TU Dortmund

Master Thesis

Summer Semester 2024

Thesis : Understanding the Impact of Automated Hyperparameter Tuning on Anomaly Detection Methods

Supervisors:
Prof. Dr. Emmanuel Müller,
M. Sc. Simon Klüttermann

Author: Jessia Erappakkal Joby
Matriculation Number: 230299

August 7, 2024

Contents

Abstract

In machine learning, anomaly detection plays an important role in detecting data points that differ from the normal data distribution. It is employed in various fields, such as finance transactions, healthcare monitoring, and the machine industry.

Tuning the hyperparameter is crucial in maximizing the model's performance since the performance of the machine learning algorithms depends on the chosen hyperparameters. In our research, different algorithms are considered for hyperparameter tuning in order to evaluate how algorithms behave after tuning and determine whether some algorithms benefit more or less from tuning than others. Manual tuning is time-consuming as it requires more experiments with different hyperparameter configurations.

This thesis uses automated hyperparameter tuning "FLAML" to achieve faster optimization than manual tuning. We apply hyperparameter tuning to datasets from different fields. The results from our research show that the models with the tuned hyperparameter have improved in performance compared to the default hyperparameters. The optimal hyperparameter obtained in this study can be directly applied to practical applications as it is tuned across datasets from various domains. This can be helpful for the researcher and data scientists as it saves time spent on hyperparameter tuning.

1 Introduction

In machine learning, anomaly detection is an important research field since it can suggest defects, system malfunctions, and malicious activity. Anomalies are data points that behave significantly different from normal data distribution. The primary goal of anomaly detection algorithms is to find anomalies, and by detecting these, the algorithm can prevent losses or faults (Han et al., 2022). These algorithms find distinguishing data points from normal distributions difficult, as anomalies are rare and domain-dependent, and noise in the data can also be challenging.

Unsupervised anomaly detection is a machine learning algorithm that identifies the anomalies without using labeled data (Goldstein and Uchida, 2016). Unsupervised algorithms are employed where there is a lack of labeled data because sometimes obtaining labeled data can be expensive. Anomaly detection algorithm identifies anomalies by learning the normal data distribution, clustering behavior, or reconstruction error. For example, the original data is reconstructed from the compressed representation in models like Autoencoder (Chen et al., 2018). During the reconstruction of original data, the data points that have high reconstruction errors are considered anomalies.

Hyperparameters are parameters that must be set before the training as they are not learned during the training process. Hyperparameter tuning is a major step in machine learning that involves determining the optimal hyperparameters for the algorithms. The optimal hyperparameters for the algorithm can significantly improve the model's performance (Yu and Zhu, 2020). Manually choosing the hyperparameters for the algorithms requires extensive experimentation and domain expertise, as they can greatly influence the performance. The manual process takes a longer time because it involves several experiments and configurations to find the optimal hyperparameters.

In our research, an automated machine learning framework such as FLAML (Fast and Lightweight AutoML) is employed to tune the hyperparameters to overcome the problems faced by manual tuning. FLAML uses cost-effective optimization to find the optimal hyperparameter that balances the performance and the cost (Wang et al., 2021). This study focuses only on tabular datasets; a total of 498 datasets are used. We tune eight anomaly detection algorithms twice, first on the subset of 170 datasets, each with a size lower than 0.1 MB, and then on the entire set of 498 datasets. The tuning approach speeds up the optimization process on smaller subsets, allowing for faster testing. We then tune models on the whole dataset to guarantee the models are effective across a wide

range of real-world dataset sizes. The optimal hyperparameters for each algorithm are identified by evaluating their performance on the test datasets. These hyperparameters maximize the average AUC score across all the test datasets. Additionally, after tuning, the optimal hyperparameters are compared with the default hyperparameter setting on the ADBench datasets (Han et al., 2022). This comparison is important for evaluating the effectiveness of the tuned hyperparameters in real-world scenarios, as ADBench provides datasets for benchmarking anomaly detection methods. It also demonstrates the improvements achieved through hyperparameter tuning. The computation time of the algorithm is also measured in addition to the performance of algorithms to provide a reliable comparison.

The motivation for this research is to tune hyperparameters to improve the performance of the unsupervised anomaly detection algorithms. Our goal is to ensure that optimal hyperparameters obtained through our research can be directly applied to real-world applications. This method saves time and resources that would be spent on hyperparameter tuning, ensuring faster implementation of anomaly detection methods. Hyperparameter tuning in unsupervised learning is quite challenging because it is difficult to find which hyperparameters improve anomaly detection without labeled data.

Previous research may have focused on specific algorithms or small-scale datasets, but our study examines multiple anomaly detection methods across a wide range of datasets from various fields. This research demonstrates how automated hyperparameter tuning improves the performance of anomaly detection methods across different algorithms and dataset sizes.

Section 2 provides a literature review that discusses various methods for hyperparameter tuning, including a detailed description of the FLAML tuning approach. In Section 3, both shallow and deep neural network methods and generative adversarial learning methods like AnoGAN are introduced and explained. It also includes a discussion of the evaluation metrics used. Section 4 explains the approach taken in this research, including the reasons for choosing specific methods and using FLAML. Section 5 illustrates the experimentation results, and a summary of findings and a brief discussion about the future work is elaborated in Section 6.

2 Literature Review

2.1 Anomaly Detection and Hyperparameter Tuning

Anomalies are data points or patterns in a dataset that deviate significantly from the normal data distribution. They indicate rare or unusual events in the dataset due to malicious activity, fraud, or errors that are inconsistent with the expected behavior of the data. Analyzing these anomalies can provide important insights into underlying problems. Anomaly detection is the algorithm for identifying data points that are categorized as anomalies or distinguishing the data points between normal and anomalous data points (Goldstein and Uchida, 2016).

Unsupervised anomaly detection is a technique that identifies anomalies in a dataset without the use of labels (normal data points are labeled as "0" and anomalous data points are labeled as "1"). This method is useful when label data is challenging to obtain, and it is usually impractical to have labeled data. The distances, densities, or data distribution are used by unsupervised methods to identify normal or anomalous data points without the use of labels. This thesis uses unsupervised anomaly detection methods and anomaly scores as an output of the anomaly detection algorithms, which indicate the likelihood of the data point being an anomaly (Goldstein and Uchida, 2016).

Hyperparameters are configuration settings used to tune the performance of a machine learning model. Hyperparameters must be set prior to the learning process, unlike the model parameters, which are learned during the training process from the input data. Tuning the hyperparameters can improve the performance of the model by finding the optimal configuration setting that is suitable for the machine learning algorithm. The model's performance can be influenced by the chosen values of the hyperparameters, and finding optimal hyperparameter values can be challenging without the labeled data for validation in unsupervised anomaly detection methods. In the absence of labeled data, it is difficult to determine which hyperparameters improve the detection of anomalies. The automated hyperparameter tuning approach can experiment with a wide range of values and combinations in order to quickly determine the optimal hyperparameter configuration that improves the model's performance on the test data compared to the manual tuning. The challenge of automated hyperparameter tuning is that there can be complex search space, and evaluating each configuration setting can be computationally expensive for high-dimensional datasets and deep-learning models (Yu and Zhu, 2020).

2.2 Meta-Learning Based Outlier Detection (MetaOD)

In the literature (Zhao et al., 2021), the author proposed a meta-learning based technique (MetaOD), where the hyperparameter configuration and model selection are chosen based on the model's past experience on historical datasets. The model for the new test dataset is selected based on the similarity of the features (meta-features) between the historical datasets and the new test dataset. The metaOD trains different anomaly detection models on the historical or meta-train datasets, where each model is trained as pair {model, configuration}. During the training process, the method extracts meta-features (essential features of a dataset), which are later used to compare the similarity between the historical and test datasets. For the prediction of the test dataset, it extracts the meta-features of the new dataset. It selects the set of models and their hyperparameter values based on the meta-features extracted that are similar to the meta-train datasets. The model and their hyperparameter values that achieve the highest performance are selected as the best model from the set of models. The metaOD proposed method converts the continuous values of hyperparameters into discrete values, reducing the complexity of hyperparmater tuning. As a result, the range of hyperparameter combinations to be evaluated is reduced. However, during the meta-learning process, the meta-training dataset has ground-truth labels. This requires metaOD to use labeled data for meta-learning, which contrasts with unsupervised anomaly detection, where labels are not used during the training process.

2.3 Automated Machine Learning (AutoML)

Automated Machine Learning (AutoML) involves automating four processes: data preparation, feature processing, model generation, and estimation. One of the AutoML frameworks is auto-sklearn (Feurer et al., 2020), which is mainly used for supervised training datasets. It relies on the widely used Python machine learning library Scikit-learn (Pedregosa et al., 2011) and expands it by automating the tuning of hyperparameters and the model selection process. The tuning procedure for auto-sklearn is based on Bayesian optimization. The Bayesian optimization method begins with a simple model, a Gaussian process. Based on the evaluation conducted, the Gaussian process captures the relationship between the hyperparameters and the model's performance. An acquisition function chooses the next hyperparameter set to be evaluated. Bayesian optimization is an iterative process that can quickly find the hyperparameters that optimize the model's

performance. Auto-sklearn uses meta-learning for the initial hyperparameter configuration, where it compares the meta-features of the current dataset with the meta-training datasets to find any similarities. Moreover, auto-sklearn uses ensemble learning to combine models that perform better than the single models. Even though auto-sklearn is highly automated, it does not provide flexibility for custom tasks. It is not suitable for unsupervised machine learning tasks, as they need labels during training to evaluate the performance of hyperparameter configurations (Feurer et al., 2020).

H2O is another AutoML framework utilized for automated supervised machine learning algorithms, especially for tabular datasets. To produce high performance, H2O AutoML trains and combines various base models, such as random forest (Qi, 2012), deep learning networks (Sewak and Sahay, 2020), and gradient boosting machines (Natekin and Knoll, 2013), to form stacked ensemble models. This framework uses random search (Bergstra and Bengio, 2012) for hyperparameter optimization. In random search optimization, the models are trained using the random selection of hyperparameters from defined ranges, and their performance is assessed based on an evaluation metric like AUC score (Bouckaert, 2006). When the base models are individually robust and have uncorrelated errors, the stacked ensembles are useful. H2O is capable of managing efficiently large datasets and complex models. Despite the many models the H2O AutoML offers, it does not include every algorithm or does not provide customization where the user would like to experiment (LeDell and Poirier, 2020).

In our research, FLAML (Fast and Lightweight Automated Machine Learning) is an automated machine learning framework used for model selection and hyperparameter tuning. It can evaluate many machine learning models and their hyperparameters quickly and efficiently. Unlike metaOD, FLAML uses a wide range of hyperparameter values for tuning and understands the interaction between various hyperparameters to decide on the next best search location. It explores the hyperparameter configuration to achieve a tradeoff between the two key goals: reduce model error and minimize computation cost. This means that FLAML aims to identify the optimal hyperparameter quickly while ensuring the model's prediction is as accurate as possible. FLAML initially begins with low-cost trials and with inaccurate models, then moves gradually to expensive trials and accurate models.

FLAML has two built-in models: 'flaml.AutoML' and 'flaml.tune'. 'flaml.AutoML' is used for hyperparameter tuning and model selection, especially supervised learning, where the data structure contains features and labels. This research uses a built-in

FLAML model called tune (flaml.tune), particularly suitable for unsupervised learning tasks, which tunes the set of hyperparameters in three steps: tuning objective, search space, and tuning constraints. The tuning objective represents performance metrics used to evaluate the training model during the tuning process. The search space involves specifying the range of hyperparameter values of the model for which we need to tune. A user has the option to limit the resources by providing tuning constraints such as time budget in seconds and the number of trials required for tuning.

Unlike MetaOD, FLAML does not require meta-learning, which means many meta-training datasets need not be collected. In comparison with auto-sklearn, FLAML allows users to tailor the tuning process to meet their specific needs. Auto-sklearn can be used only with the Python Scikit-learn library, whereas FLAML can be integrated with many machine learning frameworks, which leads to adaptability. Furthermore, FLAML is preferred over H2O AutoML because of its cost-effective optimization. FLAML is primarily designed to reduce the computational cost and improve performance by automated hyperparameter tuning. FLAML especially provides customizability, where users can limit the available resources by specifying the time constraints and number of optimization steps. In contrast, H2O doesn't provide much customization of models. The drawback of FLAML is that exploring large hyperparameter spaces can be computationally expensive. If the computational resources are limited, conducting extensive hyperparameter searches can be challenging (Wang et al., 2021).

3 Methodology

In the following section, various methods are introduced and explained. The software Visual Studio IDE (Corporation, 2022) and Python (Foundation, 2022) (version 3.11.5) with packages PyOD (Zhao et al., 2019), Scikit-learn (Pedregosa et al., 2011), Matplotlib (Hunter, 2007) are used for all calculations and visualizations.

3.1 Shallow Methods

K - Nearest Neighbors (KNN)

The k-Nearest Neighbor is the most commonly used algorithm for classification, Regression, and Anomaly detection. In this thesis, KNN is employed to identify anomalies by

calculating the anomaly score of an observation based on its distance to the k^{th} nearest neighbor within the dataset. The most commonly used distance metrics (p) for finding the distance between two observations are Manhattan, Euclidean, and Minkowski distance. The hyperparameters in KNN are k, which is the method used to calculate the anomaly score (mean, median, and largest) and the distance metric. The Minkowski distance is defined as follows (Kramer, 2013),

$$d_{\text{minkowski}}(x_i, x_j) = (\sum_{l=1}^{q} |x_{i,l} - x_{j,l}|^p)^{1/p},$$

where x_i and x_j are the i^{th} and j^{th} observation, $i = 1, 2, ..., n$, $j = 1, 2, ..., n$, respectively. n represents the total number of observations. $x_{i,l}$ and $x_{j,l}$ are the l^{th} feature of the i^{th} and j^{th} observation, $l = 1, 2, ..., q$, q is the total number of features. when the distance metric $p = 1$, the Minkowski distance equals to the Manhattan distance. when $p = 2$, the Minkowski distance is equal to the Euclidean distance.

The anomaly score is defined as,

$$\text{AnomalyScore}_{\text{mean}}(x_i) = \frac{1}{k} \sum_{j=1}^{k} d_{\text{minkowski}}(x_i, x_j),$$

x_i is the i^{th} observation for which anomaly score is calculated, x_j denotes the j^{th} nearest neighbor to x_i for $j = 1, 2, ..., k$, k represents the number of nearest neighbors. $\text{AnomalyScore}_{\text{mean}}(x_i), \text{AnomalyScore}_{\text{median}}(x_i), \text{AnomalyScore}_{\text{max}}(x_i)$ for an observation x_i is the average distance, median distance and maximum distance to the k-nearest neighbors, respectively. In large datasets, search techniques like ball-tree, KD-tree, and brute force are utilized to find the nearest neighbors effectively (Dang et al., 2015).

Brute Force Search

The brute force search algorithm calculates the distance between the query point q and every other point p in the dataset X. The algorithm calculates the distance to every other point p for a given query point q and chooses k closest points. The computational complexity for this algorithm is $O(n^2)$, where n is the total number of points in the dataset. So, this algorithm is computationally expensive for large datasets (Garcia et al., 2008).

KD-tree (k-dimensional tree)

In this search algorithm, the dataset is partitioned into k-dimensional rectangles (hyperrectangles). Here, k represents the number of features in the dataset. The tree is constructed by splitting the median value along the rectangles (hyperrectangles) recursively until each rectangle has only one point in it. For searching the nearest neighbors, the procedure is as follows, Begin with the root node and traverse to the leaf node based on the coordinates of the query point. At each node, update the list of nearest neighbors by calculating the distance to the query point. The computation complexity for building the KD-tree is $O(n\log n)$ and for searching $O(\log n)$ (Hou et al., 2018).

Ball tree

Ball tree is a binary tree (each node has at most two children) where the data is partitioned into balls (hyperspheres). Initially, the entire dataset is a root node. For the node, calculate the centroid of all points within the node. Then, choose two points within the node that are farthest away to act as new centers for the two balls (child nodes). Then, assign each point in the node to one of the two centers based on the distance to the centers from the points. Repeat these steps recursively until a minimum number of points (leaf size) in the node is reached. The steps for searching a ball tree for nearest neighbors are as follows, Begin with the root node, maintain a list to keep a record of the nodes to be explored, and another list for their nearest neighbors and their distances found so far. The node with a minimum distance from the centroid to the query point is explored. If the current node is not the leaf node, the distance from the query point to the centroid of the left and right nodes is added to the list. If the node is a leaf node, calculate the distance from the query point to each point in the node. If the distance is smaller than the current farthest distance in the nearest neighbors list, the list is updated with this point (Dolatshah et al., 2015).

Isolation Forest (IFOR)

Isolation forest is an unsupervised algorithm mainly used for anomaly detection. This algorithm works by constructing multiple Isolation trees (itrees). For constructing itrees (T), a subset is extracted from the dataset $X = x_1, x_2, ..., x_n$ by random sampling. From the subsample, feature A is randomly selected, and a split value p is chosen to be a value between the feature's maximum and minimum values. Now the subsample

is partitioned into two subnodes based on the feature and split value as left node (T_l) and right node (T_r). The left node (T_l) consists of values less than or equal to the split value($x_i(A) \leq p$), whereas the values greater than the split value belongs to the right node ($x_i(A) > p \in T_r$). The subnodes are constructed recursively until one of the requirements is met: 1. The maximum tree height is reached, 2. There is only one data point in the node or has similar data values in the node. For an observation x, path length $h(x)$ is the number of edges from the root node to x. In the isolation forest algorithm, the anomaly score $s(x, n)$ is calculated as follows,

$$s(x, n) = 2^{\frac{-E(h(x))}{c(n)}},$$

$$c(n) = H(n-1) - 2((n-1)/n),$$

$$H(n-1) = \ln(n-1) + \gamma$$

where n denotes the size of the subsample, E($h(x)$) is the average path length from the m, m represents the number of itrees, for a binary tree, the average path length of failed searches is $c(n)$, which is represented by the harmonic number $H(n)$, $\gamma = 0.577215$ is the Euler's constant. The anomaly score ranges from 0 to 1. If $s(x, n) \leq 0.5$, data points are normal points. The data points are considered anomalies if their anomaly scores are greater than 0.5 and close to 1 ($0.5 < s(x, n) \leq 1$). The hyperparameters used prior to the training are the number of trees (m), subsample size(n), and maximum tree depth (Xu et al., 2017).

Local Outlier Factor (LOF)

A density-based algorithm called local outlier factor (LOF) identifies anomalies by calculating local densities for each data point. The densities between the data point and its neighboring points are calculated in order to identify anomalies. The reachability distance ($reachability\_dist(p, o)$) between point p and o from the dataset X is defined as (Cheng et al., 2019),

$$reachability\_dist(p, o) = \max\{k\_distance(o), dist(p, o)\},$$

where $k\_distance(o)$ is the distance of k-th nearest neighbor of point o, $dist(p, o)$ is the distance from point p to o. The local reachability density ($lrd(p)$) is given as,

$$lrd(p) = \left(\frac{\sum_{o \in N_k(p)} reachability\_dist(p, o)}{|N_k(p)|} \right)^{-1},$$

where $|N_k(p)|$ denotes the number of k-nearest neighbors of the point p. Local Outlier Factor ($LOF(p)$) represents the average ratio of the local reachability density of the point p's k-nearest neighbors $lrd(o)$ to the local reachability density of the point p ($lrd(p)$) and is calculated as follows,

$$LOF(p) = \frac{\sum_{o \in N_k(p)} \frac{lrd(o)}{lrd(p)}}{|N_k(p)|}.$$

The $LOF(p)$ close to 1 is considered a normal point since it has a similar local density to its neighbors. It implies that the point p is an anomaly with lower local density compared to its neighbors having $LOF(p)$ greater than 1. If the $LOF(p)$ is less than 1, p has a high local density compared to its neighbors and is not considered to be an anomaly (Cheng et al., 2019).

Clustering-Based Local Outlier Factor (CBLOF)

Clustering based on local outlier factors identifies anomalies by clustering the data points, and then, based on the structure of the data points in the cluster, it assigns an anomaly score. By default, the pyod library CBLOF uses k-means clustering for which the dataset $X = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$, where $\mathbf{x}_i \in \mathbb{R}^d$ denotes a data point in d-dimensional space is partitioned into k clusters.

k-Means Clustering

The k-means is a classical, simple, and unsupervised clustering algorithm. It partitions the dataset into k clusters by an iterative process. Initially, the value of k is fixed, which represents the number of clusters. Then, k cluster centroids are chosen randomly : $\mu_1, ..., \mu_k$, where $\mu_j \in \mathbb{R}^d$. Next, each data point $\mathbf{x}_i$ is assigned to the nearest cluster centroid by calculating Euclidean distance which forms k clusters, $C = \{C_1, C_2, ..., C_k\}$. The new cluster centroids are recalculated based on the current data points in the clusters

as follows,

$$\mu_j = \frac{1}{|C_j|} \sum_{\mathbf{x}_i \in C_j} \mathbf{x}_i,$$

where mean of all points assigned to the cluster C_j is denoted by μ_j, $j = 1, 2, ..., k$, $|C_j|$ represents the number of data points in the cluster C_j. The reassignment steps are repeated until there is no change in the cluster centroids. This algorithm aims to minimize the sum of squared distances from the points to the cluster centroids is given as,

$$J = \arg\min_{j} \sum_{j=1}^{k} \sum_{\mathbf{x}_i \in C_j} ||\mathbf{x}_i - \mu_j||^2,$$

$||\mathbf{x}_i - \mu_j||^2$ is the Euclidean distance between the point $\mathbf{x}_i$ and the cluster centroid μ_j (Sinaga and Yang, 2020).

After clustering, the clusters are classified into either small (SC) or large clusters (LC) based on a threshold value β denotes the minimum number of data points for a large cluster, and α represents the number of data points in the large clusters to the number of data points in the small clusters. The α has a value between 0 and 1. Suppose the clusters are sorted in descending order such that $|C_1| \geq |C_2| \geq ... \geq |C_k|$ and the value of $b = 1, 2, ..., k-1$ is chosen if one of the requirements are satisfied,

$$(|C_1| + |C_2| + ... + |C_b|) \geq n * \alpha$$

$$C_b / C_{b+1} \geq \beta,$$

the small cluster is defined as $SC = C_j | j > b$ and the large cluster is defined as $LC = \{C_j | j \leq b\}$, $j = 1, 2, ..., k$. Then, calculate the cluster-based local outlier factor without weights for each data point $\mathbf{x}_i$ as follows,

$$\text{CBLOF}(\mathbf{x}_i) = \begin{cases} \min\limits_{m \in LC}(||\mathbf{x}_i - \mu_m||), \text{where } \mathbf{x}_i \in C_j, C_j \in SC, \text{for } m = 1, 2, ..., b \\ ||\mathbf{x}_i - \mu_j||, \text{where } \mathbf{x}_i \in C_j, C_j \in LC \end{cases}$$

The CBLOF weighted $\text{CBLOF}_w(\mathbf{x}_i)$ uses the weight as the size of the clusters and is defined as,

$$\text{CBLOF}_w(\mathbf{x}_i) = \begin{cases} |C_j| * \min\limits_{m \in LC}(||\mathbf{x}_i - \mu_m||), \text{where } \mathbf{x}_i \in C_j, C_j \in SC, \text{for } m = 1, 2, ..., b \\ |C_j| * ||\mathbf{x}_i - \mu_j||, \text{where } \mathbf{x}_i \in C_j, C_j \in LC, \end{cases}$$

where $||\mathbf{x}_i - \mu_j||$ represents the Euclidean distance and $|C_j|$ is the size of the cluster C_j. Then, a threshold value decides whether the data point is an anomaly or normal. If a data point has CBLOF greater than the threshold, it is considered to be an anomaly or else considered to be a normal point (Duan et al., 2009).

Support Vector Machine (SVM)

The support vector machine (SVM) algorithm is a supervised machine learning algorithm employed for regression and classification tasks. In supervised learning, the model is trained on the labeled dataset $\mathbf{X} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), ..., (\mathbf{x}_n, y_n)\}$ where $\mathbf{x}_i \in \mathbb{R}^d$ represents d-dimensional space, n is the number of data points, binary class labels $y_i \in \{-1, +1\}$ (classification), and $y_i \in \mathbb{R}$ (regression) denotes the continuous values. SVM can effectively handle both linear and non-linear training data in high-dimensional spaces (Cervantes et al., 2020).

Case 1: Linear Separable Training Data

Considering binary classification task, where labels are $y_i \in \{-1, +1\}$. To find an optimal hyperplane that separates the classes and hyperplanes with maximum margin (distance from the hyperplane to the closest data point $\mathbf{x}_i$ from either class $(+1, -1)$) is the main objective of the SVM. The formula for the hyperplane is defined as $\mathbf{w}^T\mathbf{x}_i + b = 0$. The problem objective can be given as follows,

$$\min_{\mathbf{w},b} ||\mathbf{w}||_2^2$$

subject to,

$$y_i(\mathbf{w}^T\mathbf{x}_i - b) \geq 1, \ \forall i = 1, 2, ..., n$$

The margin that is maximized is represented as follows $\gamma = \frac{1}{||\mathbf{w}||}$. Here, $\mathbf{w} = [w_1, w_2, ..., w_d] \in \mathbb{R}^d$ denotes the weight vector normal (perpendicular) to the hyperplane, d is the number of features or dimensional space, and $b \in \mathbb{R}$ is the bias term, $||\mathbf{w}||_2^2$ is the squared Euclidean norm which is defined as follows,

$$||\mathbf{w}||_2^2 = (\sqrt{w_1 + w_2 + ... + w_d})^2 = \sum_{j=1}^{d} w_j^2$$

The above-stated problem objective is applicable only if the training dataset is linearly separable, that is there are no overlaps or misclassifications. In cases where there are overlaps, and some misclassifications occur in the linear separable dataset, the soft margin approach is used. The problem objective for the soft margin approach is that the hyperplane should meet the following requirements,

$$\min_{\mathbf{w},b,\zeta} ||\mathbf{w}||_2^2 + C\sum_{i=1}^{n}\zeta_i$$

subject to,

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 - \zeta_i,\ \zeta \geq 0,\ \forall i = 1, 2, ..., n$$

Here, ζ_i are the slack variables (non-negative) that allow some points to be within the margin, and C is a regularization parameter, which is a trade-off between the classification error and the maximum-margin (Cervantes et al., 2020).

Kernel functions

In cases where the data cannot be separated linearly, the kernel function (kernel trick) is employed to transform the input data to high-dimensional space, which makes it easier to separate the classes. Some of the commonly used kernel functions are as follows (Patle and Chouhan, 2013),

a) Linear Kernel

Linear kernel calculates the dot product of the input data in the original dimensional space. The linear kernel function ($K(\mathbf{x}_i, \mathbf{x}_j)$) is formulated as, $K(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^T \cdot \mathbf{x}_j$, where $\mathbf{x}_i, \mathbf{x}_j$ are the data for the observations $i, j = 1, 2, ..., n$, n is the size of the training data.

b) Radial Basis Function (RBF) Kernel

Radial basis function kernel or Gaussian kernel is one of the most widely used kernel functions in SVM. The RBF kernel is given as, $K(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma||\mathbf{x}_i - \mathbf{x}_j||^2)$, here γ determines the width of the gaussian function, $||\mathbf{x}_i - \mathbf{x}_j||^2 = \sum_{k=1}^{d}(x_{ik} - x_{jk})^2$ is the squared Euclidean norm, $i, j = 1, 2, ..., n$, d is the number of dimensions or number of features in the dataset.

c) Sigmoid Kernel

The sigmoid kernel (hyperbolic tangent kernel) is based on the sigmoid function and is defined as, $K(\mathbf{x}_i, \mathbf{x}_j) = \tanh(\gamma \mathbf{x}_i^T \mathbf{x}_j + \beta)$, where γ and β are the parameters of the sigmoid function.

Case 2: Non-Linear Separable Training Data

The SVM transforms the data into high-dimensional space using the kernel function, where it can be linearly separated if the training data is not linearly separable. In practical application, The problem objective uses lagrangian multipliers α_i and is defined as,

$$\max_{\alpha} \left(\sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) \right)$$

subject to,

$$0 \leq \alpha_i \leq C, \sum_{i=1}^{n} \alpha_i y_i = 0,$$

$K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j)$ is the kernel function that transforms the input data into high-dimensional space, $\phi(x_i)$ and $\phi(x_j)$ are the feature mapping function that transforms x_i and x_j into high-dimensional space for $i, j = 1, 2, ..., n$, respectively, α_i are the lagrangian multiplier which represents the importance of individual data points in defining the problem objective. α_i is constrained between 0 and C (regularization parameter) (Cervantes et al., 2020).

One-Class Support Vector Machine (OCSVM)

One-class SVM is an unsupervised machine learning algorithm, and it is a type of SVM algorithm used for anomaly detection. The aim of one-class SVM is to find a hyperplane (decision boundary) that optimizes the distance between the data points and the origin. Let us consider the training dataset $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$, n is the size of training data, where $\mathbf{x}_i \in \mathbb{R}^d$ in a d-dimensional space. The decision boundary $g(\mathbf{x})$ is given as,

$$g(\mathbf{x}) = \mathbf{w}^T \phi(\mathbf{x}_i) - b = 0, \; i = 1, 2, ..., n$$

The kernel function that maps the training data $\mathbf{X}$ into high-dimensional space is represented as $\phi(\mathbf{x}_i)$, for the point $i = 1, 2, ..., n$, $\mathbf{w} = [w_1, w_2, ..., w_d]^T$ is the weight vector

normal to the hyperplane, $\mathbf{w}^T$ is the transpose of the weight vector $\mathbf{w}$, and b is the bias term. The problem objective for one-class SVM is defined as,

$$\min_{\mathbf{w},b,\zeta} \left(\frac{1}{2}||\mathbf{w}||_2^2 + \frac{1}{\nu n} \sum_{i=1}^{n} \zeta_i - b \right)$$

subject to,

$$(\mathbf{w}^T \phi(\mathbf{x}_i)) \geq b - \zeta_i,\ \zeta_i \geq 0,\ \forall i = 1, 2, ..., n$$

Where $||\mathbf{w}||_2^2$ is the squared Euclidean norm, slack variables denoted by ζ_i, $i = 1, 2, ..., n$, that allow certain data points to lie outside the hyperplane (decision boundary), ν is the regularization parameter ($0 < \nu \leq 1$) that controls the balance between margin maxima and the number of anomalies allowed. The above problem objective equation is transformed into a dual problem objective to reduce the number of variables using the kernel trick to one vector. The dual problem objective is formulated as follows,

$$\max_{\alpha} \left(\sum_{i=1}^{n} \alpha_i K(\mathbf{x}_i, \mathbf{x}_j) - \sum_{i,j=1}^{n} \alpha_i \alpha_j K(\mathbf{x}_i, \mathbf{x}_j) \right)$$

subject to,

$$0 \leq \alpha_i \leq \frac{1}{\nu n}, \sum_{i=1}^{n} \alpha_i = 1,$$

here α_i are the Lagrangian multipliers for the point $i = 1, 2, ..., n$, $K(\mathbf{x}_i, \mathbf{x}_j)$ is the kernel function that transforms the training data into high-dimension for $i, j = 1, 2, ..., n$. After solving the problem objective, the new point $\mathbf{x}$ is classified either to normal or anomaly point using the decision function, and it is given as,

$$f(\mathbf{x}) = \mathbf{w}^T \phi(\mathbf{x}) - b$$

The new point $\mathbf{x}$ is considered to be an anomaly if $f(\mathbf{x}) < 0$ and normal point if $f(\mathbf{x}) \geq 0$. The hyperparameters used in the one-class SVM algorithm are kernel function ($K(\mathbf{x}_i, \mathbf{x}_j)$), gamma ($\gamma$) is relevant for RBF, and sigmoid kernel functions are used, and coeff0 (β) is an independent term for sigmoid kernels (Amer et al., 2013).

Support Vector Data Description (SVDD)

The support vector data description (SVDD) algorithm is primarily used for one-class classification and anomaly detection. The SVDD aims to find a hypersphere with the

majority of normal points inside, and the anomalies outside of it. Given n, the size of the training dataset $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$, here $\mathbf{x}_i \in \mathbb{R}^d$, $i = 1, 2, ..., n$ d is the number of features (dimensions). The objective of the algorithm is to minimize the radius R of the hypersphere that consists of data points and can be formulated as follows,

$$\min_{R, \mathbf{c}, \zeta_i} R^2 + C \sum_{i=1}^{n} \zeta_i$$

subject to (linearly separable training data),

$$||\mathbf{x}_i - \mathbf{c}||^2 \leq R^2 + \zeta_i,\ \zeta_i \geq 0,\ i = 1, 2, ..., n$$

subject to (non-linearly separable training data),

$$||\phi(\mathbf{x}_i) - \mathbf{c}||^2 \leq R^2 + \zeta_i,\ \zeta_i \geq 0,\ i = 1, 2, ..., n$$

Where $R \in \mathbb{R}$ is the radius of the hypersphere, $\mathbf{c} \in \mathbb{R}^d$ represents the center of the hypersphere, d is the number of features, regularization parameter C that acts as a balance between the volume of the hypersphere and penalizes the data points that lie outside the hypersphere, ζ_i, $i = 1, 2, ..., n$ is the non-negative slack variables that provide flexibility to include the presence of outliers. when the training data is non-linearly separable, SVDD uses kernel functions $K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)$, $\forall i, j = 1, 2, ..., n$, $\phi(\mathbf{x}_i)$ and $\phi(\mathbf{x}_j)$ is the mapping of data point $\mathbf{x}_i$, $\mathbf{x}_j$ to the high-dimensional space for i, $j = 1, 2, ..., n$, respectively. The above objective problem is solved by transforming it to a dual objective problem that includes Lagrangian multipliers α_i and is given as,

$$\max_{\alpha_i} \sum_{i=1}^{n} \alpha_i K(\mathbf{x}_i, \mathbf{x}_i) - \sum_{i,j=1}^{n} \alpha_i \alpha_j K(\mathbf{x}_i, \mathbf{x}_j)$$

subject to,

$$0 \leq \alpha_i,\ \sum_{i=1}^{n} \alpha_i = 1$$

A new point $\mathbf{x}$ is considered anomaly if $||\phi(\mathbf{x}) - \mathbf{c}||^2 > R^2$ (Chang et al., 2013).

3.2 Deep Neural Networks (DNN) - Based Methods

The deep neural network includes multiple layers, which makes it ideal for learning important features from the data. The deep neural network architecture has three layers, which are the input layer, hidden layers, and output layer. The training data with size n is represented as $X = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\} \in \mathbb{R}^d$, where d is the number of features, $\mathbf{x}_i$ is a feature vector for $i = 1, 2, ..., n$. The input layer receives the input data, and the number of neurons in this layer depends on the number of features in the input data.

The hidden layers in this thesis employ a fully connected architecture (where each neuron in the current layer is connected to every other neuron in the previous and the next layer). In DNN, there are more than two hidden layers, and each layer has a different number of neurons. A linear transformation followed by a non-linear transformation (activation function) is applied in each hidden layer and is defined as (Sewak and Sahay, 2020),

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + b^{(l)},$$

$$\mathbf{h}^{(l)} = \sigma(\mathbf{z}^{(l)}),$$

$$\mathbf{W}^{(l)} = \begin{bmatrix} w_{11}^{(l)} & w_{12}^{(l)} & \cdots & w_{1n^{(l-1)}}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n^{(l)}1}^{(l)} & w_{n^{(l)}2}^{(l)} & \cdots & w_{n^{(l)}n^{(l-1)}}^{(l)} \end{bmatrix}, \ \mathbf{h}^{(l)} = \begin{bmatrix} h_1^{(l)} \\ \vdots \\ h_{n^{(l)}}^{(l)} \end{bmatrix}$$

here the weight matrix $\mathbf{W}^{(l)}$ of size $n^{(l)} \times n^{(l-1)}$, where each element $\mathbf{w}_{ij}^{(l)}$ in the matrix denotes a weight connecting from j^{th} neuron in the previous layer $(l-1)$ to the i^{th} neuron in the current layer l is represented by $\mathbf{w}_{ij}^{(l)} = [w_{1j}^{(l)}, w_{2j}^{(l)}, ..., w_{n^{(l)}j}^{(l)}] \in \mathbb{R}^{n^{(l)}}$, $j = 1, 2, ..., n^{(l-1)}$, $n^{(l)}$ is the number of neurons in the current layer, $n^{(l-1)}$ denotes the number of neurons in the previous layer $(l-1)$, $l = 1, 2, ..., L-1$, $\mathbf{b}^{(l)} = [b_1^{(l)}, b_2^{(l)}, ..., b_{n^{(l)}}^{(l)}] \in \mathbb{R}^{n^{(l)}}$ is the bias vector , $\mathbf{z}^{(l)} = [z_1^{(l)}, z_2^{(l)}, ..., z_{n^{(l)}}^{(l)}]$, $z_i^{(l)} \in \mathbb{R}^{n^{(l)}}$ represents the pre-activated vector for the neuron i in the current layer l, $\sigma(\cdot)$ is the non-linear activation function, $\mathbf{h}^{(l)}$ and $\mathbf{h}^{(l-1)}$ denotes the output vector of size $n^{(l)}$ and $n^{(l-1)}$ of the activation from the current layer l and previous layer $l-1$, respectively.

The activation functions are responsible for introducing non-linearity in the hidden and output layers. The activation functions are applied on the linear transformation $\mathbf{z}$, which is a pre-activated vector in the layer $l = 1, 2, ..., L$. Some of the common activation functions used in this thesis are (Sewak and Sahay, 2020),

Linear Activation Function is more suitable when the output is a real value and is given as, $\sigma(\mathbf{z}) = \mathbf{z}$. Sigmoid Activation Function is applied then the output ranges from $(0, 1)$ and is defined as $\sigma(\mathbf{z}) = \frac{1}{1+e^{-\mathbf{z}}}$. For the Rectified Linear Unit (ReLU) activation function, the output vector ranges from $[0,)$ when the ReLU activation function is applied on pre-activated vector $\mathbf{z}$ and is formulated as $\text{ReLU}(\mathbf{z}) = \max(0, \mathbf{z})$. The Hyperbolic Tangent (Tanh) activation function outputs the value in the range $(-1, 1)$ and is given as $\tanh(\mathbf{z}) = \frac{e^{\mathbf{z}}-e^{-\mathbf{z}}}{e^{\mathbf{z}}+e^{-\mathbf{z}}}$.

The output layer $\hat{\mathbf{y}} = \mathbf{h}^{(L)}$ is the final layer that provides outputs based on the features learned by the hidden layers. This layer uses a distinct activation function that differs from the one used in the hidden layer activation function. The problem that the neural network is intended to solve determines the number of neurons in the output layer $n^{(L)}$. In the anomaly detection task, the output layer usually has one neuron indicating the likelihood of the input being an anomaly (Sewak and Sahay, 2020).

The optimal network parameters (weights $\mathbf{W}^{(l)}$ and bias $\mathbf{b}^{(l)}$) are found by minimizing the loss function during the training process. The neural network is trained on the training data set, and output is produced by either the probability or class labels in the case of a classification task. The loss function represents how well the predictions of the network match with the actual target labels. In this thesis, binary cross-entropy loss is employed and is given as (Ho and Wookey, 2020),

$$L(\mathbf{y}, \hat{\mathbf{y}}) = -\frac{1}{n}[\mathbf{y}^T \log(\hat{\mathbf{y}}) + (1 - \mathbf{y})^T \log(1 - \hat{\mathbf{y}})],$$

here $\hat{\mathbf{y}}$ is the predicted output probabilities, $\mathbf{y}$ is the actual target values, n is the total size of the training data, $\mathbf{y}^T$ is the transpose of the actual target values.

For the optimization of parameter $\theta = [\mathbf{W}, \mathbf{b}]^T$ that is weights and biases during the training, adaptive moment estimation (ADAM) optimization algorithm is used. ADAM optimization algorithm employs an adaptive learning rate using the first moment(mean) and second moment (average of the squared gradients) for different parameters. The parameter θ, the first moment and the second moment are $m_t = 0$ and $v_t = 0$ at each iteration $t = 0$, respectively, are initialized. The gradients of the loss function $L(\mathbf{y}, \hat{\mathbf{y}})$ with respect to the parameter θ are defined as (Nanni et al., 2021),

$$g_t = \nabla L(\mathbf{y}, \hat{\mathbf{y}}) = \frac{\partial L(\mathbf{y}, \hat{\mathbf{y}})}{\partial \theta},$$

where g_t denotes the gradient at iteration t, $\hat{\mathbf{y}}$ is the predicted output, $\mathbf{y}$ is the original output. The first (m_t) and second moment (v_t) are updated as (Nanni et al., 2021),

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t,$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2,$$

where hyperparameters of exponential decay rates for the first and second moment are represented as $\beta_1, \beta_2 \in [0, 1)$, g_t is the gradients at the iteration t, g_t^2 denotes the element-wise square of the gradients at iteration t. ADAM algorithm uses the bias-corrected updates of the moments as the above equation for the updates of moments will be very small due to the initialization of moments to zero. The bias-corrected updates for the first (m_t) and second (v_t) moment is formulated as (Nanni et al., 2021),

$$\hat{m}_t = \frac{m_t}{1 - \beta_1},$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2},$$

the paramets θ_t at iteration t is updated as follows,

$$\theta_t = \theta_{t-1} - \lambda \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon},$$

where ϵ is a small positive number to avoid division by zero error, the learning rate is denoted as λ.

The other hyperparameters that are considered in DNN for updating the network's parameters are batch size, epochs, and dropout rate. The number of samples or observations that must be processed to update the network's parameter is known as batch size. Adam optimization algorithm uses mini-batch gradient descent ($1 <$ batch size $< n$), where more than one observation but less than the entire size of the dataset (n) is used during training. The hyperparameter epochs represent the total number of times (iteration) the algorithm has to be trained on the entire dataset (Johnson et al., 2020). The dropout method is used to enhance the neural network performance and to prevent overfitting during training. The hyperparameter dropout rate defines the probability of setting the input unit to zero at random (Ba and Frey, 2013).

Deep Support Vector Data Description (Deep SVDD)

The deep SVDD algorithm combines deep neural network and SVDD algorithm for the anomaly detection task. The neural network is useful for feature representation of the data, while SVDD is a helpful one-class classification approach to identify anomalies in the data. The main objective of the algorithm is to find a minimum hypersphere that contains the feature representation of the normal data. The activation function $\phi(\mathbf{x}; \mathbf{W})$ applied on the input space $\mathbf{x} \in \mathcal{X} \subseteq \mathbb{R}^d$ to get the feature representation output space $\mathcal{F} \subseteq \mathbb{R}^p$ and is defined as, $\phi(\mathbf{x}; \mathbf{W}) : \mathcal{X} \to \mathcal{F}$, where the network weights are $\mathbf{W} = \{\mathbf{W}^1, \mathbf{W}^2, ..., \mathbf{W}^L\}$, $\mathbf{W}^{(l)}$ represents the weight for the layer $l = 1, 2, ..., L$. Given the training dataset $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$, n is the size of the data and the objective of the deep SVDD is formulated as (Ruff et al., 2018),

$$\min_{R, \mathbf{W}} R^2 + \frac{1}{\nu n} \sum_{i=1}^{n} \max(0, ||\phi(\mathbf{x}; \mathbf{W}) - \mathbf{c}||^2 - R^2) + \frac{\lambda}{2} \sum_{l=1}^{L} ||\mathbf{W}^{(l)}||_F^2 \,,$$

here $R > 0$ is the radius of the hypersphere, $\mathbf{c} \in \mathcal{F}$ represents the center of the hypersphere, and a penalty is applied to the points that lie outside the hypersphere that is if the distance of the points from the center is greater than the radius ($||\phi(\mathbf{x}; \mathbf{W}) - \mathbf{c}||^2$), $\nu \in (0, 1]$ denotes the hyperparameter that acts as a balance between allowing certain points to lie outside the sphere and the volume of the hypersphere, the weight decay regularizer parameter $\lambda > 0$ is the last term with network weights $\mathbf{W}$ and $|| \cdot ||_F$ is the Frobenius norm (Euclidean norm). The network weights are learned from minimizing the objective function such that normal points lie close to the center whereas anomalies lie outside the sphere. The anomaly score s for a test point $\mathbf{x} \in \mathcal{X}$ is given as,

$$s(\mathbf{x}) = ||\phi(\mathbf{x}; \mathbf{W}^*) - \mathbf{c}||^2,$$

where $\mathbf{W}^*$ is the learned weight parameter of the network, $||\phi(\mathbf{x}; \mathbf{W}^*) - \mathbf{c}||^2$ is the distance between the center of the sphere $\mathbf{c}$ to the point. If the distance of the test point $\mathbf{x}$ is greater than the radius R^2, then the point is considered as an anomaly (Ruff et al., 2018).

Autoencoder

Autoencoder is a type of neural network that uses the input data to compress into lower-dimension (Encoder), and the compressed data is used to reconstruct the input

data (Decoder). Autoencoders for anomaly detection learn the compressed representation of data to identify patterns in the data and then use reconstruction errors to identify anomalies, where anomalies are more difficult to reconstruct than normal data points. This thesis uses a fully connected autoencoder, which consists of the encoder, and decoder, where the encoder and decoder are similar to deep neural network architecture. Given the training data set of d-features with n sample size as, $X = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\} \in \mathbb{R}^d$ and the encoding function is given as (Chen et al., 2018),

$$\mathbf{h}_i = f(\mathbf{x}_i) = \sigma(\mathbf{W}_1\mathbf{x}_i + \mathbf{b_1}),$$

where $\mathbf{h}_i \in \mathbb{R}^m$ represents a latent representation (lower-dimensional) vector of size m ($m < d$) for $i = 1, 2, ..., n$, d-features input vector of the i^{th} sample is denoted as $\mathbf{x}_i$, $\sigma(\cdot)$ is the activation function that has been used in the deep neural architecture for introducing non-linearity in the network, $\mathbf{W}_1 \in \mathbb{R}^{m \times d}$ represents the weight matrix of the encoder with the size of $m \times d$, and $\mathbf{b_1} \in \mathbb{R}^m$ is the bias vector of the encoder of size m. The original input $\mathbf{x}_i$ is reconstructed from the latent representation $\mathbf{h}_i$ and the decoding function is defined as (Chen et al., 2018),

$$\hat{\mathbf{x}}_i = g(\mathbf{h}_i) = \sigma(\mathbf{W}_2\mathbf{h}_i + \mathbf{b_2}),$$

where $\hat{\mathbf{x}}_i$ is the reconstructed output for i^{th} data point, $i = 1, 2, ..., n$, n is the size of the sample, $\mathbf{W}_2$ is the weight matrix for the decoder of the size $d \times m$, $\mathbf{b_2}$ is the bias vector of the decoder of size d, $\sigma(\cdot)$ is the same activation function used in the encoder. The weight parameters are updated to minimize the overall loss function (average reconstruction error), and the overall loss function is formulated as, $\mathcal{L}(X, \hat{X}) = \frac{1}{n}\sum_{i=1}^{n} L(\mathbf{x}_i, \hat{\mathbf{x}}_i)$, where the training data of n sample size is $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\} \in \mathbb{R}^d$, d is the number of features, and the reconstructed output of size n is $\hat{\mathbf{X}} = \{\hat{\mathbf{x}}_1, \hat{\mathbf{x}}_2, ..., \hat{\mathbf{x}}_n\} \in \mathbb{R}^d$, $L(\mathbf{x}_i, \hat{\mathbf{x}}_i)$ represents the loss function (reconstruction error) for each data point $\mathbf{x}_i$ for all $i = 1, 2, ..., n$. The loss function used in this thesis for the autoencoder is the binary cross entropy loss. The autoencoder learns to encode and decode the training data using the Adam optimization algorithm (Chen et al., 2018).

3.3 Generative Adversarial Learning Method

Anomaly Detection with Generative Adversarial Networks (AnoGAN)

Generative Adversarial Networks (GANs) are made up of two neural networks, which are a generator and a discriminator. They compete against each other in a way that one's success is the other's failure, which is called an adversarial process. The generator aims to produce synthetic data (fake data) from random noise, and the discriminator tries to distinguish between normal data and synthetic data. The generator and discriminator are trained simultaneously, where the generator produces synthetic data that the discriminator cannot distinguish from the real data, and the discriminator becomes better at identifying real data from the synthetic data (Han et al., 2021) the training dataset of sample size n from the unknown distribution $p_{data}(\mathbf{x})$, $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\} \in \mathbb{R}^d$, where d is the number of features (dimensions). The input to the generator $G(\mathbf{z})$ is the random noise vector $\mathbf{z}$ from the distribution $p_{\mathbf{z}}(\mathbf{z})$, which is used to generate synthetic data that is similar to the distribution $p_{data}(\mathbf{x})$ of the real data. The discriminator $D(\mathbf{z})$ takes the input as the real data $\mathbf{x}$ or the synthetic data $G(\mathbf{z})$ produced by the generator and outputs the probability that the input is real or synthetic data. The generator G wants to minimize, and the discriminator wants to maximize the value function $V(D, G)$ which is defined as follows (Schlegl et al., 2017),

$$\min_G \max_D V(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{data}(\mathbf{x})}[\log(D(\mathbf{x}))] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})}[\log(1 - D(G(\mathbf{x})))].$$

The generator is trained to minimize the value function $V(D, G)$ to fool the discriminator, while simultaneously discriminator is trained to maximize the value function by distinguishing the real and synthetic data samples. A variant of GAN called AnoGAN is used to identify anomalies in the data. In AnoGAN, the real data $\mathbf{x}$ is mapped to the low-dimension representation known as latent space $\mathbf{z}$ such that the generated output data $G(\mathbf{z}) = \mathbf{z} \rightarrow \mathbf{x}$ is close to the $\mathbf{x}$. The mapping of real data to latent space consists of residual loss and the discriminator loss function, which is given as follows (Schlegl et al., 2017),

$$L_R(\mathbf{z}) = \sum_{i=1}^{n} |\mathbf{x}_i - G(\mathbf{z})_i|,$$

$$L_D(\mathbf{z}) = \sum_{j=1}^{m} |f(\mathbf{x})_j - f(G(\mathbf{z}))_j|,$$

the residual loss $L_R(\mathbf{z})$ determines the similarity between the real data $\mathbf{x}_i$ and the generated data $G(\mathbf{z})_i$ for the feature $i = 1, 2, ..., n$, n is the sample size, $f(\mathbf{x})_j$ and $f(G(\mathbf{z}))_j$ are the j-th feature vectors of real and generated data points extracted from the intermediated layer of the discriminator. The latent space $\mathbf{z}$ is optimized by backpropagation and the combined loss function $L(\mathbf{z})$ for mapping the real data to latent space is defined as (Schlegl et al., 2017),

$$L(\mathbf{z}) = (1 - \lambda)L_R(\mathbf{z}) + \lambda L_D(\mathbf{z}),$$

the trade-off between residual and discriminator loss is represented by the weighting factor λ. The anomaly score $s(\mathbf{x})$ for test data $\mathbf{x}$ is computed using the optimized latent vector $\mathbf{z}^*$ in the combined loss function and is formulated as (Kim et al., 2021),

$$s(\mathbf{x}) = (1 - \lambda)L_R(\mathbf{z}^*) + \lambda L_D(\mathbf{z}^*).$$

If the anomaly score $s(\mathbf{x})$ is high, then the test data $\mathbf{x}$ is likely to be an anomaly, whereas $s(\mathbf{x})$ is low, then the test data $\mathbf{x}$ is considered to be normal data (Kim et al., 2021).

3.4 Critical Difference (CD) plot

When comparing several algorithms across different datasets, a critical difference (CD) plot is a common graphical representation used in statistical analysis. The x-axis in the CD plot represents the multiple algorithms being compared.

The Friedman test is a non-parametric test that does not assume any specific distribution for the data, which is used to identify any significant differences among different algorithms. The null hypothesis (H_0) states that there are no significant differences in the distributions of algorithms being compared, while the alternate hypothesis (H_1) is at least one of the algorithms is significantly different from the others. Each dataset in an algorithm is ranked based on its performance. The average rank is calculated $\bar{R}j$ for each algorithm across all datasets and is defined as (Liu and Xu, 2022),

$$\bar{R}j = \frac{1}{n}\sum_{i=1}^{n} R_{ij},$$

where R_{ij} denotes the rank of j^{th} algorithm on the i^{th} dataset, for $i = 1, 2, ..., n$, n represents the number of dataset , $j = 1, 2, ..., k$, k is the number of algorithms being

compared. Test statistics is a number derived from the data used in hypothesis testing to determine whether to accept or reject the null hypothesis (H_0). The Friedman test statistic χ_F^2 is given as (Liu and Xu, 2022),

$$\chi_F^2 = \frac{12n}{k(k+1)} \left(\sum_{j=1}^{k} \bar{R}j^2 - \frac{k(k+1)^2}{4} \right).$$

The test statistic follows chi-square distribution (χ^2) with a degree of freedom $k-1$, where k is the number of algorithms. The significance level is the probability of accepting H_1 (rejecting H_0) when H_0 is true, which is $P(H_1|H_0)= \alpha$. The significance level is denoted by α. The chi-square distribution table is used to determine the critical value at the significance level ($\alpha = 0.5$) for the chi-square distribution. If χ_F^2, the test statistic is greater than the critical value, we reject the null hypothesis. If the test statistic χ_F^2 is lesser than the critical value, we fail to reject the null hypothesis. The Nemenyi post-hoc test is used to compare algorithms pairwise if the null hypothesis is rejected, which states that at least one algorithm differs significantly from the others. The null hypothesis (H_0) for the Nemenyi test is that the mean ranks for the two algorithms being compared do not differ significantly, whereas the alternate hypothesis (H_1) is that at least the mean ranks for one pair of algorithms differ significantly. The critical difference for the Nemenyi test is formulated as (Štěpánek et al., 2022),

$$CD = q_\alpha \sqrt{\frac{k(k+1)}{6n}},$$

where q_α denotes the studentized critical value at the significance level α, k is the number of algorithms, n is the number of dataset. If the rank difference calculated for the two algorithms is greater than the CD value, the null hypothesis is rejected. We fail to reject the null hypothesis if the rank difference calculated for the two algorithms is lesser than the CD value (Štěpánek et al., 2022).

3.5 Evaluation Metric

Area Under the Receiver Operating Characteristic Curve (AUC)

A graphical way of representing the performance of a binary classification algorithm is known as the receiver operating characteristic (ROC) curve. An anomaly detection algorithm is considered as a binary classifier, as the normal points are labeled as "0",

and anomaly points are labeled as "1". The x-axis of the ROC curve displays the false positive rate (FPR), while the y-axis represents the true positive rate (TPR) at various threshold levels. A true positive rate is the probability of actual positives or anomalous points that are correctly identified by the model. A false negative rate is the probability of actual positive (anomalies) that are incorrectly classified as negatives or normal points. The TPR and FPR are given as,

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}},$$

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}},$$

where TP is the number of actual anomalous points correctly identified by the model, FP denotes the number of actual normal points that are incorrectly classified as anomalies, TN represents the number of actual normal points is correctly predicted by the model, FN denotes the number of actual anomalous points are incorrectly classified as the normal points. The area under the ROC curve (AUC) measures the ability of the model to distinguish between the normal and anomaly points. The AUC score is calculated as follows,

$$\text{AUC} = \sum_{i=1}^{n-1} \left(\frac{\text{TPR}_{i+1} + \text{TPR}_i}{2} \right) \left(\text{FPR}_{i+1} - \text{FPR}_i \right),$$

here, the number of threshold values is denoted by n, and $(\text{FPR}_i, \text{TPR}_i)$ are the points on the ROC curve. The range of AUC value is 0 to 1. If the AUC is greater than 0.5 and less than and equal to 1 ($0.5 < \text{AUC} \leq 1$), it defines the model performs better than the random guessing. If the AUC score is equal to 0.5 ($\text{AUC} = 0.5$), then the model is equivalent to random guessing. If the AUC score is equal to 1, then the model can perfectly classify between the normal and anomaly points (Bouckaert, 2006).

4 Approach

In our research, shallow methods and deep neural network methods are used to tune hyperparameters in low and high-dimensional tabular data using models from the PyOD (Python Outlier Detection) library (Zhao et al., 2019). Shallow methods are beneficial because they capture simpler anomalies, train low-dimensional data, and have fewer hyperparameters. These models are appropriate for initial hyperparameter exploration because they are computationally efficient and easier to train. Whereas deep neural

network methods can capture complex data patterns, they are better suited for training high-dimensional data. They contain more hyperparameters to tune, which makes the models suitable for capturing complex patterns within the data. PyOD (Zhao et al., 2019) is an open-source Python framework that is highly appropriate for anomaly detection. PyOD includes more than 20 algorithms covering shallow and neural network architecture models. The ease of usage of PyOD is essential for fast prototyping and analysis of various models. PyOD has scalable algorithms appropriate for high-dimensional tabular data, which is necessary for our research focusing on complex data patterns.

Hyperparameter tuning is employed using the FLAML library (Wang et al., 2021) to optimize the model's performance. The choice of this library is because of the tradeoff between the computational cost and model error through the search process. As hyperparameter tuning is important for optimizing the model performance, but it can be computationally expensive when dealing with large datasets or complex models such as DNN. FLAML addresses this challenge through efficient search strategies that minimize the time and resources required to find the optimal hyperparameters. At the same time, it also ensures that the tuned hyperparameters achieve improved performance. The hyperparameter search space is carefully defined, which allows for comprehensive exploration while maintaining computational efficiency. FLAML is integrated with PyOD to automate and tune the hyperparameters for unsupervised anomaly detection models. Parallel processing was employed, where multiple datasets are trained at a time for each hyperparameter configuration to avoid longer training times. Each hyperparameter configuration is evaluated using a test dataset with labels, which enables measuring the model's performance using evaluation metrics such as AUC score.

5 Implementation

In this section, we detail the process and methodologies utilized to enhance the performance of the anomaly detection models through automated hyperparameter tuning. We also provide a comprehensive analysis of hyperparameter tuning results, which helps in understanding the practical steps taken and findings achieved in our research.

5.1 Experimental Setup

5.1.1 Datasets

In our research, we employed 498 open-source tabular datasets from Kaggle (kag) for hyperparameter tuning of anomaly detection methods. These datasets are from diverse domains and represent a wide variety of real-world scenarios. Two datasets are analyzed in detail to examine the statistics; for example, the wine quality data (Cortez et al., 2009) and the heart disease dataset (Teboul, 2022). The dataset is divided into training and test datasets, with 100% of the anomalies being used in the test dataset, an equal proportion of anomalies and normal samples being used in the test set, and the remaining normal samples being used in the training set (Refer to Table 1). This distribution ensures that the test dataset has a complete representation of anomalies for the evaluation. The wine quality data consists of 12 features that represent the chemical properties of the wine, such as acidity, residual sugar, density, pH, and several other features (Cortez et al., 2009), and the labels in the test dataset are binary values with "0" considered as normal wine and "1" considered as anomalies that are different from normal wine.

Table 1: Training and Testing Datasets Samples

Datasets	No.of Features		Total	No. of Anomalies
Wine Quality Data	12	Training Dataset	3429	0
		Test Dataset	2938	1469 (100%)
Heart Disease Dataset	21	Training Dataset	101396	0
		Test Dataset	86910	43455 (100%)

The heart disease dataset contains 21 features such as age, sex, cholesterol level, blood sugar level and others (Teboul, 2022) that describe the various medical features of the patients and the labels in the test dataset represent "0" for patients with no heart disease and "1" for patients diagnosed with heart disease. The training set is utilized for both model training and hyperparameter tuning, whereas the test set is reserved for evaluating the model performance for each hyperparameter configuration. This way, it ensures the evaluation metrics accurately represent the generalization capability of the model. For the purpose of evaluation, the test dataset includes labels. However, the anomaly detection model training is performed in an unsupervised learning way. The test dataset with labels is only used to measure the performance of the anomaly detection model after the training. After tuning the hyperparameters, we utilized 47 benchmark

datasets from ADBench (Han et al., 2022). ADBench datasets allow us to benchmark tuned hyperparameters against default hyperparameters, where these comparisons help to determine if tuning impacts the model's performance.

5.1.2 Experimental Framework

We ran eight algorithms, both shallow methods and deep neural network methods, on 170 datasets and 498 datasets, depending on the file size. The 170 datasets are a subset of the full 498 datasets and were selected based on the file size lower than 0.1 MB. The small datasets were employed for the initial hyperparameter tuning, allowing for faster iteration and resource-efficient testing to find the optimal hyperparameter values. This reduces the computational overhead and optimization steps. After obtaining optimal hyperparameters from training small-run datasets, the tuning process is applied to the full dataset for comprehensive evaluation. The model's hyperparameters are automatically tuned using the FLAML library, which efficiently explores the hyperparameter space defined to find the best-tuned hyperparameter. In this research, we conducted 10,000 iterations of hyperparameter configurations (trials) with FLAML. We used a time budget of 7 days for small-run datasets and 14 days for full-run datasets for each algorithm. This procedure guarantees a thorough exploration of hyperparameter space within the available computational resources and time budget. We implemented parallel processing such that ten datasets were trained concurrently for each hyperparameter configuration to reduce the overall tuning time. After training each dataset, the algorithm is evaluated on the test dataset to obtain AUC scores. The average AUC score is calculated across datasets for each trial. The hyperparameter configuration that maximized the average AUC score is considered the best-tuned hyperparameter. The performance of the algorithms for the best-tuned hyperparameters is compared against the default hyperparameters on the ADBench datasets, which aims to emphasize the improvements achieved through the hyperparameter tuning.

5.1.3 Hyperparameter configuration and Ranges

Our research uses the machine learning algorithms discussed in Section 3, and each algorithm is tuned with hyperparameters to optimize the model's performance. The hyperparameter ranges chosen for this research for shallow algorithms are shown in Table 2. For KNN, we tune the hyperparameter for $n\_neighbors$ to determine the number

of neighbors to include to form a cluster, *method* for calculating anomaly score (mean, median, and largest), and the parameter for Minkowski distance (p). In the Isolation

Table 2: Hyperparameters of the Shallow Algorithms

Algorithm	Hyperparameters	Ranges
KNN	$n\_neighbors$	[1, 1000]
	$method$	{mean, median, largest}
	p	[1, 6]
IFOR	$n\_estimators$	[1, 1000]
	$max\_samples$	[0.1, 1.0]
	$max\_features$	[0.1, 1.0]
	$bootstrap$	{True, False}
LOF	$n\_neighbors$	[1, 1000]
	$algorithm$	{auto, ball_tree, kd_tree, brute}
	$leaf\_size$	[10, 50]
	$metric$	{minkowski, manhattan, euclidean}
	p	[1, 6]
OCSVM	$kernel$	{linear, rbf, sigmoid}
	nu	[0.01, 1.0]
	$gamma$	[0.01, 10]
	$coef0$	[-1.0, 1.0]
CBLOF	$n\_clusters$	[1, 100]
	$alpha$	[0.1, 0.9]
	$beta$	[1.00001, 100.0]
	$use\_weights$	{True, False}

Forest (IFOR) algorithm, the hyperparameters considered are $n\_estimator$ represents the number of decision trees, $max\_samples$ is the number of samples to draw from the training dataset to build an isolation tree, $max\_features$ denotes the maximum number of features to consider for splitting the node of the tree and *bootstrap* is a categorical variable (True, False) to allow selecting samples from the dataset with the replacement. Five hyperparameters chosen to tune the local outlier factor (LOF) algorithm are $n\_neighbors$ is the number of neighbors, $algorithm$ is the method used to compute nearest neighbors, which is a categorical value consisting of auto, ball_tree, Kd_tree, and brute, $leaf\_size$ is used only if the ball_tree or Kd_tree is used to compute the nearest neighbors, $metric$ represents distance function which can be minkowski, manhattan, or euclidean, p ranges from 1 to 100 is the parameter for Minkowski distance.

The hyperparameters considered for clustering-based local outlier factor (CBLOF) are $n\_clusters$, $alpha$, and $beta$, $use\_weights$. $n\_clusters$ represents the number of clusters

to form, ranging from 1 to 100. *alpha* and *beta* are used to classify small and large clusters, ranging from 0.1 to 0.9 and 1.0 to 100.0, respectively. *use_weights* takes a categorical value, which is either true or false. If *use_weight* is true, the cluster size is considered as a weight when calculating the anomaly score. Four hyperparameters are chosen for the one-class support vector machine (OCSVM) algorithm: the kernel method, such as linear, rbf, and sigmoid, is used for the algorithm (*kernel*), *nu* to control the balance between maximizing the margin and minimizing the number of anomalies, *gamma* from $1e^{-2}$ to $1e^{1}$ is mainly used as the coefficient for rbf and sigmoid kernel, $coef0$ used as an independent term for sigmoid kernel function.

Table 3: Hyperparameters of the Deep Neural Network Algorithms

Algorithm	Hyperparameters	Ranges
Deep SVDD	*use_ae*	{True, False}
	hidden_neurons	[64, 32], [128, 64, 32], [10, 10], [20, 20],[20, 10, 3], [20, 10, 5],[30, 20, 10, 3],[20, 10, 3, 10, 20]
	hidden_activation	{linear, relu, tanh, sigmoid, softmax}
	output_activation	{relu, tanh, sigmoid, linear, softmax}
	optimizer	{adam, sgd}
	epochs	[50, 200]
	batch_size	[16, 128]
	dropout_rate	[0.05, 1]
	l2_regularizer	[0.0001, 0.5]
	preprocessing	{True, False}
Autoencoder	*latent_size*	[1, 100]
	layers	[1, 100]
	activation	{linear, relu, tanh, sigmoid, softmax}
	decoder_activation	{linear, relu, tanh, sigmoid, softmax}
	encoder_activation	{linear, relu, tanh, sigmoid, softmax}
	output_activation	{linear, relu, tanh, sigmoid, softmax}
	loss	{binary_crossentropy, mean_squared_error}
	epochs	[50, 200]
	batch_size	[16, 128]
	encoder_dropout	[0.05, 0.5]
	decoder_dropout	[0.05, 0.5]
	learning_rate	[1e-3, 1e-1]
	regularizer	[0.0001, 0.5]
	regularisation	{l2}
	evaluation_metric	[1, 6]

Deep Support Vector Data Description (DeepSvdd) considers the following hyperparameters and the ranges are shown in Table 3: *use_ae* is the autoencoder type of the

DeepSvdd, which takes either a true or false categorical value. *hidden_neurons* denotes the number of neurons needed for each hidden layer. When *use_ae* is set to true, the hidden neurons are reversed. *hidden_activation* and *output_activation* are the activation functions used for the hidden layers and output layer, respectively. The activation functions used are linear, relu, tanh, sigmoid, and softmax. *epochs* from 50 to 200 represents the number of iterations for the training, *batch_size* (range: 16 - 128) is the number of samples to be used for a gradient update, *dropout_rate* denotes the fraction of neurons to be dropped across all the layers. *l2_regularizer* is the regularization strength parameter for the L2 regularizer, which avoids overfitting across all the layers. *dropout_rate* and *l2_regularizer* ranges from 0.0 to 1.0. If *preprocessing* is set to True, the standardization is applied to the training dataset. Autoencoder includes the following hyperparameters, which are *latent_size* represents the number of dimensions in the compressed representation of the training data, and *layers* is the number of layers in the encoder and decoder. *encoder_activation* and *decoder_activation* are the activation functions (same as DeepSvdd) in the encoder and decoder layer, respectively. *loss* is used to train the autoencoder, either binary_crossentropy or mean_squared_error. *encoder_dropout* and *decoder_dropout* are the fraction of neurons to be dropped for the encoder and decoder layer, ranging from 0.0 to 0.5. *learning_rate* (range: $1e^{-3}$ - $1e^{-1}$) is to control the step size during optimization, *regularizer* is the parameter for *regularization* which is L2 regularizer, and *evaluation_metric* is the power of the distance metric to calculate reconstruction error.

Table 4: Hyperparameters of the Adversarial Learning Algorithms

Algorithm	Hyperparameters	Ranges
AnoGAN	*activation_hidden*	{linear, relu, tanh, sigmoid, softmax}
	output_activation	{linear, relu, tanh, sigmoid, softmax}
	latent_dim_G	[1, 100]
	G_layers	[10, 10], [20, 20], [20, 10, 3], [20, 10, 5], [30, 20, 10, 3], [20, 10, 3, 10, 20], [64, 32], [128, 64, 32]
	D_layers	[10, 10], [20, 20], [20, 10, 3], [20, 10, 5], [30, 20, 10, 3], [20, 10, 3, 10, 20], [64, 32], [128, 64, 32]
	epochs	[100, 500]
	batch_size	[16, 128]
	dropout_rate	[0.05, 0.99]
	learning_rate	[1e-5, 1e-2]

Nine hyperparameters (Table 4) were selected for anomaly detection with generative adversarial networks (AnoGAN), which are *activation_hidden*, *output_activation*, *latent_dim*,

G_layers, *D_layers*, are the number of layers for generator and discriminator, respectively, *epochs*, *dropout_rate*, and *learning_rate*.

5.2 Experimental Results

5.2.1 Variability Analysis of AUC scores across Iterations

In our research, a maximum of 10,000 different hyperparameter configurations (trial) are recommended by the FLAML for each algorithm. The AUC scores for each hyperparameter trial are recorded for both small and full datasets. The variability analysis of AUC scores across iterations provides insights into each algorithm's consistency when trying different hyperparameter configurations. The standard deviation of the AUC score is used to find the variation in the algorithm's performance. When there is a high standard deviation, it indicates the algorithm's performance varies significantly with different hyperparameter configurations, implying that the algorithms are highly sensitive to the hyperparameter settings. In contrast, a low standard deviation means the algorithm performs consistently, irrespective of the changes in the hyperparameter settings. Figure 1 shows the standard deviation of AUC scores for algorithms across iterations based on small-run and full-run datasets. The x-axis represents the algorithms, and the y-axis represents the standard deviation of AUC scores across iterations. The blue line is the standard deviation of AUC scores across iterations obtained from tuning small datasets, and the orange line represents the standard deviation of AUC scores across iterations obtained from tuning the full-run datasets. As full-run datasets have 498 datasets compared to the small-run datasets, which are 170, averaging the full datasets results in a smaller standard deviation than the small-run datasets.

We observe that DeepSvdd has a high standard deviation for both small (0.06) and full-run optimization (0.05), which suggests that this algorithm is sensitive to the hyperparameter settings. This is followed by Autoencoder, and AnoGAN, which also have a high standard deviation, indicating that neural network-based algorithms are sensitive to hyperparameter tuning. Therefore, neural-network-based methods need the right hyperparameters for optimal performance. The IFOR and CBLOF algorithm shows low standard deviation for both small and full-run datasets (close to zero), implying this algorithm is less sensitive to hyperparameter tuning and shows a stable performance across the iterations for different hyperparameter configurations. KNN shows a higher standard deviation than AnoGAN in small-run optimization, suggesting that KNN also

exhibits high sensitivity to hyperparameter tuning. LOF and OCSVM show moderate standard deviations in both small and full-run datasets, which implies they are somewhat sensitive to different hyperparameter settings. However, they are more consistent compared to the neural network-based algorithms.

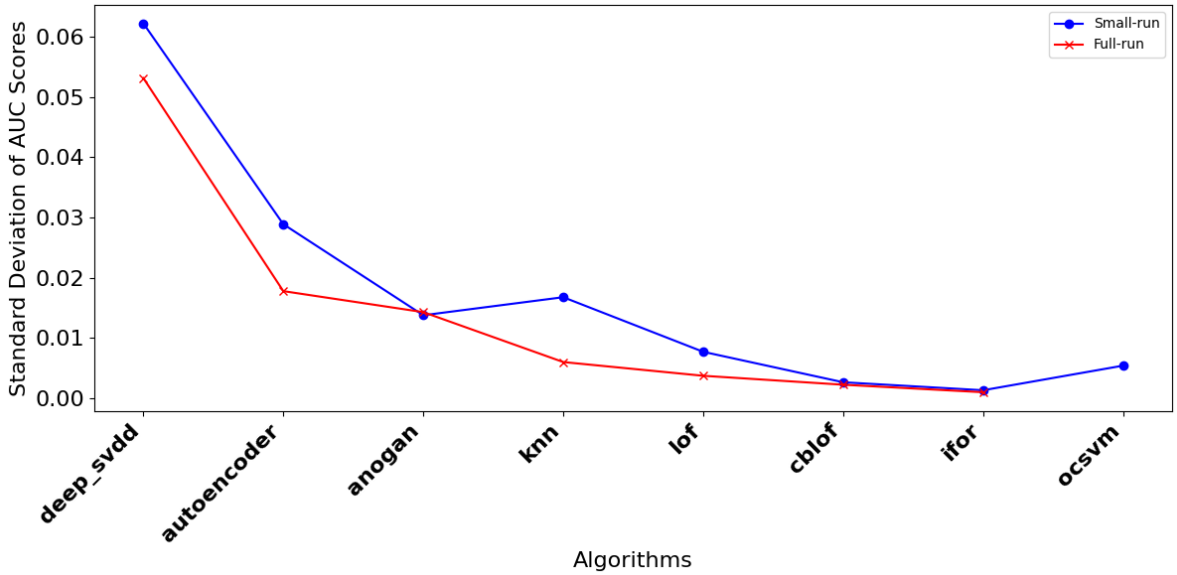

Figure 1: Variability of Algorithms across iterations evaluated on small-run (blue line) and full-run (red line) datasets (excluded OCSVM of full-run due to high computation time)

5.2.2 Tuned Hyperparameter Setting for Algorithms

Our research compared the hyperparameter-tuned values and the average AUC score for the tuned hyperparameters on the test dataset between the small-run and the full-run datasets. The FLAML library was used to tune the hyperparameters using 170 small-run datasets and 498 full-run datasets. The set of hyperparameters with the maximum average AUC score is stored for the training dataset as the optimal hyperparameters. The default values for all the algorithms are considered from PyOD (Zhao et al., 2019) which is taken from the original papers, which are shown in Tables 5 - 12. Figure 2 shows the average AUC score of different algorithms, which illustrates their performance under optimal hyperparameters on both small-run and full-run datasets. A comparison is made between the average AUC score of similar datasets from both small-run and full-run. The x-axis shows the different algorithms, each represented by two bars, and the y-axis represents the average AUC score. The blue bar represents the average AUC

scores obtained from small-run optimization, whereas the red bar indicates the average AUC scores obtained from tuning on the full-run datasets. From Table 5, for KNN, the optimal *n_neighbors* was 1, and *method* was largest for both small and full datasets, implying the closest neighbor is chosen to be effective, and the method remains consistent irrespective of the dataset size. Meanwhile, p differed with values of 5 for small datasets and 1 for full datasets. The resulting tuned values of the average AUC scores for KNN on the test dataset were 0.76 for small and full run. The optimal hyperparameters from Table 6, for LOF tuned, were 5 and 8 for *n_neighbors*. The nearest neighbor algorithm 'ball_tree' and the distance *metric* was Manhattan proved effective for both small and full-run datasets. The $leaf\_size$ tuned was 13 for small datasets and 10 for full datasets. These tuned values increased the AUC score by 1% in the small datasets (0.75) compared to the AUC score of full-run (0.74). The difference in AUC scores indicates that LOF is sensitive to hyperparameter changes. The difference in the number of neighbors may cause the sensitivity since this affects the local density estimation and, in turn, the performance.

Table 5: Hyperparameter Setting for Default and Tuned values in KNN

Hyperparameters	Default (Zhao et al., 2019)	Tuned (Small)	Tuned (Full)
n_neighbors	5	1	1
method	largest	largest	largest
p	2	5	1

The best hyperparameter for IFOR varied significantly between small and full-run optimization (Refer to Table 7). The small-run optimization for this algorithm is $n\_estimators$ at 356 and $max\_features$ at 0.97. The full-run optimization is *n_estimator* at 230 and $max\_features$ at 0.83. These differences in a higher number of estimators and features for small runs signify the need for a more complex model to capture anomalies. In contrast, reducing the number of estimators and features for full runs may result in less computational resources while maintaining the model's performance. For both optimizations, the tuned values for *max_samples* are 1.0, and *bootstrap* is false, which tells it is an optimal value for IFOR, irrespective of their dataset sizes. The AUC score for both the small and full runs is 0.71. The optimal hyperparameter for CBLOF is shown in Table 8, which requires more clusters for full datasets ($n\_clusters = 99$) compared to small datasets ($n\_clusters = 87$) to partition the data effectively. The *alpha* parameter was slightly lesser for small-run (0.71) compared to full-run datasets (0.9). The importance of cluster size parameter *beta* is slightly higher for small-runs (1.48) than for

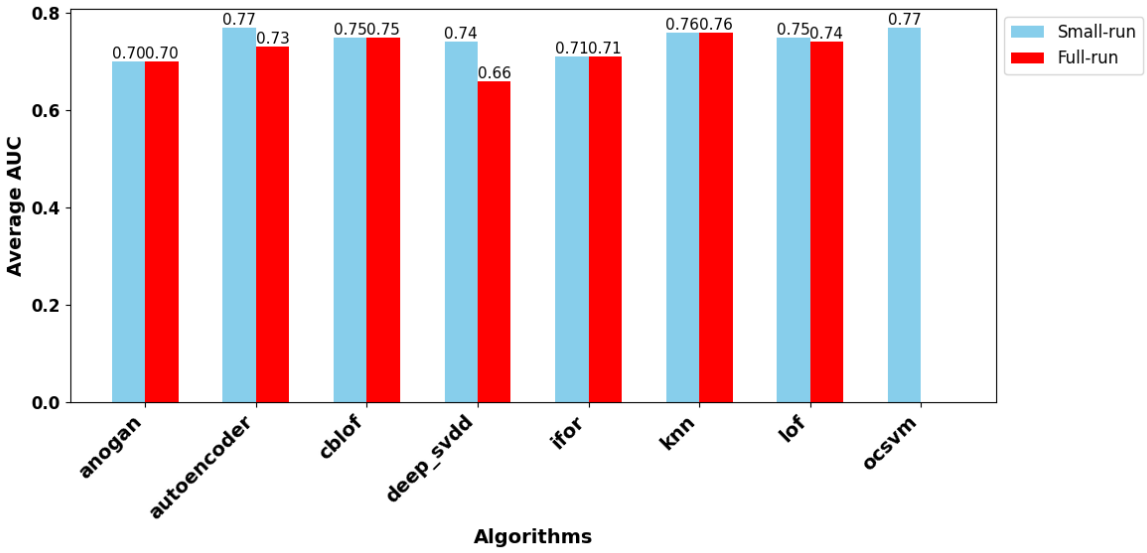

Figure 2: Comparison of AUC scores of tuned hyperparameters obtained from training small-run (blue bar) and full-run (red bar) datasets

full-runs (1.44). The cluster size was not used as weights in the anomaly score as the *use_weights* parameter was False for both datasets. CBLOF optimal values obtained from both small and full datasets performed similarly with an average AUC score of 0.75. The default and optimal hyperparameters obtained from the small dataset for OCSVM are shown in Table 9. For OCSVM, the computational time on the full datasets takes longer, so the tuned results of the OCSVM obtained from the full dataset are incomplete and, therefore, excluded from this research. The optimal *kernel* function obtained is RBF, similar to the default value. The *nu* (0.11) has slightly decreased compared to the default values (0.5). The optimal *gamma* kernel coefficient is 1.57, and for the default, it is $1/nfeatures$ (auto). The independent term used only for the sigmoid kernel function is -0.15. The resulting tuned values obtained from training small datasets have an AUC score of 0.77 on the test datasets.

Table 6: Hyperparameter Setting for Default and Tuned values in LOF

Hyperparameters	Default (Zhao et al., 2019)	Tuned (Small)	Tuned (Full)
n_neighbors	20	5	8
algorithm	auto (ball_tree)	ball_tree	ball_tree
leaf_size	30	13	10
metric	Minkowski	Manhattan	Manhattan
p	2	1	1

Table 7: Hyperparameter Setting for Default and Tuned values in Isolation Forest

Hyperparameters	Default (Zhao et al., 2019)	Tuned (Small)	Tuned (Full)
$n\_estimators$	100	356	230
$max\_samples$	auto	1.0	1.0
$max\_features$	1.0	0.97	0.83
bootstrap	False	False	False

Table 8: Hyperparameter Setting for Default and Tuned values in CBLOF

Hyperparameters	Default (Zhao et al., 2019)	Tuned (Small)	Tuned (Full)
$n\_clusters$	8	87	99
alpha	0.9	0.71	0.9
beta	5	1.48	1.44
$use\_weights$	False	False	False

For DeepSvdd in Table 10, the optimal hyperparameter hidden neurons were set to [64, 32] obtained from the small-run. In contrast, the configuration changed to a more complex structure for the full runs with [20, 10, 3] to capture the intricate pattern. The activation function for the hidden layers *hidden_activation* was 'Relu' for both optimizations. However, the output activation function (*output_activation*) differed, 'linear' for small-runs and 'sigmoid' for the full-runs. The 'Adam' *optimizer* is found to be optimal for small-run and full-run datasets. The number of *epochs* for small datasets was 51, and for full-run optimization, it was 109, suggesting that the full-run needs more iterations to train the model. Furthermore, *preprocessing* was set to False for small datasets but 'True' for full datasets, indicating the difference in the data preprocessing process. The AUC score performance for Deep SVDD showed a significant increase in small datasets (0.74) compared to full datasets (0.66), highlighting the model's capacity to capture patterns and anomalies in less complex datasets effectively. Table 11 shows the tuned values of the hyperparameters from the small and full-run optimization for AnoGAN. The full-run optimization required more generator layers ([128, 64, 32]) to handle its increased size and complexity than the small-run([30, 20, 10, 3]). The discriminator layers for both optimizations were the same, with [20, 10, 5]. The optimal generator for the full run has a larger $latent\_dim\_G$ of 99, whereas $latent\_dim\_G$ is 16 for the small run. The full run has fewer epochs (100) and batch size (100) compared to the small run, which has epochs of 167 and batch size of 127. A slightly higher $learning\_rate$ of 0.0008 was obtained for full-run optimization than the small-run of 0.0005. These tuned values resulted in an AUC score of 0.70 for the small and

Table 9: Hyperparameter Setting for Default and Tuned values in OCSVM

Hyperparameters	Default (Zhao et al., 2019)	Tuned (Small)
kernel	rbf	rbf
nu	0.5	0.11
gamma	auto	1.57
$coef0$	0.0	-0.15

Table 10: Hyperparameter Setting for Default and Tuned values in Deep SVDD

Hyperparameters	Default (Zhao et al., 2019)	Tuned (Small)	Tuned (Full)
use_ae	False	False	True
hidden_neurons	[64,32]	[64, 32]	[20, 10, 3]
hidden_activation	Relu	Relu	Relu
output_activation	Sigmoid	Linear	Sigmoid
optimizer	Adam	Adam	Adam
epochs	100	51	109
batch_size	32	25	91
dropout_rate	0.2	0.05	0.05
l2_regularizer	0.1	0.0008	0.007
preprocessing	True	False	True

full datasets. The tuned values of hyperparameters obtained from training small and full runs for Autoencoder are shown in Table 12. To handle larger and more complex datasets, the latent size was increased to 88 for the full run from 35 for the small run. The hidden layer was simple, with just one layer for the full datasets and three hidden layers for the small datasets. The *output_activation* function was 'linear' for both datasets. However, the *encoder_activation* was 'tanh' for small datasets and 'sigmoid' for full datasets. The number of training *epochs* was lowered from 163 for small datasets to 52 for full datasets. Similarly, the *regularizer* for L2 regularization was reduced from 0.0004 for small datasets to 0.0002 for full datasets. The resulting hyperparameter tuning values yielded an AUC score of 0.77 (small-run) and 0.71 (full-run). From Figure 2, it is evident that the hyperparameter tuning suggested by Flaml improved when tuned values obtained from a small-run optimization than with the full-run optimization. This implies that full-run datasets have more complex patterns than small-run datasets, so it is difficult for full-run optimization to achieve the same AUC score as small-run optimization. It also indicates that the models do not overfit the specific training and test dataset characteristics. However, the true performance of the algorithms with the tuned values can only be confirmed by evaluating how well they perform on a new, unseen dataset.

Table 11: Hyperparameter Setting for Default and Tuned values in AnoGAN

Hyperparameters	Default (Zhao et al., 2019)	Tuned (Small)	Tuned (Full)
activation_hidden	tanh	Sigmoid	Softmax
output_activation	Relu	tanh	tanh
latent_dim_G	2	16	99
G_layers	[20,10,3,10,20]	[30,20,10,3]	[128,64,32]
D_layers	[20,10,5]	[20,10,5]	[20,10,5]
epochs	500	167	100
batch_size	32	127	100
dropout_rate	0.2	0.07	0.05
learning_rate	0.001	0.0005	0.0008

Table 12: Hyperparameter Setting for Default and Tuned values in Autoencoder

Hyperparameters	Default (Zhao et al., 2019)	Tuned (Small)	Tuned (Full)
latent_size	32	35	88
layers	3	3	1
decoder_activation	Relu	Relu	Softmax
encoder_activation	Relu	tanh	Sigmoid
output_activation	Sigmoid	Linear	Linear
loss	MSE	MSE	MSE
epochs	100	163	52
batch_size	32	41	48
encoder_dropout	0.2	0.05	0.37
decoder_dropout	0.2	0.05	0.05
learning_rate	0.001	0.0001	0.017
regularizer	0.1	0.0004	0.0002
regularisation	l2	l2	l2
evaluation_metric	2	2	1

5.2.3 Critical Difference (CD) Plot Analysis of the Algorithm Rankings

The algorithms' performance on the test dataset alone cannot guarantee the generalization capability of the tuned hyperparameters, as the test datasets are related to the datasets used for the training. Additionally, to evaluate the generalization ability of the algorithms with the tuned hyperparameters, we assessed their performance on ADBench datasets, which differ from those used during training. Figure 3 illustrates the CD plot of the algorithm's rankings on the ADBench datasets using the default hyperparameter setting. The average ranks for the algorithms are plotted along the x-axis. The horizontal line represents that the algorithms are not statistically different in performance. KNN and LOF have performed best on average across all datasets, with the

lowest average rank of 2.5 and 3.4, respectively. AnoGAN performed worst on average among all algorithms, with the highest rank of 6.8. CBLOF and IFOR have performed similarly on average across the datasets, with a rank of 3.8. DeepSvdd and Autoencoder have also performed worst on average, ranking at 5.4 and 5.8. It is evident that most algorithms are horizontally connected, indicating algorithms are not statistically different from other algorithms. However, KNN performs significantly differently from OCSVM, DeepSvdd, Autoencoder, and AnoGAN. Further, DeepSvdd, Autoencoder, and AnoGAN are significantly different from KNN, LOF, CBLOF, and IFOR.

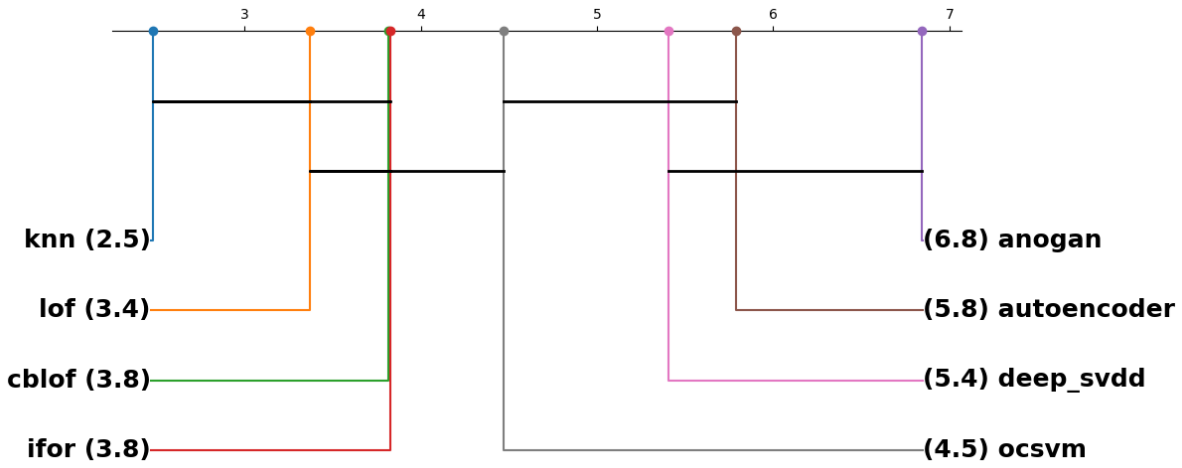

Figure 3: Algorithm Ranking based on the Default Hyperparameter Setting

The algorithm ranking on the ADBench datasets based on the tuned hyperparameter setting from the small-run datasets is shown in Figure 4. In the tuned hyperparameter, KNN performed best among all algorithms across the datasets, with an average ranking of 3. Like the default setting, CBLOF and IFOR performance is similar on average, ranking 3.8, followed by LOF and Autoencoder with an average rank of 3.9 and 4.4. OCSVM, DeepSvdd, and AnoGAN perform worst on average across all datasets, with an average ranking of 5.1, 5.3, and 6.7, respectively. The CD plot of the tuned hyperparameter setting (small-run) also suggests that most algorithms are not statistically different from others. KNN is significantly different from OCSVM, DeepSvdd, and AnoGAN. On the other hand, AnoGAN is significantly different from all algorithms except DeepSvdd. Figure 5 displays the CD plot of the algorithm rankings under a tuned hyperparameter setting obtained from the full run. The first four best-ranked algorithms in performance are KNN (2.6), IFOR (3.3), LOF (3.4), and CBLOF (3.6). OCSVM is excluded from the testing on ADBench datasets due to the high computa-

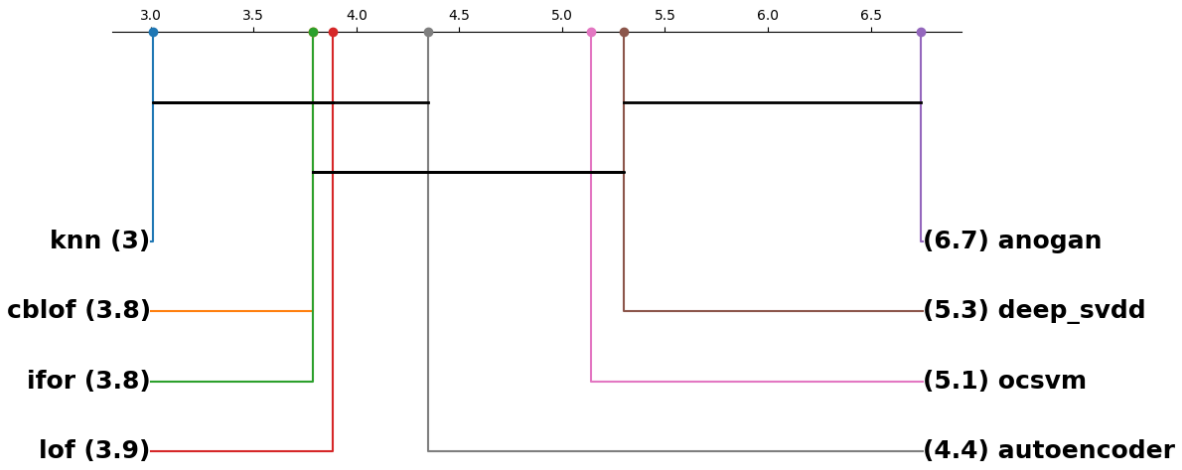

Figure 4: Algorithm Ranking based on the Tuned Hyperparameter Setting obtained from the small-run Datasets

tional time for tuning. ANoGAN is the worst-performing algorithm across all datasets, with an average rank of 5.9. Autoencoder (4.4) and DeepSvdd (4.9) are the second and third poorly performing algorithms across all datasets. KNN is significantly different in performance from Autoencoder, DeepSvdd, and ANoGAN. DeepSVDD is significantly different from the shallow algorithms (KNN, IFOR, LOF, and CBLOF). AnoGAN significantly differs in performance with all algorithms except DeepSvdd. By comparing the small-run and full-run tuned algorithms, the full-run tuned algorithms have better rankings across ADBench datasets.

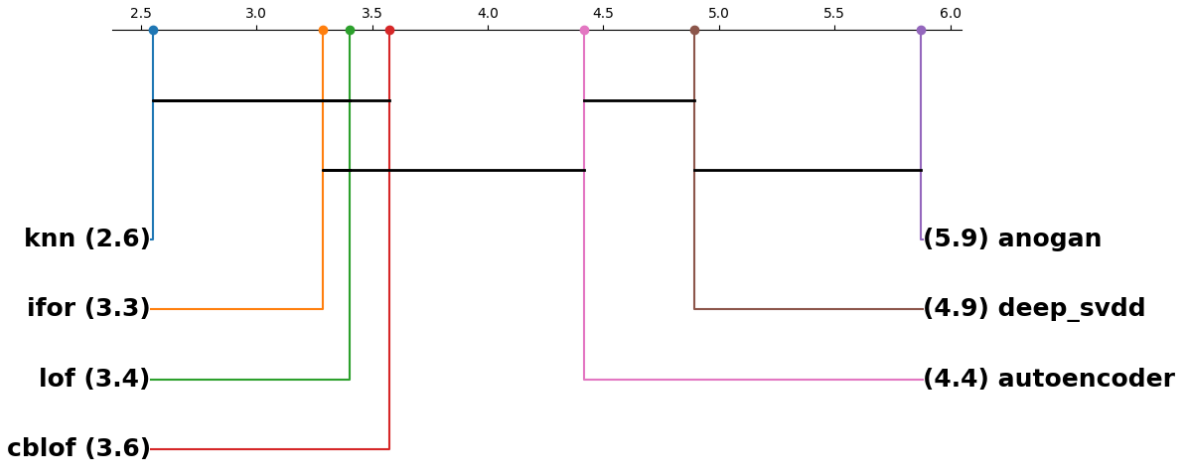

Figure 5: Algorithm Ranking based on the Tuned Hyperparameter Setting obtained from the full-run Datasets

Hence, in both default and tuned settings, the deep neural methods such as Autoencoder, DeepSvdd, and ANoGAN, except OCSVM, have the worst rankings compared to the shallow methods such as KNN, IFOR, LOF, and CBLOF. The CD plot shows that the algorithms KNN and AnoGAN show the most significant difference in performance ranking, where KNN performs best on average, and AnoGAN performs the worst on average across all the datasets under default and tuned hyperparameter settings.

5.2.4 Algorithm's Performance Comparison and RunTime Analysis

Figure 6 shows the comparison of the algorithm's performance on ADBench datasets based on three different hyperparameter settings: default hyperparameters (pink), tuned hyperparameters obtained from the small-run datasets (purple), and tuned hyperparameters obtained from the full-run datasets (blue). The algorithms are plotted on the x-axis, and the average AUC scores on the ADBench datasets for the three different hyperparameter settings are plotted on the y-axis. The IFOR, CBLOF, Autoencoder, DeepSvdd, and AnoGAN, using the optimal hyperparameters on the small-run datasets, yield an improvement in AUC scores of 3%, 3%, 15%, 9%, and 1%, respectively, on ADBench datasets when compared to the default hyperparameter setting. Similarly, the IFOR, CBLOF, Autoencoder, and DeepSvdd algorithms, using the optimal hyperparameters obtained from the full-run datasets, improve over the default hyperparameter setting in AUC scores of 3%, 3%, 13%, and 6%, respectively. It is observed that tuned hyperparameters from both small and full runs did not impact the KNN algorithm's performance over the default hyperparameter setting, which has an AUC score of 0.85. The tuned LOF, on average, did not outperform the default hyperparameter setting, where the tuned LOF shows a decrease in the AUC score of 1% compared to the default setting. Similarly, tuned ocsvm showed a decrease of 4% in the AUC score compared to the default hyperparameter setting. The decrease in the performance of tuned ocsvm could be due to the inadequate exploration of hyperparameters, as the polynomial kernel function was excluded because of its high computational time. Autoencoder and deep_svdd benefit the most from hyperparameter tuning, whereas KNN benefits the least or no improvement from tuning. AnoGAN's performance on ADBench datasets is slightly better than random guessing under all three hyperparameter settings. It is observed that AnoGAN is quite challenging to tune as it has not captured the pattern of the data effectively, which resulted in poor performance than other algorithms.

Interestingly, the algorithm's performance after tuning indicates how the algorithm's performance benefits in practice with a reasonable amount of hyperparameter tuning.

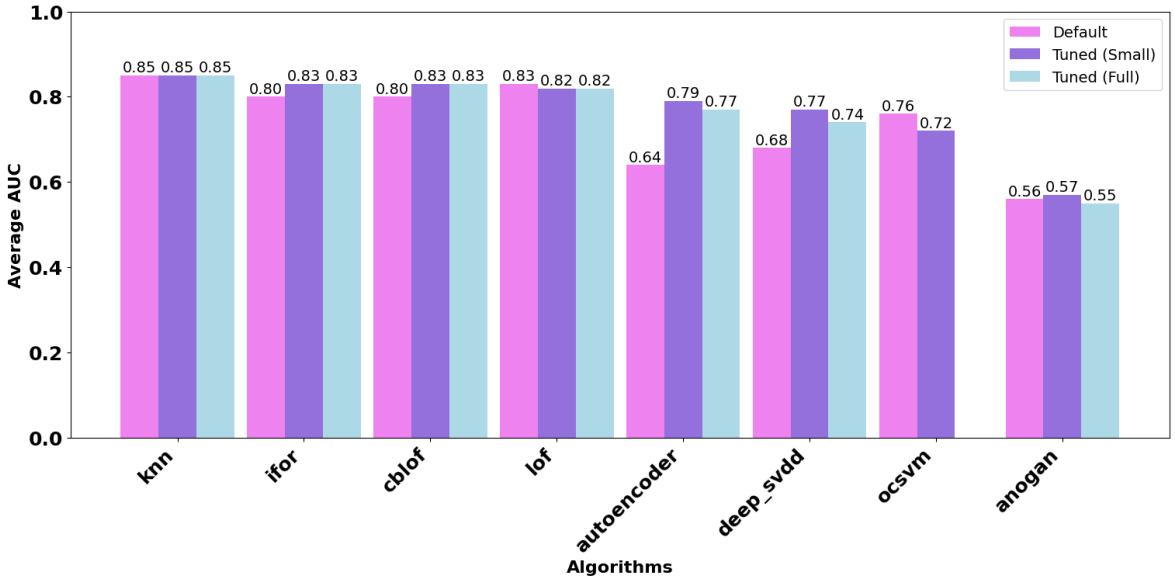

Figure 6: Comparison of AUC scores of algorithms on ADBench datasets under different hyperparameter settings: Default hyperparameter (Pink), Tuned values from Small-run optimization (Purple), Tuned values from Full-run optimization (Blue)

Figure 7 illustrates the log average execution time in seconds for each algorithm across three different hyperparameter settings. The algorithms are shown on the x-axis, where each bar represents the algorithm with the default hyperparameters (orange), tuned hyperparameters obtained from small-run (gold), and tuned hyperparameters obtained from full-run datasets (light grey). The log average time in seconds (computational cost) is represented on the y-axis. Each unit increase in the logarithmic scale on the y-axis is tenfold(10) increase in time. This makes it possible to compactly represent the broad range of time values, which makes it easier to compare the algorithms that have different computation time values. Most algorithms tuned have the same or less computation time as the default hyperparameters. But there are some exceptions; small run optimized hyperparameters have high computation time in KNN, IFOR, CBLOF, and autoencoder compared to the default hyperparameter settings. For KNN, one reason for the increased average time is that small-run optimization uses the Minkowski distance parameter of 5, whereas the default setting uses Euclidean distance, and full-run optimization uses Manhattan distance (Refer to Table 5). OCSVM and AnoGAN are the slowest algorithms compared to others, irrespective of their hyperparameter settings,

indicating they are computationally expensive. For the IFOR and CBLOF algorithms, the default hyperparameter setting is faster, and the computational cost is cheaper than the tuned values. on the other hand, for DeepSvdd, the tuned values not only reduced the computation time but also improved the performance compared to the default setting. For LOF, the algorithm's time taken when using a tuned hyperparameter from

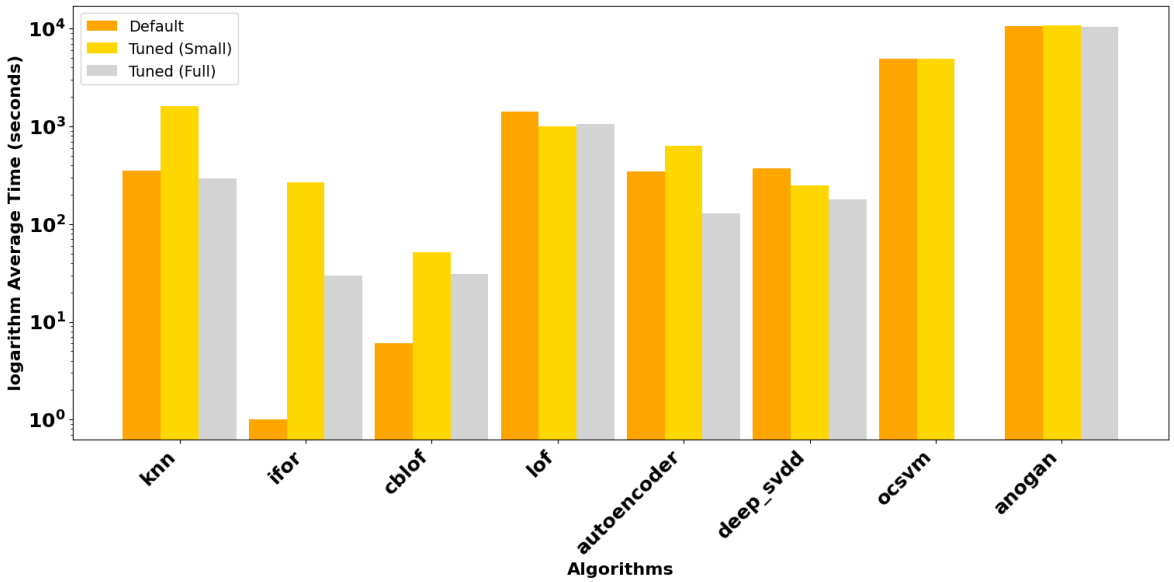

Figure 7: Comparison of the average time of algorithms on ADBench datasets

small and full runs is faster than when using the default hyperparameter setting; but this has not improved their performance. In most algorithms, the tuned values from the full-run dataset are faster than the default hyperparameter setting except for IFOR and CBLOF.

Figure 8 displays how each algorithm under different hyperparameter settings performs in regard to the average AUC score and the computation time on ADBench datasets. The plot shows an interesting relationship that compares AUC scores to the compuation time taken. The x-axis represents the average AUC and the y-axis (on a logarithmic scale) shows the average computation time in seconds taken on ADBench datasets. Three different hyperparameter configurations are shown for each algorithm: Default hyperparameters (Pink), optimal hyperparameters obtained from small run (Purple), and optimal hyperparameters obtained from full run (Blue). All hyperparameter setting for AnoGAN has very high computation time (around 10^4) and low AUC score. This implies that AnoGAN is computationally expensive and slightly better than random guessing in performance. Algorithms like KNN and LOF achieve high AUC values but

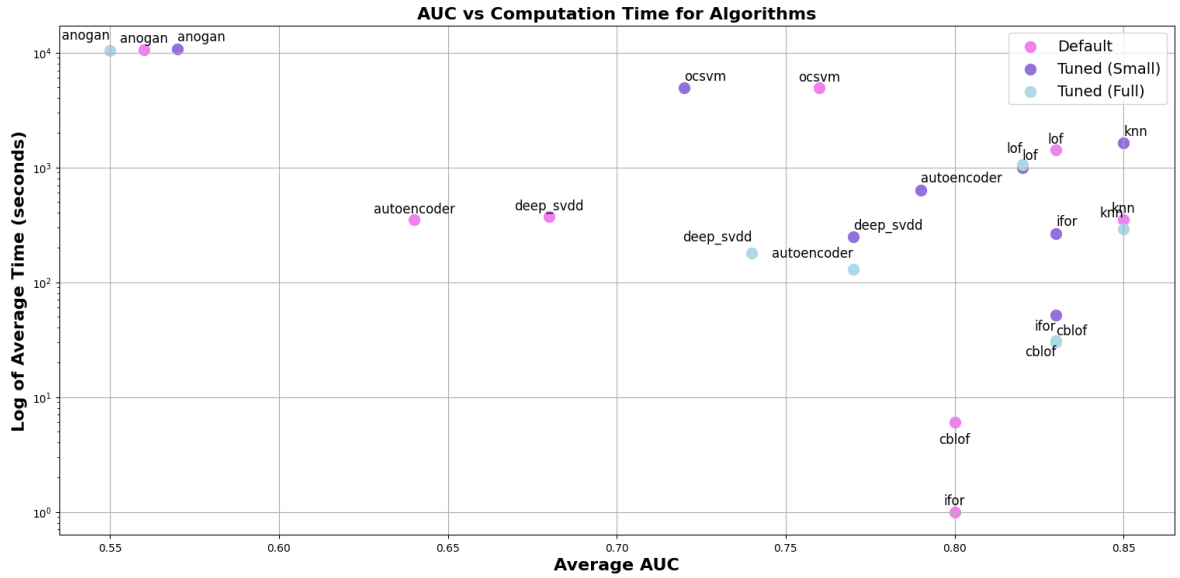

Figure 8: Average AUC Vs. Computation Time of Algorithms on ADBench Datasets

require high computation time. Algorithms such as IFOR and CBLOF tuned offer a good balance between performance and computation time. Moreover, deep learning methods like DeepSvdd and Autoencoder tuned also perform better than default. In summary, full-run optimization often improved the performance and reduced the computation time. On the other hand, if the performance is the top priority and the computational time is not a concern, small-run optimizations are recommended.

6 Conclusion and Future Work

In our research, we introduce the importance of hyperparameter tuning in the performance of anomaly detection algorithms. The hyperparameter tuning process was initially performed on a subset of 170 datasets and then extended to include the entire 498 datasets. To assess the performance and stability across different dataset sizes, eight anomaly detection algorithms were tuned on both small-run and full-run datasets. Additionally, the algorithms with optimized hyperparameters were evaluated on the ADBench datasets to assess their generalization ability on a new, unseen dataset.

FLAML recommended a maximum iteration of 10,000 hyperparameter trials, and the hyperparameter that maximized the average AUC score was selected as the optimal hyperparameter for the algorithm. The average AUC score for each trial was recorded for

both small-run and full-run datasets. Our analysis also identified the algorithms that are sensitive to the hyperparameter changes. The neural network-based algorithms showed high variability in AUC scores across the iterations, which indicates these algorithms needed the right hyperparameters for optimal performance. Meanwhile, IFOR showed more consistency in AUC scores across the iterations, showing they are less sensitive to the specific hyperparameter settings. CBLOF and LOF showed moderate variability across iterations, which implies they are somewhat sensitive to the hyperparameter setting.

The results of the critical plot provided insights into the algorithm's rankings. This plot visually represents algorithm ranks across different datasets based on the AUC scores. It is observed that KNN performed best across the ADBench datasets irrespective of whether the default hyperparameters, tuned hyperparameters from small-run datasets, or tuned hyperparameters from full-run datasets were used. On the other hand, AnoGAN performed worst on average across the ADBench datasets in all three different hyperparameter settings. These findings have also demonstrated that shallow methods perform well, whereas the optimized deep neural networks, such as DeepSvdd and Autoencoder, have also achieved high rankings across datasets compared to default settings. These findings imply the significance of hyperparameter tuning in deep neural networks, leading to performance improvements.

Most algorithms showed an improvement in AUC score after optimization compared with the default hyperparameters on ADBench datasets. DeepSvdd and Autoencoder showed high improvement, yielding 9% and 15% when using the tuned values obtained from the small-run datasets. When using tuned values obtained from the full-run datasets, DeepSvdd and Autoencoder showed an improvement of 6% and 13% compared to the default hyperparameter setting. KNN does not benefit from tuning; regardless of the three hyperparameter settings, the algorithm shows an AUC score of 0.85. Some exceptions exist where tuned LOF and OCSVM showed a decrease in AUC scores of 1% and 4% compared to the default hyperparameter. For OCSVM, the poly kernel function was not included due to its computational expense, resulting in incomplete hyperparameter search space exploration. Additionally, our study's analysis excluded the optimization of OCSVM on full-run datasets due to their high computation time. Considering the computation time for the algorithms, tuned values from the full-run dataset showed faster execution time than the default setting except for IFOR and CBLOF. OCSVM and AnoGAN are more computationally costly as they take around 10^3 and 10^4, respectively, log average time for both default and tuned hyperparameters. Our research

also highlights that the full-run optimization not only improved the performance but often also reduced the computation time. Although small-run optimization increased the performance, it led to high computation time compared to the default setting.

Finally, several key areas can be extended to the current research. In Future work, we aim to introduce contamination to the training dataset, adding anomalies to better represent real-world scenarios. In order to improve the model performance by using labeled and unlabeled data, we also intend to do hyperparameter tuning in semi-supervised learning. To assess the model performance beyond tabular data, we want to extend the research to image and time-series data. We will focus on the optimization of high-dimensional datasets. This will guarantee the optimized hyperparameters are specifically appropriate for practical applications involving complex, high-dimensional data.

Bibliography

Kaggle. `https://www.kaggle.com`.

Mennatallah Amer, Markus Goldstein, and Slim Abdennadher. Enhancing one-class support vector machines for unsupervised anomaly detection. In *Proceedings of the ACM SIGKDD workshop on outlier detection and description*, pages 8–15, 2013.

Jimmy Ba and Brendan Frey. Adaptive dropout for training deep neural networks. *Advances in Neural Information Processing Systems 26*, pages 1–9, 2013.

James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2), 2012.

R.R Bouckaert. Efficient auc learning curve calculation. In *AI 2006: Advances in Artificial Intelligence*, volume 4304, pages 181–191. Springer, Berlin, Heidelberg, 2006.

Jair Cervantes, Farid Garcia-Lamont, Lisbeth Rodríguez-Mazahua, and Asdrubal Lopez. A comprehensive survey on support vector machine classification: Applications, challenges and trends. *Neurocomputing*, 408:189–215, 2020.

Wei-Cheng Chang, Ching-Pei Lee, and Chih-Jen Lin. A revisit to support vector data description. *Dept. Comput. Sci., Nat. Taiwan Univ., Taipei, Taiwan, Tech. Rep*, 2013.

Zhaomin Chen, Chai Kiat Yeo, Bu Sung Lee, and Chiew Tong Lau. Autoencoder-based network anomaly detection. *2018 Wireless Telecommunications Symposium (WTS)*, pages 1–5, 2018.

Zhangyu Cheng, Chengming Zou, and Jianwei Dong. Outlier detection using isolation forest and local outlier factor. In *Proceedings of the conference on research in adaptive and convergent systems*, pages 161–168, 2019.

Microsoft Corporation. *Visual Studio IDE*. Microsoft Corporation, Redmond, Washington, version 16.11 edition, 2022. ISBN 978-0-12-345678-9.

Paulo Cortez, A. Cerdeira, F. Almeida, T. Matos, and J. Reis. Modeling wine preferences by data mining from physicochemical properties. `https://www.kaggle.com/datasets/yasserh/wine-quality-dataset`, 2009. Accessed on 2024-02-02.

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)*, pages 507–510. IEEE, 2015.

Mohamad Dolatshah, Ali Hadian, and Behrouz Minaei-Bidgoli. Ball*-tree: Efficient spatial indexing for constrained nearest-neighbor search in metric spaces. *arXiv preprint arXiv:1511.00628*, 2015.

Lian Duan, Lida Xu, Ying Liu, and Jun Lee. Cluster-based outlier detection. *Annals of Operations Research*, 168:151–168, 2009.

Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter. Auto-sklearn 2.0: The next generation. *arXiv preprint arXiv:2007.04074*, 24: 8, 2020.

Python Software Foundation. Python 3.10.6 documentation. `https://docs.python.org/3.10/index.html`, 2022.

Vincent Garcia, Eric Debreuve, and Michel Barlaud. Fast k nearest neighbor search using gpu. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–6. IEEE, 2008.

Markus Goldstein and Seiichi Uchida. A comparative evaluation of unsupervised anomaly detection algorithms for multivariate data. *PloS one*, 11(4):1–31, 2016.

Songqiao Han, Xiyang Hu, Hailiang Huang, Minqi Jiang, and Yue Zhao. Adbench: Anomaly detection benchmark. *Advances in Neural Information Processing Systems*, 35:32142–32159, 2022.

X. Han, X. Chen, and L.-P Liu. Gan ensemble for anomaly detection. *In Proceedings of the AAAI Conference on Artificial Intelligence*, vol 35:4090–4097, 2021.

Yaoshiang Ho and Samuel Wookey. The real-world-weight cross-entropy loss function: Modeling the costs of mislabeling. *IEEE Access*, 8:4806–4813, 2020.

Wenfeng Hou, Daiwei Li, Chao Xu, Haiqing Zhang, and Tianrui Li. An advanced k nearest neighbor classification algorithm based on kd-tree. In *2018 IEEE International Conference of Safety Produce Informatization (IICSPI)*, pages 902–905. IEEE, 2018.

John D. Hunter. Matplotlib: A 2d graphics environment. `http://matplotlib.org`, 2007. Accessed JUNE 27, 2024.

Franklin Johnson, Alvaro Valderrama, Carlos Valle, Broderick Crawford, Ricardo Soto, and Ricardo Ñanculef. Automating configuration of convolutional neural network hyperparameters using genetic algorithm. *IEEE Access*, 8:156139–156152, 2020.

D. Kim, J. Cha, S. Oh, and J. Jeong. Anogan-based anomaly filtering for intelligent edge device in smart factory. *2021 15th International Conference on Ubiquitous Information Management and Communication (IMCOM)*, pages 1–6, 2021.

Oliver Kramer. K-nearest neighbors. *Dimensionality reduction with unsupervised nearest neighbors*, pages 13–23, 2013.

Erin LeDell and Sebastien Poirier. H2o automl: Scalable automatic machine learning. In *Proceedings of the AutoML Workshop at ICML*, volume 2020. ICML San Diego, CA, USA, 2020.

J. Liu and Y. Xu. T-friedman test: A new statistical test for multiple comparison with an adjustable conservativeness measure. *Int J Comput Intell Syst 15*, pages 1–19, 2022.

Loris Nanni, Gianluca Maguolo, and Alessandra Lumini. Exploiting adam-like optimization algorithms to improve the performance of convolutional neural networks. *2018 Wireless Telecommunications Symposium (WTS)*, pages 1–5, 2021.

Alexey Natekin and Alois Knoll. Gradient boosting machines, a tutorial. *Frontiers in neurorobotics*, 7:21, 2013.

Arti Patle and Deepak Singh Chouhan. Svm kernel functions for classification. In *2013 International conference on advances in technology and engineering (ICATE)*, pages 1–9. IEEE, 2013.

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, and Michel. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Yanjun Qi. Random forest for bioinformatics. *Ensemble machine learning: Methods and applications*, pages 307–323, 2012.

Lukas Ruff, Robert A. Vandermeulen, Nico Görnitz, Lucas Deecke, Shoaib A. Siddiqui, Alexander Binder, Emmanuel Müller, and Marius Kloft. Deep one-class classification. *Proceedings of the 35th International Conference on Machine Learning*, 80, 2018.

T. Schlegl, P. Seeböck, S.M. Waldstein, U. Schmidt-Erfurth, and G Langs. Unsupervised anomaly detection with generative adversarial networks to guide marker discovery. *In Information Processing in Medical Imaging: 25th International Conference, IPMI 2017, Boone, NC, USA, June 25-30, 2017, Proceedings*, pages 146–157, 2017.

Mohit Sewak and Hemant Sahay, Sanjay K.and Rathore. An overview of deep learning architecture of deep neural networks and autoencoders. *Journal of Computational and Theoretical Nanoscience*, 17(1):182–188, 2020.

Kristina P Sinaga and Miin-Shen Yang. Unsupervised k-means clustering algorithm. *IEEE access*, 8:80716–80727, 2020.

A. Teboul. Heart disease health indicators dataset. `https://www.kaggle.com/datasets/alexteboul/heart-disease-health-indicators-dataset`, 2022. Accessed: 2024-02-02.

Chi Wang, Qingyun Wu, Markus Weimer, and Erkang Zhu. Flaml: A fast and lightweight automl library. *Proceedings of Machine Learning and Systems*, 3:434–447, 2021.

Dong Xu, Yanjun Wang, Yulong Meng, and Ziying Zhang. An improved data anomaly detection method based on isolation forest. In *2017 10th international symposium on computational intelligence and design (ISCID)*, volume 2, pages 287–291. IEEE, 2017.

Tong Yu and Hong Zhu. Hyper-parameter optimization: A review of algorithms and applications. *arXiv preprint arXiv:2003.05689*, 2020.

Yue Zhao, Zain Nasrullah, and Zheng Li. Pyod: A python toolbox for scalable outlier detection. *Journal of Machine Learning Research*, 20(96):1–7, 2019. URL `http://jmlr.org/papers/v20/19-011.html`.

Yue Zhao, Ryan Rossi, and Leman Akoglu. Automatic unsupervised outlier model selection. *Advances in Neural Information Processing Systems*, 34:4489–4502, 2021.

Lubomír Štěpánek, Filip Habarta, Ivana Malá, and Luboš Marek. A short note on post-hoc testing using random forests algorithm: Principles, asymptotic time complexity analysis, and beyond. In *2022 17th Conference on Computer Science and Intelligence Systems (FedCSIS)*, pages 489–497, 2022.

Appendix

A Additional figures

B Additional tables

Eidesstattliche Versicherung

(Affidavit)

Erappakkal Joby, Jessia — 230299

Name, Vorname
(surname, first name)

Matrikelnummer
(student ID number)

☐ Bachelorarbeit
(Bachelor's thesis)

■ Masterarbeit
(Master's thesis)

Titel
(Title)

Understanding the Impact of Automated Hyperparameter Tuning on Anomaly Detection Methods

Ich versichere hiermit an Eides statt, dass ich die vorliegende Abschlussarbeit mit dem oben genannten Titel selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

I declare in lieu of oath that I have completed the present thesis with the above-mentioned title independently and without any unauthorized assistance. I have not used any other sources or aids than the ones listed and have documented quotations and paraphrases as such. The thesis in its current or similar version has not been submitted to an auditing institution before.

Dortmund, 07.08.2024

Ort, Datum
(place, date)

J Joby

Unterschrift
(signature)

Belehrung:
Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird ggf. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin") zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Official notification:
Any person who intentionally breaches any regulation of university examination regulations relating to deception in examination performance is acting improperly. This offense can be punished with a fine of up to EUR 50,000.00. The competent administrative authority for the pursuit and prosecution of offenses of this type is the Chancellor of TU Dortmund University. In the case of multiple or other serious attempts at deception, the examinee can also be unenrolled, Section 63 (5) North Rhine-Westphalia Higher Education Act (*Hochschulgesetz, HG*).

The submission of a false affidavit will be punished with a prison sentence of up to three years or a fine.

As may be necessary, TU Dortmund University will make use of electronic plagiarism-prevention tools (e.g. the "turnitin" service) in order to monitor violations during the examination procedures.

I have taken note of the above official notification:*

Dortmund, 07.08.2024

Ort, Datum
(place, date)

J Joby

Unterschrift
(signature)

***Please be aware that solely the German version of the affidavit ("Eidesstattliche Versicherung") for the Bachelor's/ Master's thesis is the official and legally binding version.**