratizing Machine Learning (2022). doi:10.5282/edoc.30947.

[32] M. Hirzel, K. Kate, P. Ram, A. Shinnar, J. Tsay, Gradual AutoML using Lale, in: A. Zhang, H. Rangwala (Eds.), KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 14 - 18, 2022, ACM, 2022, pp. 4794–4795. doi:10.1145/3534678.3542630.

[33] Chris Thornton and Frank Hutter and Holger H. Hoos and Kevin Leyton-Brown, Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms, in: The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013, ACM, 2013, pp. 847–855. doi:10.1145/2487575.2487629.

[34] M. O'Neill, C. Ryan, Grammatical evolution, IEEE Transactions on Evolutionary Computation 5 (4) (2001) 349–358. doi:10.1109/4235.942529.

[35] T. Nyathi, N. Pillay, Comparison of a Genetic Algorithm to Grammatical Evolution for Automated Design of Genetic Programming Classification Algorithms, Expert Systems with Applications 104 (2018) 213–234. doi:10.1016/j.eswa.2018.03.030.

[36] K. Deb, S. Agrawal, A. Pratap, T. Meyarivan, A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II, IEEE Transactions on Evolutionary Computation 6 (2) (2002) 182–197. doi:10.1109/4235.996017.

[37] C. A. C. Coello, G. B. Lamont, D. A. van Veldhuizen, Evolutionary Algorithms for Solving Multi-objective Problems, Second Edition, Genetic and Evolutionary Computation Series, Springer, 2007.

[38] T. Fawcett, An introduction to ROC analysis, Pattern Recognition Letters 27 (8) (2006) 861–874. doi:10.1016/j.patrec.2005.10.010.

[39] G. Coelho, L. M. Matos, P. J. Pereira, A. L. Ferreira, A. L. Pilastri, P. Cortez, Deep Autoencoders for Acoustic Anomaly Detection: Experiments with Working Machine and In-vehicle Audio, Neural Computing and Applications 34 (22) (2022) 19485–19499. doi:10.1007/s00521-022-07375-2.

[40] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, software available from tensorflow.org (2015).
URL https://www.tensorflow.org/

[41] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. VanderPlas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine Learning in Python, Journal of Machine Learning Research 12 (2011) 2825–2830. doi:10.5555/1953048.2078195.

[42] Scikit-Learn, Local Outlier Factor (2022).
URL https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.LocalOutlierFactor.html

[43] Scikit-Learn, Isolation Forest (2022).
URL https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html

[44] Scikit-Learn, One-Class SVM (2022).
URL https://scikit-learn.org/stable/modules/generated/sklearn.svm.OneClassSVM.html

[45] TensorFlow, Convolutional Variational Autoencoder (2022).
URL https://www.tensorflow.org/tutorials/generative/cvae

[46] TensorFlow, Intro to Autoencoders (2022).
URL https://www.tensorflow.org/tutorials/generative/autoencoder

[47] M. M. Breunig, H. Kriegel, R. T. Ng, J. Sander, LOF: Identifying Density-Based Local Outliers, in: W. Chen, J. F. Naughton, P. A. Bernstein (Eds.), Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA, ACM, 2000, pp. 93–104. doi:10.1145/342009.335388.

[48] F. T. Liu, K. M. Ting, Z. Zhou, Isolation Forest, in: Proceedings of the 8th IEEE International Conference on Data Mining (ICDM 2008), December 15-19, 2008, Pisa, Italy, IEEE Computer Society, 2008, pp. 413–422. doi:10.1109/ICDM.2008.17.

[49] B. Schölkopf, J. C. Platt, J. Shawe-Taylor, A. J. Smola, R. C. Williamson, Estimating the Support of a High-Dimensional Distribution, Neural Computation 13 (7) (2001) 1443–1471. doi:10.1162/089976601750264965.

[50] K. Patra, R. N. Sethi, D. K. Behera, Anomaly Detection in Rotating Machinery using Autoencoders Based Onbidirectional LSTM and GRU Neural Networks, Turkish Journal of Electrical Engineering and Computer Sciences 30 (4) (2022) 1637–1653. doi:10.55730/1300-0632.3870.

[51] H. Gao, B. Qiu, R. J. Duran Barroso, W. Hussain, Y. Xu, X. Wang, TSMAE: A Novel Anomaly Detection Approach for Internet of Things Time Series Data Using Memory-Augmented Autoencoder, IEEE Transactions on Network Science and Engineering (2022) 1–1doi:10.1109/TNSE.2022.3163144.

[52] T. Hastie, R. Tibshirani, J. H. Friedman, The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd Edition, Springer Series in Statistics, Springer, 2009. doi:10.1007/978-0-387-84858-7.

[53] D. P. Kingma, M. Welling, An Introduction to Variational Autoencoders, Foundations and Trends in Machine Learning 12 (4) (2019) 307–392. doi:10.1561/2200000056.

[54] A. S. Edun, C. LaFlamme, S. R. Kingston, C. M. Furse, M. A. Scarpulla, J. B. Harley, Anomaly Detection of Disconnects Using SSTDR and Variational Autoencoders, IEEE Sensors Journal 22 (4) (2022) 3484–3492. doi:10.1109/JSEN.2022.3140922.

[55] M. Fenton, J. McDermott, D. Fagan, S. Forstenlechner, E. Hemberg, M. O'Neill, PonyGE2: Grammatical Evolution in Python, in: P. A. N. Bosman (Ed.), Genetic and Evolutionary Computation Conference, Berlin, Germany, July 15-19, 2017, Companion Material Proceedings, ACM, 2017, pp. 1194–1201. doi:10.1145/3067695.3082469.

[56] L. M. Matos, J. Azevedo, A. Matta, A. L. Pilastri, P. Cortez, R. Mendes, Categorical Attribute traNsformation Environment (CANE): A python module for categorical to numeric data preprocessing, Software Impacts 13 (2022) 100359. doi:10.1016/j.simpa.2022.100359.

[57] M. Hollander, D. A. Wolfe, E. Chicken, Nonparametric Statistical Methods, John Wiley & Sons, 2013.