FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Implementation of split guiding metric for Isolation Forest
Student:	Bc. Maroš Kramár
Supervisor:	Ing. Adam Valenta
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2023/2024

Instructions

Explore the problem of Anomaly Detection in data. Describe the algorithms typically used for Anomaly Detection, focus on the tree-based approaches (Isolation Forest and its variants).
Study the optimized computational framework for Isolation Forest [1], which replaces the random selection of the split point with a gradient method that aims to maximize the separability of the data.
Get familiar with the Map/Reduce framework and open-source H2O Machine Learning Platform
Explore the possibilities of combining non-axis-parallel splits from the Extended Isolation Forest algorithm [2] with the split guiding metric.
Implement the selected algorithm into the H2O-3 Machine Learning platform.
Evaluate the implementation's anomaly detection performance, and training duration performance on various datasets, toy data, and discuss the results.

[1] Zhen Liu, Xin Liu, Jin Ma, Hui Gao, "An Optimized Computational Framework for Isolation Forest", Mathematical Problems in Engineering, vol. 2018, Article ID 2318763, 13 pages, 2018. https://doi.org/10.1155/2018/2318763.

[2] S. Hariri, M. C. Kind, R. J. Brunner, "Extended Isolation Forest", IEEE Transactions on Knowledge and Data Engineering, vol. 33, no. 4, pp. 1479-1489, 2021. https://doi.org/10.1109/TKDE.2019.2947676.

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 15 February 2023 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Implementation of split guiding metric for Isolation Forest

Bc. Maroš Kramár

Department of Applied Mathematics

Supervisor: Ing. Adam Valenta

May 4, 2023

Acknowledgements

I would like to thank my supervisor Ing. Adam Valenta for his guidance and patience during my work on the thesis. I would also like to thank the H2O.ai company for giving me the opportunity to contribute to a large open–source project. Finally, I would like to thank my family and friends for all their support during my studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on May 4, 2023 . . . . . .

Czech Technical University in Prague

Faculty of Information Technology

Citation of this thesis

Kramár, Maroš. *Implementation of split guiding metric for Isolation Forest.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Abstrakt

Detekcia anomálií je dôležitou súčasťou analýzy dát a strojového učenia so širokým spektrom využitia. V dobe, keď sa údaje spracúvajú v obrovských množstvách, môžu byť tradičné metódy výpočetne nákladné a nemusia byť dobre škálovateľné na veľkých, mnohodimenzionálnych dátových sadách. Isolation Forest je unikátny koncept, ktorý je založený na explicitnej izolácii anomálnych dátových bodov namiesto profilovania normálnych bodov. Postupom času boli objavnené niektoré obmedzenia tohto modelu a navrhnuté rozšírenia na ich odstránenie. V práci sú vybrané rozšírenia skúmané a experimentuje sa s ich kombináciou. Vybraná metóda je implementovaná do open–source platformy pre strojové učenie, H2O–3. Záver obsahuje vyhodnotenie novo pridanej implementácie, porovnanie s dvoma existujúcimi a diskusiu výsledkov.

Klíčová slova Detekcia anomálií, nesupervizované učenie, Isolation Forest, Extended Isolation Forest, Fair Cut Forest, H2O.ai

Abstract

Anomaly detection is an important part of data analysis and machine learning with many applications. In an era, where data is being processed in massive quantities, traditional methods can be computationally expensive and may not scale well on large, high–dimensional datasets. The Isolation Forest is a unique concept, which is based on explicitly isolating anomalous data points rather than profiling normal ones. Over time, some limitations of the model have been discovered and some extensions have been proposed to address them. The thesis studies selected extensions and experiments with combining their approaches. The selected method is implemented into the H2O–3 open–source machine learning platform. The added implementation is evaluated against two existing implementations and the results are discussed.

Keywords Anomaly detection, unsupervised learning, Isolation Forest, Extended Isolation Forest, Fair Cut Forest, H2O.ai

Contents

List of Figures

List of Algorithms

List of Tables

Introduction

Motivation and objectives

As a part of machine learning techniques, anomaly detection plays an important role with many applications. It is a process of identifying patterns in the data that deviate from the normal or usual behavior. Such methods have various uses from de–noising data for other data analysis purposes to preventing failures and disasters and have a great impact on our world. As the amount of data generated by modern data mining systems continues to grow, so does the need for effective and efficient anomaly detection methods.

Many traditional methods, mostly based on statistical theory or distance measures between the observations, may have limitations in terms of data size, dimensionality, distributional assumptions or other factors. The need for more efficient, faster, more accurate and easily scalable methods is driving many scientist to explore new possibilities. In 2008, a method based on a new concept of isolation was proposed, called the Isolation Forest. The method explicitly separates anomalous points instead of profiling normal ones. The structure is based on a decision tree, where at each node the data is randomly split into two partitions recursively until the points are isolated. The idea is that anomalies should be easier to separate, meaning the depth of such points in the tree is lower. Many trees are built in this way and the prediction is based on the average of all the trees. The random aspect of the algorithm is responsible for very fast training times and the tree structure allows for high parallelism and scalability.

While it has been shown that Isolation Forest is able to outperform many of the existing methods with the configuration of hyperparameters being not very dataset specific, making it a very universal anomaly detection method, it

does have some limitations. Over time, some extensions have been developed to mitigate these problems. The first limitation is that the Isolation Forest always splits the data by a single, uniformly randomly selected feature, which can affect the performance if the class of the data point depends on a relation between multiple variables. This is addressed by the Extended Isolation Forest. The second is the random selection of the split point, which may not be optimal. If the choice of the split point can be guided by some optimization criteria, the predictions may be more accurate. This idea is explored in the Optimized Computational Framework for Isolation Forest.

The first objective is to try to combine these two approaches into one method, build a prototype and test the performance. The second goal is to select a method and implement it into the open–source machine learning platform H2O–3. The method should use a split guiding metric that helps to choose a more relevant split instead of the uniform random selection of splits. The aim is to improve the results of the methods currently implemented in the library: the standard Isolation Forest and the Extended Isolation Forest. The evaluation and comparison with the previous implementations should follow with a discussion of the results. The H2O–3 is a well–known and widely used platform by many companies and individuals. The implementation of an improved anomaly detection algorithm could be a benefit for all the users of this platform or an incentive for new users to join.

Organization

The thesis is divided into five chapters. The first one provides a basic insight into anomaly detection in general, the data formats, types of errors, and types of anomaly detection models. The second chapter presents some of the most common methods and algorithms. The third chapter discusses the Isolation Forest algorithm in detail and some of its limitations and extensions to overcome them: Extended Isolation Forest and Optimized Computational Framework for Isolation Forest. In the fourth chapter, experiments with combining the two methods and the results of the experiments are presented. Then, two other methods based on similar idea, the SCiForest and the Fair Cut Forest are studied, evaluated and compared. The last chapter describes the implementation of the selected method into the H2O–3. The implementation is then evaluated and compared to the previous implementations. Finally, some ways to further improve the implementation are discussed.

Chapter 1

Anomaly detection

1.1 Definition

Anomaly detection, in the context of data analysis, is a process of identifying anomalous data points within a given set of data. The term outlier detection is often used interchangeably. Hawkins defines an outlier as an observation, which deviates so much from the other observations as to arouse suspicions that it was generated by a different mechanism [1] (e.g. the data point could come from different statistical distribution). It can also be defined as an observation (or subset of observations) that appears to be inconsistent with the remainder of that set of data [2]. The common assumption about anomalies in a data set is that they represent only a small part of the whole.

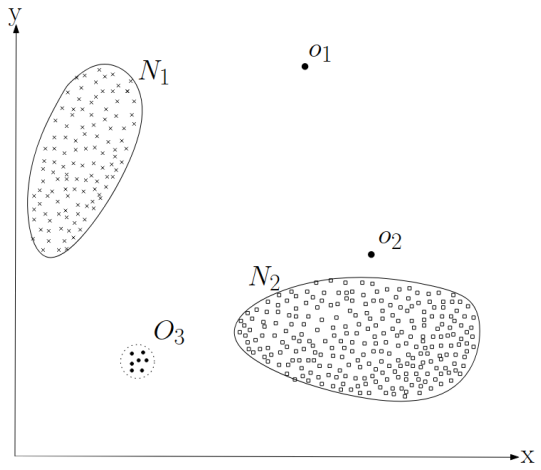


Figure 1.1: Example of outliers in 2–dimensional data [3]

There are many ways how outliers can appear in a data set. If the data is collected from a set of sensors, a failure could occur, giving the measurement a distorted value. Another factor could be human error if the data is created or handled manually. Anomaly detection has many applications in today's data analysis, some of which are: detecting fraudulent banking transactions, unauthorised access to computer systems, medical diagnosis or event detection in sensor networks [4]. This means that in systems that use anomaly detection, it is important and even critical that these methods work correctly.

Anomalies in a data set can be of different nature and can be classified into the following categories [3]:

- **Point anomalies**, also called scattered anomalies, are individual data points that are anomalous when compared to the rest of the data set.
- **Contextual anomalies** are data points that are anomalous in a particular context. For example, a sudden increase in website traffic without any reason (such as holiday).
- **Collective anomalies** are groups of data points that are anomalous when considered together, but may not be anomalous individually.

1.2 Input data and model training

Detection can be performed on a variety of data formats, including numerical data such as time series or sensor readings, categorical data, text and even multimedia (e.g. anomaly detection in medical or satellite images). The most typical data for this task is in tabular form, consisting only of numerical data, which in principle all formats mentioned above could be converted to. In data mining, this is usually referred to as a dataset. Formally, the dataset $\mathbf{X}$ can be defined as a matrix where each row represents a single instance (also known as a sample, example or data point) and each column represents an attribute, also known as a feature:

$$\mathbf{X} \in \mathbb{R}^{N,p},$$

where N is a number of data points and p is a number of features. The individual sample is a vector $x \in \mathbb{R}^p$.

$$\mathbf{X} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} = \begin{pmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,p} \\ x_{2,1} & x_{2,2} & \dots & x_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N,1} & x_{N,2} & \dots & x_{N,p} \end{pmatrix}$$

To find outliers in a dataset, we should find a model that describes the data and can distinguish between a normal and an anomalous instance. The model can be thought of as a function where the input is an instance x and the output can be one of the following [4]:

- **Binary label** – denotes the class of the data point (e.g. 0 as normal instance, 1 as anomalous).
- **Anomaly score** – quantifies the level of "outlierness" of each point. This allows the data points to be ranked in order of their outlier tendency. Anomaly scores can also be converted to binary values with thresholding, but are usually preferred because they carry more information.

The model can be built using various algorithms and the process of building a model is called training (or fitting). The input for the training is a training dataset. The goal is to construct a model that can make predictions that are as close to reality as possible. In other words, we want to minimize the error of the model. In anomaly detection we identify two types of errors [5]:

- **Masking** (type one error) occurs when anomalous data is misclassified as normal, resulting in a false negative classification. This can happen when the anomalies are subtle or difficult to distinguish from normal data, or when the algorithm is not sufficiently sensitive to detect them.
- **Swamping** (type two error) occurs when normal data is misclassified as anomalous, resulting in a false positive classification. This may be caused by the detection algorithm being too sensitive.

The question is, how does the model know which class do the points belong to, so that it can be evaluated and improved. The answer is that in most cases it doesn't because the data doesn't contain any information about what class does the data points belong to. From the view of training data (or training supervision), anomaly detection models can be divided into three categories [4]:

- **Supervised anomaly detection**, where the model uses training data that contains labels for both normal instances and outliers, which can be used for evaluation and further improvement of the model. These methods are equivalent to classification in unbalanced datasets and are designed for application–specific anomaly detection.

- **Semi–supervised anomaly detection**, where the model has labels for only part of the data. The labels can be obtained by manual expert labeling, for example.
- **Unsupervised anomaly detection** methods are the most typical in anomaly detection. The model is built without the knowledge of the data point class and the data is modeled under the assumption that only a small portion of instances in the input data are anomalous. It can often produce false positive or false negative results, so it is often used in an exploratory setting, where the discovered outliers are provided to the expert for further examination.

1.3 Novelty detection

Novelty detection is closely related to anomaly detection and sometimes the two fields can be confused. The main goal of novelty detection is to identify previously unseen patterns in the data available during training, rather than identifying outliers, that are included in the dataset. Some methods used for anomaly detection can also be suitable for novelty detection, but their performance may depend on the specific model being used. Novelty detection can often be referred to as a one–class classification, where the model is trained on the normal instances and then identifies the instances, that do not belong to that class [6].

Chapter 2

Common methods and algorithms

This thesis will mainly focus on unsupervised anomaly detection methods on tabular multi–dimensional data with continuous numerical features. There are several different approaches that can be used for this task, some of which will be presented in this chapter.

2.1 Statistical methods

Most of statistical methods have some assumptions about the data distribution, the most common being normality of the data, and may have limitations depending on the size or dimensionality of the data. Advantages are low computational cost and good interpretability due to well–established theory.

The **Z–score**, also known as standard score or z–ratio [7], is one of the simplest methods for detecting outliers. We assume a univariate normal distribution X with mean μ and variance σ^2. For each value, we calculate a difference from the mean, normalized by its standard deviation:

$$z_{score}(x, X) = \frac{x - \mu}{\sigma}.$$

In practice, the parameters of the distribution are unknown, but the μ and σ can be estimated by the sample mean $\bar{x}$ and the sample standard deviation σ. The score represents the number of standard deviations by which the value x is above or below the average. We set a threshold by which we determine whether the point is anomalous (if it surpasses this threshold) or normal. This method assumes that the data has a normal distribution, otherwise it

may not be as effective. The Z–score is sensitive to extreme values, which can bias the mean and standard deviation. The multivariate analogy uses **Mahalanobis distance** [8], which is a measure of distance that takes into account the correlation between variables:

$$d_M(x, X) = \sqrt{(x-\mu)^T \Sigma^{-1} (x-\mu)},$$

where X is a probability distribution on $\mathbb{R}^p$ with mean $\mu = (\mu_1, \mu_2, .., \mu_p)$ and positive–semidefinite covariance matrix $\Sigma \in \mathbb{R}^{p,p}$ and $x = (x_1, x_2, .., x_p)$ is a data point.

Statistical methods can also be in the form of tests, where we test a formulated hypothesis H_0 against an alternative hypothesis H_A by evaluating the test statistic specific to that test. The evaluation is based on data samples (population) and the result is a p–value. We decide the result by setting a significance level (usually 5%) and comparing it with the p–value of the test. One such example can be **Grubbs's test** [9], defined as follows:

$$H_0 : \text{There are no outliers in the data set.}$$

$$H_A : \text{There is exactly one outlier in the data set.}$$

The test statistic is:

$$G = \frac{\max_{i=1..N} |x_i - \bar{x}|}{\hat{\sigma}},$$

where x_i is i–th data point, $\bar{x}$ is sample mean and $\hat{\sigma}$ is sample standard deviation. This method, similarly to the z–score, also only works on univariate datasets with a normal distribution and can also suffers from sensitivity to extreme values. It detects only one outlier at a time, which can be removed from the dataset and the test can be repeated again.

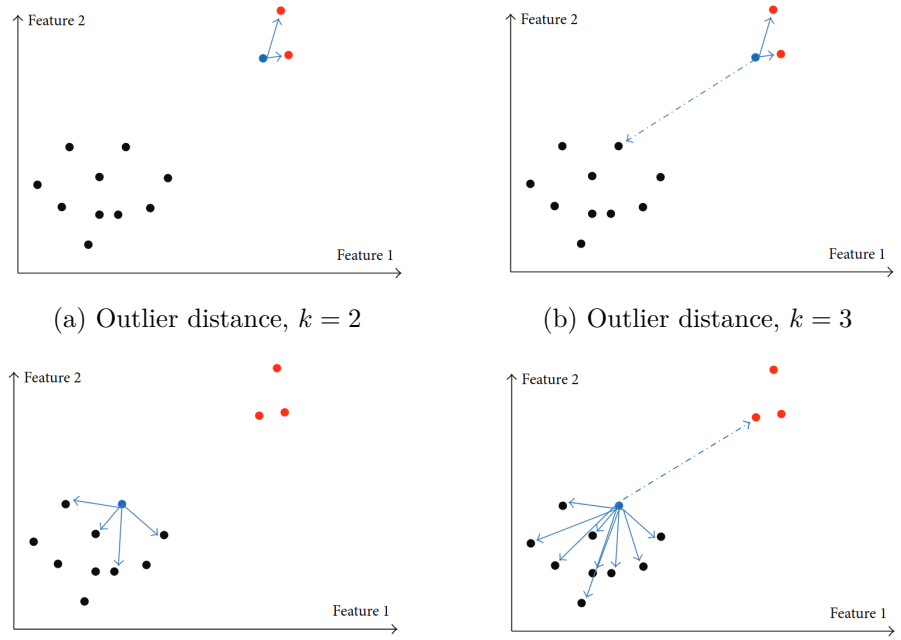
There also are other tests like **Dixon's Q test** [10], which has limits for the data size ($\leq$ 30 samples) or **Rosner's test** [11], which requires manual identification of the outliers beforehand. In the age of big data, this is not very convenient.

2.2 Distance–based methods

Another class of methods that makes no assumptions about the distribution of the data is based on the distance measures in the vector space of the data points. The idea behind measuring the distance is that points that are similar are close to each other. These methods aim to identify points that are far away from the majority of the data points, which are then considered anomalous. Various distance metrics can be used, which can be set as a hyperparameter for a given method. Some examples are: Euclidean distance, Manhattan distance or cosine similarity [12].

The most common method in this category is the **k–nearest neighbors (kNN)** algorithm [13]. Its main use is in supervised learning, but it can also be adapted for unsupervised anomaly detection by assigning each point an anomaly score, which is computed as the distance between that point and its k–th nearest neighbor. Based on the scores, we set a threshold that determines the label of the data point [15].

A drawback of this model is its sensitivity to the choice of hyperparameter k, which needs to be tuned independently for each dataset. The effect of different settings is illustrated in Figure 2.1. Another problem common to methods that use a distance measure is the curse of dimensionality [16] – meaning it is hard to find meaningful nearest neighbors. In high–dimensional data spaces, the number of possible combinations of feature values increases exponentially with the number of dimensions, making it difficult to find relationships in the data, especially when using Euclidean distance. To mitigate this effect, some kind of dimensionality reduction technique should be used.



(a) Outlier distance, $k = 2$

(b) Outlier distance, $k = 3$

(c) Normal point distance, $k = 4$

(d) Normal point distance, $k = 10$

Figure 2.1: k–nearest neighbors and the effect of different values of k [14]

Algorithm 1 k–nearest neighbors

Input: $\mathbf{X} = (x_1, x_2, .., x_N)^T$ - input data, k - number of nearest neighbors to consider, d - distance metric
Output: $Y = (y_1, y_2, .., y_N)$ - anomaly score for each $x \in \mathbf{X}$
  for i = 1,2,..,n **do**
    $x'_i \leftarrow k$–th nearest neighbor of x_i, according to distance metric d
    $Y_i \leftarrow d(x_i, x'_i)$
  end for
  return Y

2.3 Density–based methods

The idea behind density–based methods is that the regions of the vector space where normal points are located have a higher density of points, while outlier regions have a much lower density. These methods also use distance metrics, but in a more complex way.

One such method, called the **Local Outlier Factor (LOF)** [17], computes a local density given by the k–nearest neighbors for each point and compares them between neighbors. The normal data point is expected to have similar local density as its neighbors, while an anomaly is expected to have a significantly lower density than its neighbors. In other words, an anomaly is a point that is far from its neighbors in terms of density. This method returns an anomaly score for each data point, in this case called the outlier factor.

Let's define the *k–distance* of point x_i in dataset $\mathbf{X}$ as the distance between x_i and its k–th nearest neighbor x_j according to the distance function $d(x_i, x_j)$, which is a hyperparameter of this method. The set of k–nearest neighbors of data point x_i, called the *k–distance neighborhood*, denoted $N_k(x_i)$, contains every object whose distance from x_i is not greater than the *k–distance* [19]. If some points have equal distance from x_i, the size $|N_k(x_i)|$ can be greater than k. Now we define the reachability distance $rd(x_i, x_j)$ and local reachability density $lrd(x_i)$:

$$rd(x_i, x_j) = \max(k\text{–}distance(x_i), d(x_i, x_j)),$$

$$lrd(x_i) = \left(\sum_{x_j \in N_k(x_i)} \frac{rd(x_i, x_j)}{|N_k(x_i)|} \right)^{-1}.$$

Finally, we define the local outlier factor, which represents the anomaly score for each data point:

$$LOF(x_i) = \frac{\sum_{x_j \in N_k(x_i)} lrd(x_i)}{|N_k(x_i)|} * lrd(x_i)^{-1}.$$

Based on the value of $LOF(x_i)$:

- $LOF(x_i) \sim 1 \Rightarrow$ Similar density than neighbors of x_i.
- $LOF(x_i) < 1 \Rightarrow$ Higher density than neighbors of x_i, meaning x_i could be a normal point.
- $LOF(x_i) > 1 \Rightarrow$ Lower density than neighbors of x_i, meaning x_i could be an anomaly.

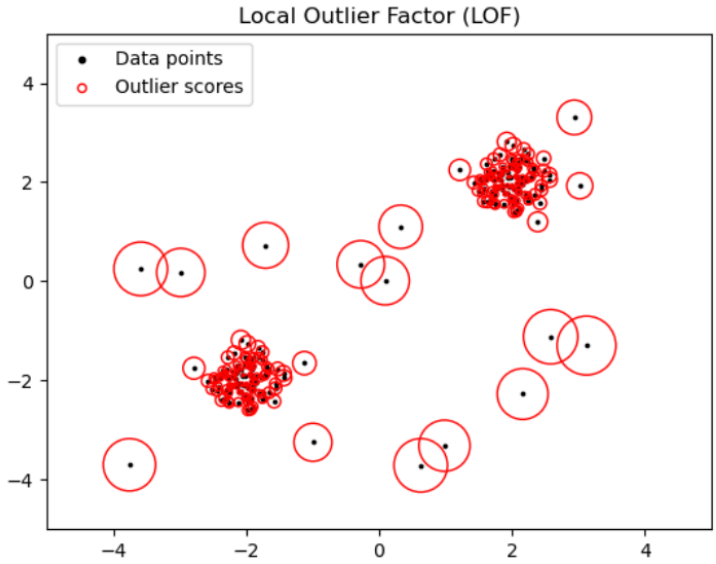


Figure 2.2: Visualization of Local Outlier Factor [18]

This method is able to detect anomalies that are located in low–density regions even if they are surrounded by normal data points and handle datasets with non–uniform distributions and different densities. Like the kNN, it is also sensitive to the choice of the hyperparameters k and the distance function d, which may require tuning.

Another widely used method that utilizes density is the Density–Based Spatial Clustering of Applications with Noise, mostly known by its abbreviation, the **DBSCAN** [20]. It is a clustering algorithm that can also be used for anomaly detection. Following is a high–level overview of the algorithm.

Algorithm 2 DBSCAN

Input: **X** – dataset, ϵ – size of the neigborhood, $minPts$ – neighborhood size noise threshold
  for each unvisited $x \in$ **X do**
    Mark x as visited
    $N_x \leftarrow$points in the ϵ-neighborhood of point x
    if $|N_x| < minPts$ **then**
      Mark x as noise
      Continue with next unvisited x
    else
      Create a new cluster C and add x and all $x' \in N_x$ to the cluster
      Recursively repeat for all $x' \in N_x$ and add to C
      Mark all $x' \in N_x$ as visited
    end if
  end for

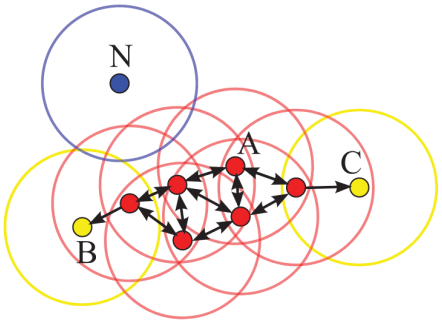


Figure 2.3: DBSCAN cluster model [21]

The Figure 2.3 illustrates one example. Parameter $minPts$ was set to 4. Marked red are core points of the cluster, being directly reachable from point A. Yellow are the edge points, belonging to the cluster but not directly reachable from A. Point N was labeled as noise.

In the case of anomaly detection, we consider outliers to be all the points x that have been marked as noise. DBSCAN may have hard time clustering data with large differences in densities between the clusters, since the combination of hyperparameters may not be appropriate for all the clusters. This algorithm is again sensitive to the choice of hyperparameters [22] and may require dimensionality reduction, as in the previously mentioned methods. There are several extensions of this algorithm, such as the hierarchical version HDBSCAN [23].

2.4 One–class support vector machine

One–class support vector machine (OCSVM) [24] allows unsupervised anomaly detection. It is based on the support vector machine (SVM) [25], which is a supervised classification method and it works as follows: suppose we have data set $\mathbf{X} = (x_1, x_2, .., x_N)^T, x \in \mathbb{R}^p$ (data point with p features) and a set of labels $Y = (y_1, y_2, .., y_N)$ for each of the data points, where $y \in \{-1, 1\}$ are the data classes. The SVM tries to construct a separating hyperplane defined as follows:

$$w^T x + b = 0,$$

where $w \in \mathbb{R}^p$ is a normal vector of the hyperplane and $b \in \mathbb{R}$ is the intercept. The goal is to find such hyperplane, that:

$$y_i(w^T x_i + b) \geq 1,$$

$$\forall i = 1..N.$$

If we assume, that the data is linearly separable, meaning that such a hyperplane exists, we try to find one that has the largest margin between the data points belonging to different classes – we search for w and b that minimize $\frac{1}{2}||w||^2$ under the above constraints. In practice, the linear separability of the data is very rare. The way to solve this problem is to allow errors ξ_i of misclassified data points and introduce new parameter C that quantifies the penalization. The modified minimization function is:

$$\frac{1}{2}||w||^2 + C\sum_{i=1}^{n} \xi_i,$$

with constraints:

$$y_i(w^T x_i + b) \geq 1 - \xi_i,$$

$$\xi_i \geq 0,$$

$$\forall i = 1..N.$$

The SVM can use Kernel trick to map the data points into a higher dimensional feature space, which helps to separate data that are not linearly separable in the original space (e.g. when data points have polynomial dependency) without actually working in the higher dimensional feature space [26].

One–class SVM assumes that all input data points belong to one class. The goal is to find a boundary between the regions of the data points and the remaining space. OCSVM uses the origin of the space as the only instance of the negative class and all the data points represent the positive class. This way, the binary SVM classification can be applied to the problem of anomaly detection [27].

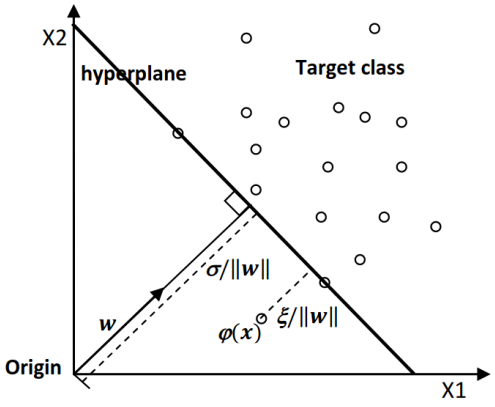


Figure 2.4: One–class SVM separating hyperplane [27]

There is also a variant called Support Vector Data Description (SVDD), which constructs a hypersphere with a minimum radius R and center a that comprises most of the data points [28]. In this case, one minimizes:

$$R^2 + C \sum_{i=1}^{n} \xi_i,$$

with constrains:

$$||x_i - a||^2 \leq R^2 + \xi_i,$$
$$\xi_i \geq 0,$$
$$\forall i = 1..N.$$

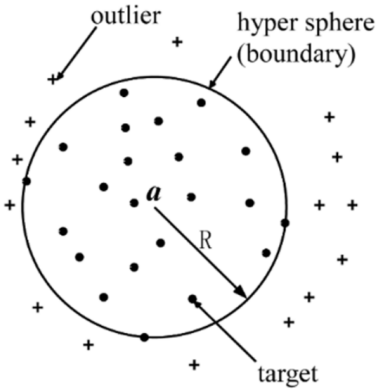


Figure 2.5: Support Vector Data Description [29]

2.5 Autoencoder

The autoencoder [30] is a model based on neural network that can be used for unsupervised learning. It consists of two main parts: an encoder and a decoder. The encoder takes an input, which is transformed to lower–dimensional representation called a latent space or code. The decoder then takes the compressed representation and reconstructs the original input with the goal of minimizing the reconstruction error. The error is computed by comparing the original input to the output of the decoder in various ways, such as mean squared error (MSE) or binary cross–entropy loss. During training, the autoencoder learns to extract the most important features of the input data and to reconstruct it from the compressed representation. It has various use–cases like data compression, dimensionality reduction, data denoising and anomaly detection.

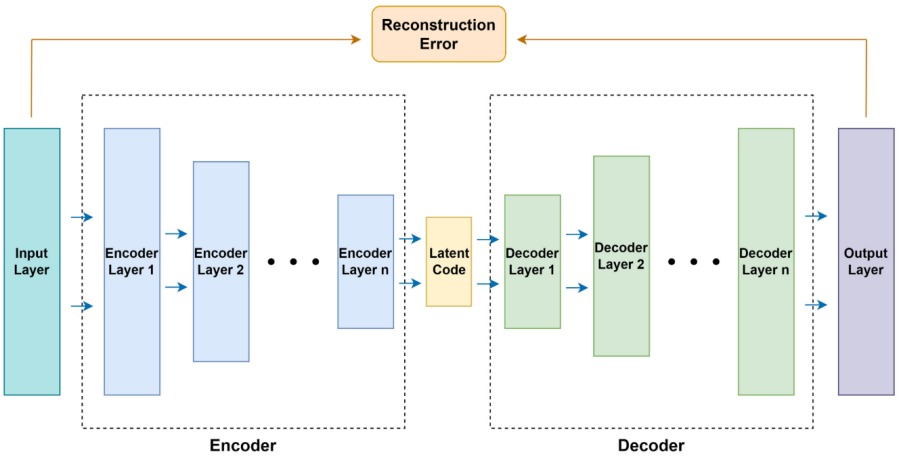


Figure 2.6: Autoencoder [31]

The process of detecting outliers works as follows. First the autoencoder is trained on the whole data set. Once the model is trained, the reconstruction error of each data point is calculated by comparing the reconstructed data point with the original input. Data points that have a high reconstruction error, i.e. they are difficult to reconstruct compared to the rest, are flagged as potential anomalies. A threshold for the errors is set and the data is labeled based on this condition [32]. The advantage of autoencoders for anomaly detection is their ability to learn complex and non–linear representations of data.

Chapter 3

Isolation forest

3.1 Definition

The Isolation Forest algorithm (also known as iForest or IF) [33], created in 2008, proposes a fundamentally different method that explicitly isolates anomalies instead of profiling normal points. The term isolation is used in the context of separating an instance from the rest of the instances. Anomalies are assumed to be "few and different" therefore they should be easier to isolate. Note that a slightly different notation is used than the one in the original paper to be consistent with the rest of this thesis.

The model is based on binary tree structure where each node is either:

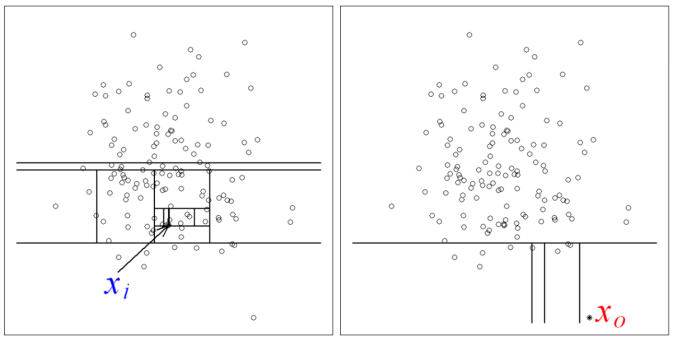
- a leaf (or external) node with no child or
- an internal node with one test and exactly two child nodes. The test consists of selected feature $\mathbf{X}_{:,q}$ and split value $s \in \mathbf{X}_{:,q}$, which splits the data points into two sets: $\mathbf{X}_l = \{x_i | x_{i,q} < s\}$ and $\mathbf{X}_r = \{x_i | x_{i,q} \geq s\}$.

The tree is constructed recursively, where each node gets the partition $\mathbf{X}' \subseteq \mathbf{X}$, $\mathbf{X}'$ being $\mathbf{X}_l$ or $\mathbf{X}_r$ from its parent, randomly chooses a feature $\mathbf{X}'_{:,q}$ and then randomly selects a split value (between the minimum and maximum values of the $\mathbf{X}'_{:,q}$). The points are again split into the two partitions and passed to child nodes. The construction can be terminated either by complete isolation of the given partition ($|\mathbf{X}'| = 1$) or by reaching a depth limit. Each external node is assigned an anomaly score according to its depth in the tree (will be discussed later). We assume that outliers require fewer splits to isolate, so the depth of such a node should be smaller compared to the rest of the points [33].

Algorithm 3 iTree

Input: $\mathbf{X}' \in \mathbb{R}^{N,p}$ – input data, e – current tree height, l – height limit
Output: an iTree

```
if e ≥ l or |X'| ≤ 1 then
    return exNode{Size ← |X'|}
else
    randomly select feature index q ∈ [1, p]
    randomly select a split point s between min and max values of X_{:,q}
    X_l ← {x_i | x_{i,q} < s}
    X_r ← {x_i | x_{i,q} ≥ s}
    return inNode{Left ← iTree(X_l, e + 1, l),
                  Right ← iTree(X_r, e + 1, l),
                  SplitAtt ← q,
                  SplitValue ← s}
end if
```



(a) Isolating x_i (b) Isolating x_o

Figure 3.1: Isolating normal point (x_i) vs outlier (x_o) [33]

3.1.1 Ensemble

As the name suggests, this algorithm constructs a model composed of multiple trees, also known as a tree ensemble. Ensemble methods in decision trees combine the results of multiple trees in order to improve their predictive performance. The idea is that many "weak learners" (simpler models) working together are more powerful than one strong model and also prevent overfitting [34]. In the case of decision trees, the typical way to retain simplicity is to limit the maximum depth and randomly selecting only a small portion of the training data for each tree with the goal of creating a forest as diverse as possible. Finally, in the prediction phase the data point is passed through all the trees to obtain a result and the prediction is an average of all the trees.

Isolation Forest uses subsampling (random selection without replacement) to select the data for each tree. The sample size, denoted as ψ, should be kept small, as it has been shown that smaller sizes produce better iTrees because the swamping and masking effects are reduced [33]. In the case of iForest, the paper suggests that 100 trees should be used, the optimal sample size for each tree should be $\psi = 256$ and the tree depth should be limited by $l = \lceil \log_2 \psi \rceil$, assuming that this is approximately the average tree height [35]. The idea is that we are only interested in points with shorter–than–average path lengths, which are more likely to be outliers. So the hyperparameters of this model are: t – the number of trees and ψ – the subsampling size.

Algorithm 4 iForest

Input: $\mathbf{X}$ – input data, t – number of trees, ψ – subsampling size
Output: a set of t *iTrees*
  Initialize *Forest*
  set height limit $l = ceiling(\log_2 \psi)$
  for $i = 1$ to t **do**
    $\mathbf{X}' \leftarrow sample(\mathbf{X}, \psi)$
    $Forest \leftarrow Forest \cup iTree(\mathbf{X}', 0, l)$
  end for
  return *Forest*

3.1.2 Predictions

The output of the prediction for the data point x is an anomaly score based on the depth of the external node where the data point x lands after being passed through the tree. The depth is equivalent to the path length from the root node to that external node, denoted as $h(x)$. The isolation tree has a limited depth – therefore the $h(x)$ doesn't correctly represent the path length, as if it were fully built. This can be compensated by approximating the value with the estimate of the unsuccessful search in the binary search tree:

$$\hat{h}(x) = h(x) + c(n), \tag{3.1}$$

where n is the count of training data points $|X'|$ in the external node and

$$c(n) = \begin{cases} 2H(n-1) - \frac{2(n-1)}{n} & \text{for } n > 2, \\ 1 & \text{for } n = 2, \\ 0 & \text{otherwise.} \end{cases} \tag{3.2}$$

The $H(.)$ is the harmonic number estimated as $ln(.) + \gamma$, where γ is the Euler–Mascheroni constant ($\gamma \approx 0.5772156649$) [36]. Using this approximation, we compute the anomaly score $\hat{h}(x)$ for each tree in the ensemble. The final score is calculated as follows:

$$s(x, N) = 2^{\frac{-E(\hat{h}(x))}{c(N)}}, \tag{3.3}$$

where $E(\hat{h}(x))$ is the average $\hat{h}(x)$ over the trees in ensemble. Based on the anomaly score, the following conclusion are can be made [33]:

- $s(x_i, N) \to 1 \Rightarrow x_i$ is an anomaly.
- $s(x_i, N) \to 0 \Rightarrow x_i$ is a normal point.
- $\forall i = 1..N : s(x_i, N) \approx 0.5 \Rightarrow$ the dataset doesn't contain any distinct anomalies.

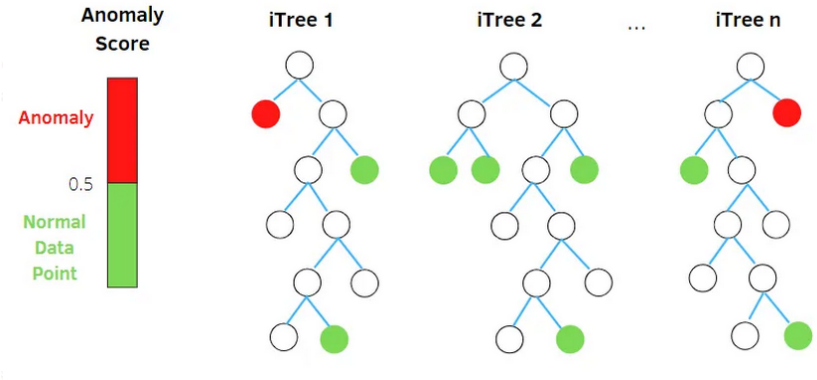


Figure 3.2: Anomaly score evaluation in iForest [37]

Algorithm 5 PathLength

Input: x – an instance, T – an iTree, e – current path length; to be initialized to zero when first called
Output: path length of x

```
if T is an external node then
    return e + c(T.size){c(.) is defined in Equation (3.2)}
end if
a ← T.splitAtt
if x_a < T.splitValue then
    return PathLength(x, T.left, e + 1)
else
    return PathLength(x, T.rigth, e + 1)
end if
```

Isolation Forest is fast, has low memory requirements, is easy to implement and can handle large datasets with high dimensionality. The algorithm is not as sensitive to the choice of hyperparameters as most of the other methods. It has proven to be one of the best all–around methods as measured by aggregated metrics across benchmarks, where it can outperform other methods on some types of datasets, but not all of them [38].

3.2 Extensions

Over time, some limitations of the algorithm have been discovered. This section focuses on surveying the areas where iForest is lacking and potential ways how the algorithm could be improved.

3.2.1 Non–parallel splits

One downside of Isolation Forest is that each split is done by a single feature. If we imagine this in the vector space, the data points $x \in \mathbf{X}$ are being split by a separating hyperplane, which is always parallel to the axis. The hyperplane f can be defined by a normal vector n and a intercept b in a linear equation:

$$f(x_i) = x_{i,1}n_1 + x_{i,2}n_2 + .. + x_{i,n}n_p - b,$$

where at least one n_i is non–zero. The hyperplane splits the vector space into two subsets:

$$\{x_i | f(x_i) < 0\},$$
$$\{x_i | f(x_i) \geq 0\}.$$

In axis–parallel split, the normal vector of the hyperplane contains the value 1 in one dimension and zeros in all the others, while the intercept represents the split value of that feature. There is no fundamental reason why these hyperplanes should be restricted in this way [39]. The class of the points is often related to combination of multiple variables, which means the axis–parallel splits cannot isolate these points as effectively. Splitting the vector space this way can also introduce bias and artifacts in the predicted anomaly scores (score maps) [39] causing an effect called “ghost regions”, which can be seen in Figure 3.3c and can cause false negative points in the prediction.

3.2.2 Extended Isolation Forest

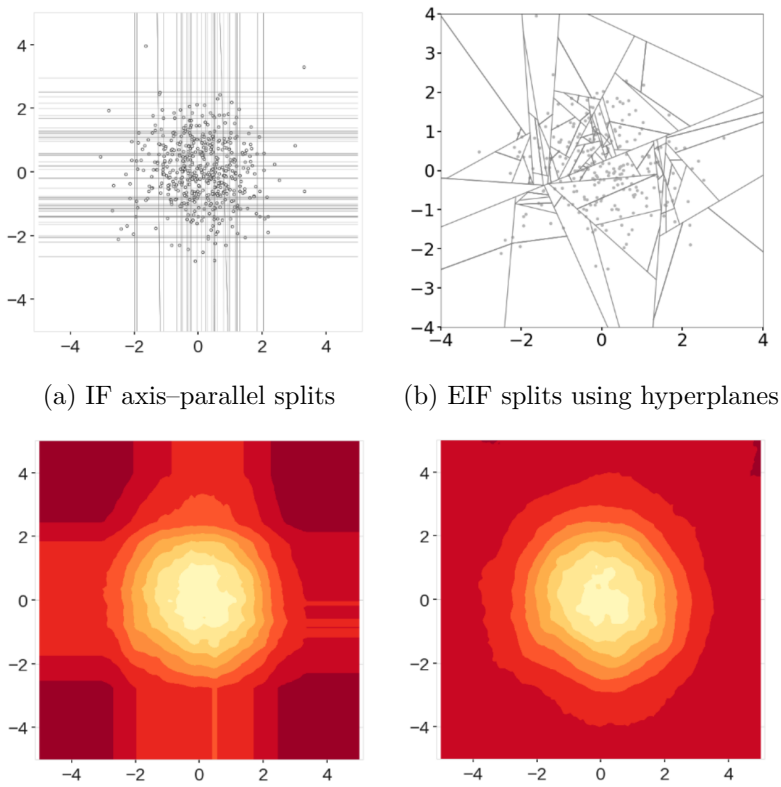
The **Extended Isolation Forest** (EIF in short) [39] proposes that the splits should be made non–parallel to the axis. Instead of selecting a random feature and split value, we generate a random slope (equivalent of normal vector) and a random intercept – which represents a hyperplane in the vector space that splits it into two parts. This is the new splitting criterion in the isolation tree.

This idea is based on the Oblique decision tree [40]. The normal vector $n \in \mathbb{R}^p$ is chosen randomly over the unit p–sphere, accomplished by drawing each coordinate of the vector from a normal distribution $\mathcal{N}(0, 1)$. The intercept $b \in \mathbb{R}^p$ is also a vector instead of a real number – meaning we can shift the split point of each feature independently. Based on this, we get the split criteria:

$$\mathbf{X}_l = \{x_i | (x_i - b)^T n < 0\},$$

$$\mathbf{X}_r = \{x_i | (x_i - b)^T n \geq 0\},$$

$$\forall i = 1..N.$$



(a) IF axis–parallel splits

(b) EIF splits using hyperplanes

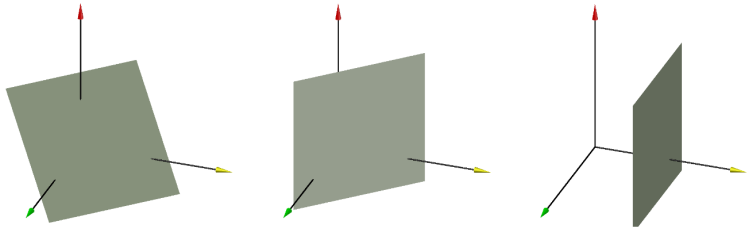
(c) IF scoremap

(d) EIF scoremap

Figure 3.3: Benefits of using non-parallel splits [39]

Let's take a closer look at the structure of the normal vector. It consists of p components, at least one of which is non–zero. The EIF introduces a new hyperparameter called the extension level, which determines how many of the vector components are set to non–zero. The extension level can be set in the range of $[0, p - 1]$ with the recommendation to use the full extension $(p - 1)$.

As mentioned earlier, the axis-parallel splits used in the standard iForest use a normal vector that contains one non–zero component. This means that the iForest is a special case of EIF with extension level set to 0 and EIF is its generalization.



(a) Ex 2 (b) Ex 1 (c) Ex 0

Figure 3.4: Illustration of various extension levels in EIF [39]

These are the only changes to the original iForest algorithm. The path length (except the splitting method) and the anomaly score evaluation remain the same. Here is the full pseudocode for building the tree:

Algorithm 6 EIF Tree

Input: $\mathbf{X}' \in \mathbb{R}^{N,p}$ – input data, e – current tree height, l – height limit, $extLevel$ – extension level

Output: an iTree

if $e \geq l$ or $|\mathbf{X}'| \leq 1$ **then**

return $exNode\{Size \leftarrow |\mathbf{X}'|\}$

else

randomly select a normal vector $n \in \mathbb{R}^p$, drawing each coordinate from normal distribution $\mathcal{N}(0, 1)$

randomly select an intercept $b \in \mathbb{R}^p$ in the range of $\mathbf{X}'$

set coordinates of n to zero, according to $extLevel$

$\mathbf{X}_l \leftarrow \{x_i | (x_i - b)^T n < 0\}$

$\mathbf{X}_r \leftarrow \{x_i | (x_i - b)^T n \geq 0\}$

return inNode$\{Left \leftarrow iTree(\mathbf{X}_l, e+1, l, extLevel)$,
$Right \leftarrow iTree(\mathbf{X}_r, e+1, l, extLevel)$,
$Normal \leftarrow n$,
$Intercept \leftarrow b\}$

end if

3.2.3 Split point guiding criteria

Another important aspect we can study is the randomness of the algorithm. The Isolation Forests select the split feature and value uniformly at random. If we could find a way to guide the search of the split criteria to more appropriate regions, there is a possibility that the results could be improved.

3.2.4 Optimized computational framework for Isolation Forest

The Optimized computational framework for Isolation Forest (OIF) [41] is another extension of the Isolation Forest algorithm. It proposes a metric that quantifies the suitability of the split and an algorithm that searches for the best split at each step, instead of choosing uniformly at random.

The proposed metric is based on the probability density function (PDF), which represents the distribution of attribute values. The assumption is that the corresponding attribute values for each class (anomalous or normal) tend to cluster together and have some centralized probability density values [41]. The less these clusters overlap, the more separable they are when we look at their mixed distribution. This is illustrated in Figure 3.5. On the left plot we can see that the two distributions overlap heavily, so the mixed probability density has only one peak. This is much harder to separate, in contrast to the distributions in the right plot where it's easy to see that the mixed distribution consists of two separate distributions.

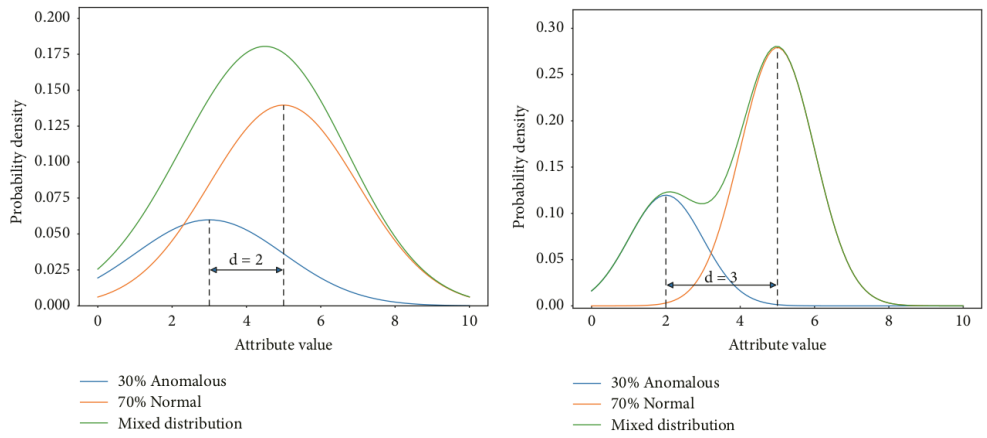


Figure 3.5: Visualization of separability between normal and anomalous points on two different mixed distributions [41]

Based on these assumptions, we aim to find a split point that best separates the two distributions. The proposed metric that quantifies the separability for each feature and the split point v is called a separability index and can be computed as:

$$sep(X^k, v) = \frac{e^{\sqrt{(E(x|x\in X^k, x<v) - E(x|x\in X^k, x>v))^2}} var(X^k)^2}{\alpha var(x \mid x \in X^k, x < v) + \beta var(x \mid x \in X^k, x > v)},$$

where X^k is vector of unique values of the k–th feature in dataset $\mathbf{X}$, v is the split value and:

$$\alpha = \frac{|\{x \mid x \in X^k, x < v\}|}{|X^k|},$$

$$\beta = \frac{|\{x \mid x \in X^k, x > v\}|}{|X^k|},$$

are the ratios of the two sets after splitting with respect to the whole X^k. When maximizing the separability index, we maximize the difference between the means of the two sets after splitting multiplied by the variance of the feature values and minimize the sum of their variances weighted by their sizes. Simply put, we try to get the two clusters of points to be as far apart as possible and each to have a smallest possible variance.

The paper also proposes an algorithm for fast search of the split point called *gradFindSplit* [41]. For a given feature X^k, we first have to get all unique values of this feature and sort them in ascending order. Then the separability index is computed for each of the values. This exhaustive search would be slow if the data set was large, so a gradient search method is used. The gradient for two adjacent values is defined as:

$$grad = \frac{sep(X^k, x_{i+1}) - sep(X^k, x_i)}{x_{i+1} - x_i}. \tag{3.4}$$

The value of *grad* can indicate whether the current split point is close to the optimal one – meaning it maximizes the separability index. Based on the value:

- If $grad \approx 0 \Rightarrow$ separability index is possibly approaching its maximum.
- The more *grad* deviates zero, the more values can be skipped over.

To quantify how many values can be skipped over, the method defines a *step* variable:

$$step = \begin{cases} \dfrac{3}{1 + e^{grad * log_1 0 |X^k|}} * \dfrac{|X^k|}{100}, & grad < 0 \\ 0.7 - \dfrac{1.3}{1 + e^{grad * log_1 0 |X^k|}} * \dfrac{|X^k|}{100}, & grad \geq 0 \end{cases} \tag{3.5}$$

Here is the detailed description of this algorithm in pseudocode:

Algorithm 7 gradFindSplit

Input: X^k - sorted values of the k–th attribute
Output: *bestX* - the attribute value having the largest value of the separability index, *bestSep* - the largest value of the separability index

```
step ← ⌈|X^k| * 0.01⌉
bestSep ← sep(X^k, x_1)
bestX ← (x_1 + x_2)/2
i ← 0
while i < |X^k| do
    i ← i + step
    currSep ← sep(X^k, x_i)
    if currSep > bestSep then
        bestSep ← currSep
        bestX ← (x_i + x_{i+1})/2
    end if
    update step following formulas (3.4) and (3.5)
end while
return bestX, bestSep
```

The method also suggests picking k random features in each step, where for each one the best split point according to *sep* is found. Then the combination of the feature and its split point with the highest *sep* is selected for the split. The k becomes the new hyperparameter for this method. By increasing k we decrease the random aspect of the algorithm (which may not be beneficial in all cases) and also increase the computation time for each tree.

Algorithm 8 OIF Tree

Input: $\mathbf{X}' \in \mathbb{R}^{N,p}$ – input data, e – current tree height, l – height limit, $k \in [1, p]$ – number of features to try at each step
Output: an iTree

```
if e ≥ l or |X'| ≤ 1 then
    return exNode{Size ← |X'|}
else
    Q ← k uniformly randomly selected features
    bestSep ← −∞
    for each q ∈ Q do
        x, sep ← gradFindSplit(q)
        if sep > bestSep then
            bestX ← x
            bestSep ← sep
            bestAttr ← q
        end if
    end for
    X_l ← {x_i ∈ X' | x_{i,bestAttr} < bestX}
    X_r ← {x_i ∈ X' | x_{i,bestAttr} ≥ bestX}
    return inNode{Left ← iTree(X_l, e + 1, l, k),
                  Right ← iTree(X_r, e + 1, l, k),
                  SplitAtt ← bestAttr,
                  SplitValue ← bestX}
end if
```

Chapter 4

Oblique split guiding metric

The two variations presented each extend the basic Isolation Forest algorithm in its own direction. The OIF may seem like a step back from the point of view that it uses the axis–parallel splits. On the other hand, the EIF may benefit from using a non–uniformly–random splits guided by some criterion. In this chapter, I present my experiments with combining the two methods together. All the algorithms in this section use the same method for computing path length, anomaly score, subsampling, and building the ensemble, as in the standard Isolation Forest, so the focus is set on algorithms for single tree building. A summary evaluation and performance comparison of the methods is provided at the end of the chapter.

4.1 Evaluation method

For prototyping purposes, the Python programming language was used for all the following implementations along with the NumPy library [42] and the source codes are included in the attached repository. All implementations except the standard iForest use parallelism for tree building to speed up the training time. Since the models are randomized, each evaluation was repeated 5 times and the average of the measurements was used as the result. Each time the model is fitted to the full dataset and then the predictions are made on the same data. The measured anomaly scores are then compared to the labels from the dataset. The experiments were run on an Intel Core i5–4200U processor with two cores (4 threads) at 1.6 GHz and 8 GB of memory.

4.1.1 Datasets

The datasets for evaluation were obtained from ODDS (Outlier Detection DataSets) [43] and are commonly used to evaluate anomaly detection models. They contain labels (ground truth) for each of the data points, which are

used for evaluation only, since all the used methods are unsupervised. The selected datasets differ in types of anomalies they contain, so that the detection performance can be tested in various scenarios.

Dataset	# of samples	# of features	Anomaly ratio
Annthyroid	7200	6	7.4%
Pima	768	8	34.9%
PenDigits	6870	16	2.3%
Ionosphere	351	33	35.9%

Table 4.1: Properties of datasets used for evaluation

The Annthyroid dataset consists of 3 ordinal classes, where the center class is considered normal and the two outer as anomalous (clustered outliers). Most of the outliers can be distinguished by a single high–skew column. The Pima dataset is an unbalanced binary–classification dataset with minority class labeled as outliers, which are of scattered type. Ionosphere is a high–dimensional dataset with scattered outliers. PenDigits dataset is a multi–label classification scenario where outliers are a single down–sampled class while inliers are the non–downsampled classes [38].

4.1.2 Metrics

Since the datasets are labeled, we can compare the model predictions with the ground truth. A commonly used metrics for binary classification, where the output of the model is in the form of a probability value, were used [44]:

Area under ROC curve (AUROC) – is obtained by calculating the area under the receiver operating characteristic (ROC) curve, which is a graphical representation of the trade–off between the true positive rate (TPR) and the false positive rate (FPR) of the model. The ROC curve is generated by varying the threshold of the predicted probability values and calculating the TPR and FPR at each threshold. The AUROC ranges from 0 to 1, with a higher value indicating better performance of the classification model (AUROC of 1.0 indicates perfect prediction).

Area under Precision–recall curve (AUPR) – the calculation is similar to AUROC but uses the trade–off between the precision and recall at different probability thresholds. This metric gives a more informative picture in the case of unbalanced datasets, which is well–suited for anomaly detection and may be a better option than the AUROC.

4.2 Implementation of OIF

The first task for the experiments was to implement the OIF. The downside is that the authors did not publish a reference implementation and I didn't find any other existing one, so my implementation had to rely solely on the pseudocodes and description in the paper. I started by implementing the original Isolation Forest to use as a reference for evaluation and also as a base for the extensions. I verified the implementation by comparing my results with those in the paper [33] and they are consistent.

Annthyroid			Pima		
AUROC	AUPR	Time [s]	AUROC	AUPR	Time [s]
0.826	0.300	0.178	0.678	0.512	0.204

Ionosphere			PenDigits		
AUROC	AUPR	Time [s]	AUROC	AUPR	Time [s]
0.853	0.802	0.286	0.950	0.265	0.374

Table 4.2: Evaluation of iForest implementation

The first prototype of OIF was implemented without the *gradFindSplit* algorithm, using only the split point guiding criteria, to measure the effect of the metric on the detection performance. The use of min–max normalization in the method for finding the split point significantly improved the results. The authors of the OIF paper [41] don't specify the parameters used for evaluation. The parameters shared with the standard iForest were set to the recommended values [33]: 100 trees in the ensemble with the subsampling set to 256. Different settings for the parameter k, which determines the number of randomly selected features to try at each split, were tested to analyze the influence on the results.

	Annthyroid			Pima		
k	AUROC	AUPR	Time [s]	AUROC	AUPR	Time [s]
1	0.831	0.257	8.830	0.748	0.589	7.977
5	0.847	0.310	39.036	0.739	0.573	33.714
all	0.846	0.316	46.354	0.738	0.564	53.832

	Ionosphere			PenDigits		
k	AUROC	AUPR	Time [s]	AUROC	AUPR	Time [s]
1	0.831	0.794	11.656	0.941	0.183	11.670
5	0.722	0.675	50.546	0.934	0.163	51.423
all	0.516	0.454	305.466	0.909	0.109	164.503

Table 4.3: Evaluation of OIF implementation

The results are similar to those of the standard iForest with a small improvement on the Anthyroid dataset and significant improvement in the case of the Pima dataset. We can observe that the hyperparameter k has only a small influence on the quality of the model on the Pima and Annthyroid datasets, while in other cases, the performance can decrease with increasing the parameter, which can indicate some bias in the metric and that more randomness is beneficial in these cases. The standard deviation of the evaluated metrics doesn't exceed 0.01 for any of the configurations, meaning that the model is consistent in its predictions. The authors have also evaluated their implementation on the Pima and Ionosphere dataset [41] and the results are consistent with mine with the setting $k = 1$.

4.2.1 The gradFindSplit method

The *gradFindSplit* method, which aims to to speed up the process of finding an optimal split, utilizes the *gradient* defined in (3.4) for computing a *step* variable (3.5), which determines how many data points to skip over. I noticed that in some of the datasets the *gradient* can be a large number because of the small variances in the denominator, which can lead to numerical instability and overflow in the exponential function. This was the case in my implementation and had to be solved in order to work correctly. Data normalization can help a bit, but it is not enough for some datasets. Another solution is to use an epsilon constant in the denominator, which can slightly affect the results. The measured times show that finding the split point with the use of *gradFindSplit* is about the same as when iterating through all the values.

	Time [s]	
Dataset	exhaustive search	gradFindSplit
Annthyroid	8.830	11.441
Pima	7.977	13.253
Ionosphere	11.656	13.401
PenDigits	11.670	15.758

Table 4.4: Single tree training times using gradFindSplit vs exhaustive search

The algorithm may be beneficial when using large subsample sizes, but this contradicts the statement in [33] saying that the sample size should be kept small to reduce masking and swamping effects. Nevertheless, I did a small experiment to test the performance on a larger dataset, where it might make some sense. I chose the ForestCover dataset from the ODDS [43], which has 286048 rows and 5 columns, with an anomaly point ratio of 0.9%. The training times were measured on a single tree with different subsampling sizes.

	Time [s]	
Subsampling size	exhaustive search	gradFindSplit
256	0.23	0.28
2048	1.77	2.11
8192	5.64	6.27
32768	22	21.7
131072	60	60
262144	95	91

Table 4.5: Single tree training times using gradFindSplit vs exhaustive search

The times are similar even on a larger dataset. Based on the experiment, I will not use the *gradFindSplit* algorithm going further, but it is possible that it could be faster on certain datasets when using very large sample sizes, so it is a potential point of future research if there is motivation to further optimize the Isolation Forest for big data processing.

4.3 Experiments with multi–dimensional separability index

The separability index proposed in OIF only considers a single feature at a time. If we want to use the oblique splits, the metric needs to be adapted to work in multiple dimensions or the feature space has to be projected into one–dimensional array of values.

My idea to extend the separability index to multiple dimensions was this: instead of subtracting the means of the feature values after the split, the distance between the centroids $E(\mathbf{X}_l)$ and $E(\mathbf{X}_r)$ (vectors of column–wise means) of the two sets $\mathbf{X}_l$ and $\mathbf{X}_r$ can be used, taking into account all the features. The variance can be replaced by a trace of a covariance matrix $\Sigma_{\mathbf{X}} \in \mathbb{R}^{p,p}$ – that means the sum of all the variances of the features. The modified metric can then be defined:

$$sep(\mathbf{X}', \mathbf{X}_l, \mathbf{X}_r) = \frac{e^{\sqrt{(E(\mathbf{X}_l)-E(\mathbf{X}_r))^2}} tr(\Sigma_{\mathbf{X}})^2}{\alpha tr(\Sigma_{\mathbf{X}_l}) + \beta tr(\Sigma_{\mathbf{X}_r})}$$

This metric can then be implemented in the Extended Isolation Forest by replacing the uniformly random split intercept selection with one that maximizes the metric value. The normal vector n can be randomly sampled in the same way as in EIF, but the intercept is optimized according to the separability index. Since the intercept in EIF is a p–dimensional vector, the problem

of finding the minimum of a single variable function grows to constrained optimization of a multi–variable function, which is much more difficult. The constraints are the $[min, max]$ in each feature. This split criterion can be expressed as

$$(x - b)^T n < 0,$$
$$b \in \mathbb{R}^p,$$

where x is the data point, b is the intercept and n is the normal vector. To implement a prototype of this idea, I tried using a function $minimize()$ from the SciPy library [45] to first test if it would work. Since this optimization takes place in every split, I used the Truncated Newton method [46], which can limit the number of iterations, for the price of getting only an approximation, which should be sufficient. I have evaluated the algorithm with different extension level settings. The algorithm is able to outperform standard iForest on some of the datasets, but the computation times are very slow compared to the iForest and its extensions, so only one attempt to generate the normal vector was used.

	Annthyroid			Pima		
ext	AUROC	AUPR	Time [s]	AUROC	AUPR	Time [s]
0	**0.768**	**0.329**	**90.379**	0.726	0.598	**124.580**
1	0.716	0.194	115.487	**0.740**	**0.616**	165.031
5	0.676	0.142	138.481	0.728	0.594	181.241
full	0.690	0.182	134.638	0.738	0.595	179.439

	Ionosphere			PenDigits		
ext	AUROC	AUPR	Time [s]	AUROC	AUPR	Time [s]
0	**0.875**	**0.842**	643.112	0.965	0.333	**334.074**
1	0.835	0.748	600.854	**0.965**	**0.338**	395.147
5	0.785	0.617	692.706	0.945	0.302	825.031
full	0.799	0.649	**582.115**	0.944	0.281	720.183

Table 4.6: Evaluation of OIF with p-dimensional intercept optimization

The results indicate that using axis–parallel splits (extension level of 0) with this metric works best on the Annthyroid and Ionosphere datasets, while for the other two the setting doesn't have much impact on the performance. The parameter also has a direct impact on the training time, as the metric takes longer to compute and the optimization becomes increasingly difficult.

After this evaluation, I realized that a simpler solution can be implemented. The p–dimensional intercept can be replaced by a 1–dimensional one, which can be used together with the normal vector to create a separating hyperplane. If the data points were projected onto the hyperplane, the result would be an array of singular values representing the possible split points, which would simplify the problem of finding the best split to minimizing a single variable function. Also, this way the *gradFindSplit* algorithm or the exhaustive search could be used on the transformed data, as in the OIF implementation. The split criterion with this approach can be denoted as:

$$x^T n < b,$$

$$b \in \mathbb{R}.$$

For testing, I used the same parameters as in the model above. An exhaustive search is used to find the optimal split point.

	Annthyroid			Pima		
ext	AUROC	AUPR	Time [s]	AUROC	AUPR	Time [s]
0	0.756	0.359	33.165	0.586	0.476	**47.688**
1	**0.794**	**0.427**	**32.689**	0.581	0.467	48.830
5	0.744	0.303	33.169	**0.622**	**0.480**	52.644
full	0.737	0.291	32.922	0.604	0.454	53.910

	Ionosphere			PenDigits		
ext	AUROC	AUPR	Time [s]	AUROC	AUPR	Time [s]
0	0.764	0.629	**76.190**	**0.961**	**0.335**	**59.088**
1	0.807	0.696	77.423	0.958	0.333	59.740
5	0.829	0.722	77.771	0.943	0.217	61.484
full	**0.834**	**0.745**	81.583	0.913	0.176	65.150

Table 4.7: Evaluation of OIF with 1-dimensional intercept optimization

With this approach, the training times were reduced, but the results remained similar or slightly worse in some cases. The training times are more consistent across the configurations because the number of used features has little influence on the optimized metric, so the duration depends only on the dataset size. The results are not very promising when compared to the standard iForest. This method performs worse in 3 out of 4 cases, while the training times are much longer.

After additional research focusing on the split criterion, I found another method and its variant that is based on the same principle I had in mind: non–parallel splits along with a split guiding criterion. The difference is that the method looks on the split metric differently. The new metric considers only the data points after projection onto the hyperplane and not the data points in the original vector space. This reduces the processing time and at the same time improves the results.

4.4 SCiForest

SCiForest stands for Isolation Forest with Split–selection Criterion [47]. The algorithm replaces uniform random splits by generating a linear combination of a selected number of features at each step and then finding the best split point guided by a gain criterion that minimizes the average standard deviation between the two split sets. The linear combination is equivalent to generating a random hyperplane and projecting the data points onto it. The paper [47] defines hyperplane as:

$$f(x) = \sum_{j \in Q} c_j \frac{x_j}{\sigma(X^j)} - b,$$

where x is the data point, Q is a set of randomly selected feature indices, X^j is a vector of values of the j-th feature, c_j is a random number uniformly sampled from the interval $[-1, 1]$ and b is the hyperplane intercept, which is equivalent to the selected split point. To find the split point, we project the all the data points in the branch onto the hyperplane, obtaining a set Y. The split point $b \in Y$ separates Y into $Y_l = \{y \in Y | y < b\}$ and $Y_r = \{y \in Y | y \geq b\}$, and the one that maximizes the following gain criterion is selected:

$$gain_{averaged}(Y) = \frac{\sigma(Y) - \frac{\sigma(Y_l)+\sigma(Y_r)}{2}}{\sigma(Y)} \tag{4.1}$$

In each step, τ hyperplanes should be generated (the paper [47] suggests 10) along with finding the best split point for each one and selecting the pair with the maximum gain. The optimal number of randomly selected features to create the hyperplane has been empirically found to be 2, specified by the parameter q. The advantage of this method is that the data projection and gain computation are fast operations, so the training phase should be much faster than the previously proposed methods without the need for sorting or gradient methods (exhaustive search is used here).

Algorithm 9 SCiForest Tree

Input: $\mathbf{X}' \in \mathbb{R}^{N,p}$ – input data, e – current tree height, l – height limit, τ – number of hyperplanes, q – number of features used in the hyperplane
Output: an iTree
  if $e \geq l$ or $|\mathbf{X}'| \leq 1$ **then**
    return $exNode\{Size \leftarrow |\mathbf{X}'|\}$
  else
    $f \leftarrow$ a hyperplane with the best split point b that yields the highest *gain* among τ hyperplanes of q randomly selected attributes.
    $\mathbf{X}_l \leftarrow \{x_i \in \mathbf{X}' | f(x_i) < 0\}$
    $\mathbf{X}_r \leftarrow \{x_i \in \mathbf{X}' | f(x_i) \geq 0\}$
    return inNode$\{Left \leftarrow iTree(\mathbf{X}_l, e+1, l, \tau, q)$,
      $Right \leftarrow iTree(\mathbf{X}_r, e+1, l, \tau, q)$,
      $SplitPlane \leftarrow f\}$
  end if

4.5 Fair Cut Forest

Fair Cut Forest (FCF) [48], developed in 2019 originally for missing data imputation, is a method based on the SCiForest algorithm, but improved in some areas – mainly the gain used to select the best split point in each step. The SCiForest model uses an averaged gain, which favors partitions where only one or a few points are put into a single branch while the majority go to the other, which isolates extreme values well, but may not produce enough splits in the non–extreme regions [38]. This can also cause the trees to require a much higher maximum depth to be able to separate the anomalies correctly. The effect can be shown on experiment preformed in [38], which visualizes the difference between the extrapolated depth and the real depth in the case of standard iForest and SCiForest, and shows that on average the extrapolated depth by SCiForest can be several times smaller than in the case of iForest.

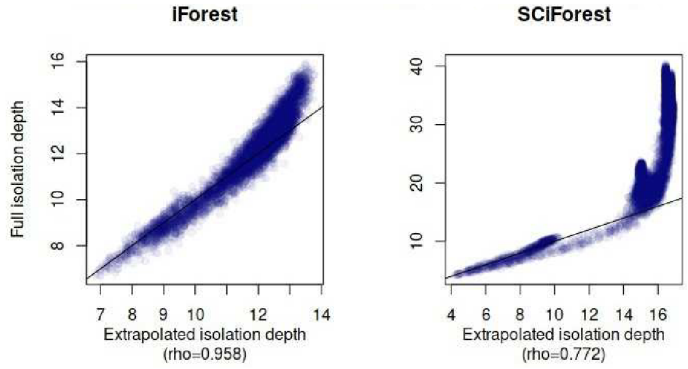


Figure 4.1: Real vs extrapolated isolation depth in iForest and SCiForest [38]

The metric in the Fair Cut Forest, called the pooled gain, takes into account not only the homogeneity of the two split sets, but also the difference in size, so the average of the standard deviations is weighted. This split criterion produces splits that work especially well in mixed distributions. The maximum pooled gain from a split point across different variables is produced by splits where the separability is clearer or where clusters are more easily formed [38]. It is defined as:

$$gain_{pooled}(Y) = \frac{\sigma(Y) - \frac{|Y_l|\sigma(Y_l)+|Y_r|\sigma(Y_r)}{|Y_l|+|Y_r|}}{\sigma(Y)} \tag{4.2}$$

The comparison between the split points can be seen in the Figure 4.2. On the left plot, the split point in the data with normal distribution $\mathcal{N}(0,1)$ guided by the pooled gain is the same as the density estimated center. When the same is repeated on data with gamma distribution $\Gamma(1,1)$, the pooled gain split point matches the averaged gain one. Finally, if we use a data based on Gaussian mixture, the pooled gain clearly separates the two mixed distributions, while the averaged gain attempts to isolate extreme values.

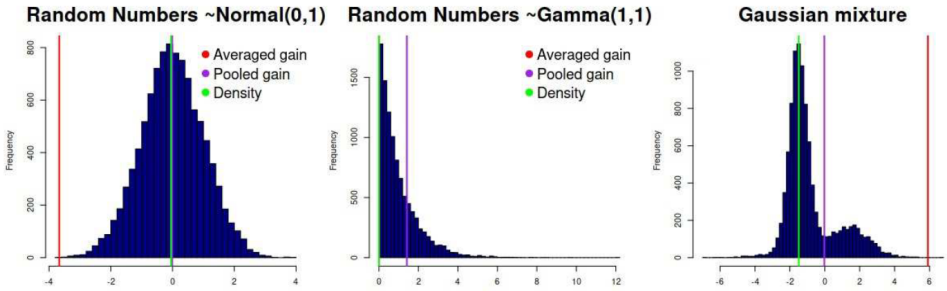


Figure 4.2: A comparison of split points using different guiding criteria [38].

The method randomly selects features for the linear combination one at a time. If the feature has only rows with one unique value, it can be discarded and another one selected. If all the features have only one value, the node is labeled as external and the recursion ends. The FCF also uses a standardization of the data points by subtracting the mean and dividing by the standard deviation across the features:

$$\mathbf{X}_{:,q(standardized)} = \frac{\mathbf{X}_{:,q} - E(\mathbf{X}_{:,q})}{\sigma(\mathbf{X}_{:,q})}$$

The points are standardized before each split and the $E(\mathbf{X}_{:,q})$ and $\sigma(\mathbf{X}_{:,q})$ are a part of the node structure, so the data in the prediction phase could be standardized using the same parameters. Another slight difference is that the coefficients of the hyperplane normal vector are sampled from a normal Gaussian distribution $\mathcal{N}(0,1)$ instead of a uniform distribution. The algorithm for building a Fair Cut tree is essentially the same as in the SCiForest, with the differences mentioned above.

4.5.1 My implementation

My implementations share the same codebase because these methods are quite similar. Both implementations use data point standardization at every step (as suggested in [38]). The only differences are:

- Distribution of normal vectors – SCiForest implementation uses uniform distribution $U(-1,1)$ and FCF uses normal distribution $\mathcal{N}(0,1)$ to sample normal vector elements from.
- Guiding metric – SCiForest uses the averaged gain (4.1), while FCF uses the pooled gain (4.2) metric for split point selection.
- Naming of the hyperparameters – Both methods use the same notation for parameter τ, specifying the number of hyperplanes generated at each step, where the best one is selected. The difference is in the parameter that determines the number of features used in the normal vector. The SCIForest paper [47] denotes it as q, the Fair Cut Forest uses p in [38] and m in [48]. I decided to use the extension level parameter (ext) as to stay consistent with the evaluation in the previous experiments. The extension level parameter is equivalent to $q-1$ in [47], $p-1$ in [38] and $m-1$ in [48].
- The SCiForest implementation does not use the range penalties at prediction time, as was described in the paper [47]. The prediction phase is the same as in standard Isolation Forest or the FCF.

The recommended configuration for SCiForest is 100 trees, subsampling size of 256, using a linear combination of two features (extension level of 1) and $\tau = 10$ attempts to generate the hyperplane. On the tested datasets, we observe that the performance varies with different configurations: Annthyroid and Pima benefit from a full extension, in contrast to Ionosphere and PenDigits, where smaller extension level is preferred. The parameter τ can slightly increase the accuracy in some cases, at the price of increased training times, mostly visible on the Ionosphere dataset. From the results it is difficult to find a universal configuration.

		Annthyroid			Pima		
τ	ext	AUROC	AUPR	Time [s]	AUROC	AUPR	Time [s]
1	0	0.643	0.158	10.046	0.551	0.396	**9.771**
	1	0.675	0.198	**9.837**	0.609	0.425	10.006
	5	0.757	0.348	10.152	**0.617**	0.466	10.162
	full	**0.757**	**0.365**	10.188	0.608	0.477	10.254
5	0	0.632	0.147	**45.127**	0.555	0.400	**43.023**
	1	0.701	0.197	45.692	0.612	0.427	46.483
	5	0.749	0.334	46.252	**0.631**	**0.480**	47.575
	full	**0.767**	**0.360**	48.367	0.606	0.477	48.023

		Ionosphere			PenDigits		
τ	ext	AUROC	AUPR	Time [s]	AUROC	AUPR	Time [s]
1	0	**0.878**	**0.842**	**9.719**	0.933	0.154	**9.338**
	1	0.854	0.812	10.402	**0.953**	**0.272**	10.056
	5	0.821	0.771	10.265	0.936	0.212	10.309
	10	0.830	0.774	10.546	0.920	0.192	10.431
	15	0.817	0.764	10.742	-	-	-
	full	0.808	0.751	11.193	0.924	0.176	10.721
5	0	**0.878**	**0.840**	**42.092**	0.950	0.196	**40.494**
	1	0.860	0.818	45.698	**0.950**	**0.235**	45.559
	5	0.829	0.777	47.421	0.934	0.204	47.538
	10	0.840	0.790	47.879	0.929	0.215	48.016
	15	0.831	0.778	48.911	-	-	-
	full	0.841	0.794	51.492	0.920	0.193	48.910

Table 4.8: SCiForest hyperparameter evaluation

The FCF results are an improvement over the SCiForest on all tested datasets. For the default configuration, it was recommended to use a single try for generating normal vector ($\tau = 1$) with the extension level of 1. However, my observations were a little different. Based on the evaluation, I find the best configuration to be one when using a full extension.

		Annthyroid			Pima		
τ	ext	AUROC	AUPR	Time [s]	AUROC	AUPR	Time [s]
1	0	**0.896**	0.392	**10.619**	0.737	0.534	**10.175**
	1	0.887	0.403	11.103	0.730	0.540	11.155
	5	0.883	0.436	13.497	**0.739**	0.554	14.201
	full	0.885	**0.447**	14.529	0.737	**0.559**	14.861
5	0	0.889	0.392	**46.507**	0.736	0.531	**44.466**
	1	**0.891**	0.421	48.893	0.732	0.534	49.565
	5	0.883	0.430	62.724	0.738	0.554	62.968
	full	0.883	**0.436**	62.224	**0.742**	**0.564**	68.280

		Ionosphere			PenDigits		
τ	ext	AUROC	AUPR	Time [s]	AUROC	AUPR	Time [s]
1	0	0.819	0.799	**10.100**	**0.958**	0.232	**9.981**
	1	0.865	0.850	10.944	0.945	0.218	11.337
	5	0.904	0.886	13.511	0.953	**0.273**	14.049
	10	0.911	0.897	15.813	0.943	0.259	17.670
	15	0.915	0.901	18.490	-	-	-
	full	**0.917**	**0.902**	25.860	0.953	0.258	19.459
5	0	0.824	0.804	**43.865**	**0.950**	0.204	**42.882**
	1	0.862	0.847	48.646	0.945	0.213	48.877
	5	0.908	0.894	59.853	0.944	0.224	62.341
	10	0.913	0.897	71.224	0.939	0.240	79.329
	15	0.916	0.902	82.964	-	-	-
	full	**0.918**	**0.903**	119.755	0.944	**0.258**	89.759

Table 4.9: Fair Cut Forest hyperparameter evaluation

4.6 Summary

Here is a summary of all the models presented in this section. In this overview, each model is evaluated with a single configuration that was found to be the best fit for that model. The best result in each dataset is highlighted. Times are not highlighted because the fully random Isolation Forest will always be faster than methods that use some computation. The method with optimization of the p-dimensional intercept from my experiments is labeled as OIF-p and the 1-dimensional variant as OIF-1.

Model	# of trees	Subsample size	τ	ext
IF	100	256	-	-
OIF	100	256	1	-
OIF–p	100	256	1	0
OIF–1	100	256	1	1
SCIF	100	256	1	5
FCF	100	256	1	full

Table 4.10: Configuration of selected models used for evaluation

		IF	OIF	OIF–p	OIF–1	SCIF	FCF
Annthyroid	AUROC	0.826	0.831	0.768	0.794	0.757	**0.885**
	AUPR	0.300	0.257	0.329	0.359	0.348	**0.447**
	Time [s]	0.178	8.830	90.38	32.69	10.15	14.53
Pima	AUROC	0.678	**0.748**	0.726	0.581	0.617	0.737
	AUPR	0.512	0.589	**0.616**	0.467	0.466	0.559
	Time [s]	0.204	7.978	165.0	48.83	10.16	14.86
Ionosphere	AUROC	0.853	0.830	0.875	0.807	0.821	**0.917**
	AUPR	0.801	0.793	0.842	0.696	0.771	**0.902**
	Time [s]	0.286	11.65	643.1	77.42	10.27	25.86
PenDigits	AUROC	0.950	0.941	**0.965**	0.958	0.936	0.953
	AUPR	0.265	0.183	0.333	**0.335**	0.212	0.213
	Time [s]	0.374	11.67	334.1	59.09	10.31	19.45

Table 4.11: Performance comparison of selected models

Based on the evaluation results and also the theoretical analysis, I selected the Fair Cut Forest model for the final implementation. It was able to outperform other methods on some datasets and was a very close competitor on others. Good results were obtained with a single configuration and the model should be able to deliver even better results when the parameters are configured according to the dataset properties.

Chapter 5

Implementation into the H2O–3 platform

5.1 The platform

H2O–3 is an open source, in–memory, distributed, fast, and scalable machine learning and predictive analytics platform developed by H2O.ai, that allows users to build machine learning models on big data [49]. Many global organizations develop their data analysis tools based on this platform [50]. In addition to the open source platform, the company also offers paid products such as the H2O Driverless AI [51] – automatic machine learning system or enterprise support for the open source products.

The open source platform uses a Java computing engine that implements the machine learning algorithms. The engine works as a server that can be run locally or in the cloud and provides interfaces for many languages, the main ones being Python, R and Scala, so that data analysts can use their favorite language without sacrificing performance and scalability. The H2O–3 supports many data sources besides local such as the Hadoop Distributed File System (HDFS), SQL or Amazon S3 cloud object storage and more. It also can also be run on Hadoop or Kubernetes clusters. For distribution, the H2O uses its own implementation of MapReduce framework and Java Fork/Join for multi–threading [49].

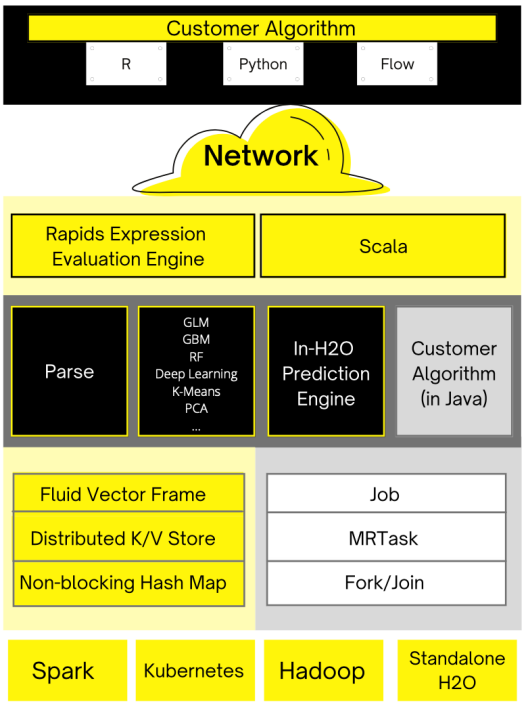


Figure 5.1: Architecture of H2O–3 platform [52]

5.2 MapReduce

MapReduce is a programming model and framework that enables distributed processing of large datasets on clusters. These datasets are split into small chunks that can be then processed in parallel. As the name suggests, it consists of two main components: the map function, which takes the input data chunk and performs some operation on it and the reduce function, which combines the results of the map function into a final output [58].

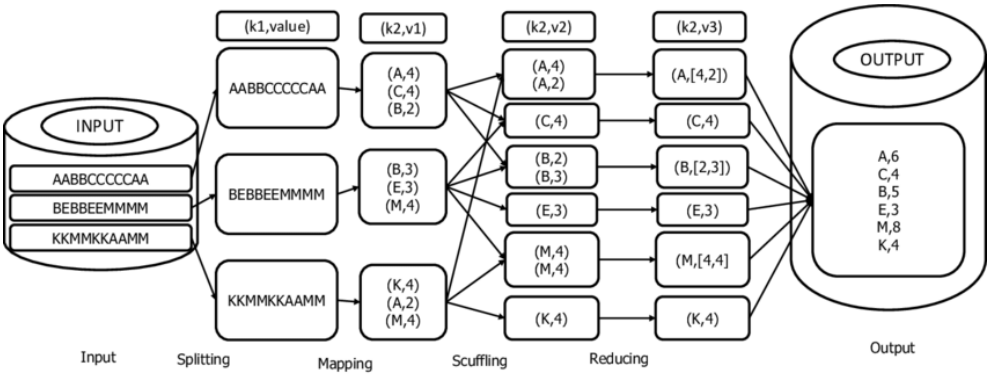


Figure 5.2: Example of MapReduce workflow [59]

5.3 Existing implementation

There are two existing implementations of the Isolation Forest in the H2O–3 library. The first one implements the standard Isolation Forest [53]. It includes fully distributed tree building and prediction phase, but has some drawbacks. The estimation of the unsuccessful searches in the binary search tree (3.2) is absent in this implementation because the distributed binary search tree structure ($DTree$ [54]) implemented in the H2O doesn't work with number of rows in the leaves. This can lead to inaccurate predictions, as was discussed in [55].

Second, a newer implementation was created as part of a diploma thesis [55], implementing the Extended Isolation Forest [56]. It is built from scratch without relying on the previous implementation and the source code is simpler and cleaner. However, some sacrifices have been made, where the tree building is not distributed, but it has been shown to be even faster on standard sizes of sample sizes, such as the recommended 256. For larger sample sizes (10 000+), it is more efficient to use the distributed computation. There are future plans to improve the algorithm by implementing a new distributed tree structure that will enhance the performance on very large datasets.

5.4 Fair Cut Forest implementation

The Fair Cut Forest implementation is based on the Extended Isolation Forest implementation and shares the same class structure, since the general skeleton for Isolation Forest and its extensions is the same. This will also simplify future updates for both algorithms.

The input of the model is specified by a parameter $training\_frame$, the type of which is an H2O data structure called the $Frame$. The $Frame$ is a distributed tabular data storage structure that can be used in MapReduce tasks [57]. The implementation allows setting a *seed* parameter – it will consistently build the same model when using the same seed. A trained model can be exported as a MOJO (Maven Old Java Object), allowing the model to be used outside of the H2O cluster. By its nature, the Isolation Forest algorithm supports only ordinal features. The implementation allows categorical features to be converted according to a specified schema using the *categorical_encoding* parameter.

The hyperparameters are:

- *ntrees* – the number of isolation trees in the ensemble. The default value is 100.
- *sample_size* – subsampling size (number of randomly chosen data points without replacement that each tree is built on). The default value is 256.
- *extension_level* – represents the number of non-zero elements in the normal vector (number of features used at each split) -1. It is equivalent to the parameter $p-1$ in [38], $m-1$ in [48] and $q-1$ in [47]. When set to 0, it is equivalent to axis–parallel split using single feature. The value can be set in the interval of $[0, p-1]$, where p is the number of features. Default value is 1, but it is recommended to use full extension $p-1$.
- *k_planes* – number of tries for generating a hyperplane at each split, where the one with maximum pooled gain is selected. The default value is 1. It is equivalent to parameter τ in [48] and [47].

The output of the model prediction is an H2O *Frame* with columns *anomaly_score* (defined in (3.3)) and *mean_length* (defined in (3.1)).

5.5 Structure

The following is a description of the modified or added files, classes and functions. All the files are included in the attached repository, or in the pull request [60]. To build and run the algorithm, the entire H2O–3 library has to be downloaded and built.

`h2o-algos/src/main/java/hex/tree/isoforfaircut`

class FairCutForest

Represents the training phase of the algorithm. Contains the *buildIsolationTreeEnsemble* function, which is responsible for subsampling and building the ensemble, implementing the pseudocode 3.1.1.

class FairCutForestModel

Defines the model hyperparameters and contains a function *score*0, which is used to calculate the anomaly scores score for the given input (prediction phase).

class FairCutForestMojoWriter

Adds the ability to export the model MOJO.

`h2o-algos/src/main/java/hex/tree/isoforfaircut/isolationtree`

class IsolationTree

Represents the tree structure and implements functions used for single tree building.

subclass *Node* – defines the node structure, which contains all the necessary information: data points, height of the node, number of rows, split criteria and information whether the node is external.

subclass *SplitCriteria* – encapsulates all the parameters that define the split. The normal vector and the intercept are stored to represent the separating hyperplane used for the prediction. The means and standard deviation of the features in the subsets are stored so that the points in prediction can be standardized with the same parameters. The gain value is stored to determine the best hyperplane among the hyperplanes generated. Finally, the data points projected onto the hyperplane are stored for faster splitting (no need to project them again).

buildTree – the main function for single tree construction.

$findSplit$ – finds the intercept (split point) for a given hyperplane that maximizes the pooled gain metric.

split – responsible for splitting the subset into two branches based on a given *SplitCriteria* instance.

interface AbstractCompressedNode and classes CompressedLeaf, CompressedNode and CompressedIsolationTree

These classes are used to reduce memory usage. The aim is to store only the data needed for the predictions. Upon branching in the training function, each node is converted to its compressed form. The *AbstractCompressedNode* interface is used to convert the structure of a particular type of node (leaf of internal node) into a binary form in *FairCutForestMojoWriter*. A fully built tree is stored as a *CompressedIsolationTree*.

class IsolationTreeStats

Used to store the statistics about the training process. Some examples are minimum, maximum or average statistics for depth, isolated and non–isolated points.

`h2o-algos/src/main/java/hex/api/`

class RegisterAlgos

Added entry for the FairCutForest to generate the APIs.

`h2o-algos/src/main/java/hex/schemas`

classes FairCutForestModelV3 and FairCutForestV3

Required classes for API generation, containing definitions of the model input parameters and output.

`h2o-algos/src/test/java/hex/tree/isoforfaircut`

Java unit tests (based on the tests of H2O EIF [56]).

`h2o-genmodel/src/main/java/hex/genmodel/algos/isoforfaircut`

classes FairCutForestMojoModel, FairCutForestMojoReader

Used to import the model MOJO and make predictions using that model.

`h2o-bindings/bin/gen_{python, R}.py`

These files are responsible for generating the API for the Python and R languages, so the entries for the new algorithm have been added here.

`h2o-bindings/bin/custom/{python, R}/gen_faircutforest.py`

Custom documentation strings for the Python and R interfaces that are added to the generated APIs.

`h2o-py/tests/testdir_algos/isoforfaircut/`

Tests for the Python API (grid search, categorical data and saving/loading a trained model) (based on the tests of H2O EIF [56]).

5.6 Example

Here is a basic usage of the Fair Cut Forest implementation using a Python API.

```
import h2o
from h2o.estimators import H2OFairCutForestEstimator

# Connect to the h2o server
h2o.init()

# Create the Frame from some tabular data (e.g. Pandas
   DataFrame or NumPy Array)
trainin_frame = h2o.H2OFrame(training_data)
testing_frame = h2o.H2OFrame(testing_data)

# Define the model
FCF_h2o = H2OFairCutForestEstimator(
    model_id = "fcf.hex",
    ntrees = 100,
    sample_size = 256,
    extension_level = training_data.shape[1] - 1,
    k_planes = 1)

# Train the model
FCF_h2o.train(training_frame = training_frame)

# Predict on the testing data
pred = FCF_h2o.predict(testing_frame)
```

5.7 Evaluation

This section discusses the evaluation of the implementation and a comparison with the co–existing implementations in the H2O–3 library. The evaluation consists of three parts. First, the implementations are tested on real datasets with ground truth, where the performance metrics and time are measured in the same way as in the previous chapter. The scalability tests follow. Finally, the models are trained on a selection of synthetic toy datasets to visualize the score maps and discuss the results in the novelty detection scenario.

5.7.1 Anomaly detection performance

The implementations were evaluated according to the same metrics used in the previous section – AUROC and AUPR. Training and prediction times were also measured. Each model was trained on the full data (with a limited subsample size, so the larger datasets are not fully used) and the predictions were made again on the whole dataset. The evaluation was repeated 5 times for each dataset. The H2O implementations were tested on a more powerful hardware: Intel Xeon E3–1245 with 4 cores (8 threads) at 3.7 GHz and 32 GB of memory.

A wider range of datasets from ODDS [43] were used for the evaluation, most with larger numbers of samples and features. These datasets are a standard for evaluating anomaly detection algorithms and are used in many papers.

Dataset	# of samples	# of features	Anomaly ratio
Annthyroid	7200	6	7.4%
ForestCover	286048	10	0.9%
http (KDDCUP99)	567479	3	0.4%
Ionosphere	351	33	35.9%
Mammography	11183	6	2.3%
mnist	7603	100	9.2%
OptDigits	5216	64	3%
PenDigits	6870	16	2.3%
Pima	768	8	34.9%
Satellite	6435	36	32%
Shuttle	49097	9	7%

Table 5.1: Properties of datasets used for final evaluation

The models were built using the following parameters:

	ntrees	sample_size	extension_level	k_planes
H2O FCF	100	256	full	1
H2O EIF	100	256	full	-
H2O IF	100	256	-	-

Table 5.2: Configuration of H2O implementations used for evaluation

		H2O FCF	H2O EIF	H2O IF
Annthyroid	Train time [s]	0.309	0.239	0.516
	Predict time [s]	0.594	0.554	0.224
	AUROC	**0.888**	0.640	0.818
	AUPR	**0.444**	0.167	0.313
ForestCover	Train time [s]	0.440	0.226	4.145
	Predict time [s]	6.059	6.480	1.873
	AUROC	0.739	0.723	**0.883**
	AUPR	0.021	0.016	**0.063**
http (KDDCUP99)	Train time [s]	0.299	0.220	4.801
	Predict time [s]	5.888	6.529	1.532
	AUROC	**0.998**	0.993	0.981
	AUPR	**0.380**	0.191	0.092
Ionosphere	Train time [s]	0.469	0.265	0.472
	Predict time [s]	0.221	0.226	0.227
	AUROC	**0.915**	0.893	0.835
	AUPR	**0.897**	0.857	0.784
Mammography	Train time [s]	0.225	0.235	0.434
	Predict time [s]	0.750	0.797	0.214
	AUROC	0.804	**0.869**	0.798
	AUPR	0.111	0.190	**0.289**
mnist	Train time [s]	0.840	0.637	1.047
	Predict time [s]	0.635	0.430	0.227
	AUROC	**0.825**	0.806	0.762
	AUPR	**0.307**	0.280	0.227
OptDigits	Train time [s]	0.629	0.470	0.647
	Predict time [s]	0.831	1.245	0.226
	AUROC	0.645	**0.681**	0.528
	AUPR	0.042	**0.045**	0.029
PenDigits	Train time [s]	0.440	0.265	0.597
	Predict time [s]	0.642	0.675	0.223
	AUROC	0.949	**0.956**	0.908
	AUPR	**0.277**	0.269	0.212

		H2O FCF	H2O EIF	H2O IF
Pima	Train time [s]	0.216	0.224	0.216
	Predict time [s]	0.222	0.231	0.222
	AUROC	**0.741**	0.655	0.602
	AUPR	**0.558**	0.502	0.450
Satellite	Train time [s]	0.469	0.223	0.432
	Predict time [s]	0.663	0.619	0.224
	AUROC	**0.794**	0.700	0.665
	AUPR	**0.716**	0.689	0.553
Shuttle	Train time [s]	0.434	0.231	1.061
	Predict time [s]	3.114	3.106	0.842
	AUROC	**0.997**	0.993	0.988
	AUPR	**0.925**	0.812	0.872

Table 5.3: Evaluation of H2O implementations

The Fair Cut Forest implementation was able to achieve the highest value of AUROC on 7 out of the 11 datasets tested and the highest AUPR on 8 datasets from the three implementations tested.

The training and prediction times are a huge improvement over the prototype Python implementations in the previous chapter. The EIF is on average faster than the FCF, but all training times are within a second. The training of the IF takes longer because it uses the $DTree$ structure for distributed computation, which is slower on a standard setting of sample size in Isolation Forest [55]. Overall, the training times are relatively low even on the biggest datasets due to the small subsampling size. As for the prediction phase, the H2O IF is faster because there is no computation in the tree traversal, unlike the FCF and EIF, where the points are projected on the splitting hyperplane and in the FCF also standardized.

Overall, the FCF implementation appears to be the best option of the three when used on an unlabeled dataset with no option to tune the parameters, considering both detection and time performance.

5.7.2 Scalability tests

The scalability tests were performed on the ForestCover dataset. Each model was evaluated 5 times and the results are an average of the run times. The default configuration was used for each implementation, the same as in the previous evaluation.

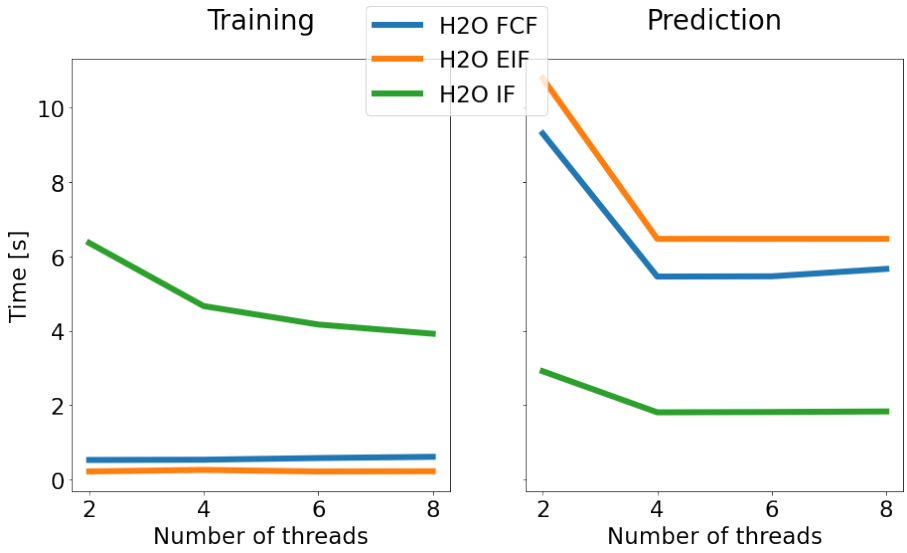


Figure 5.3: Scalability of prediction phase

The training times of EIF and FCF with the sample size of 256 and 100 trees are almost constant for all configurations of the number of threads. In the case of IF, the training times are logarithmically smaller with the number of threads, as was shown in [55]. In the prediction phase there is a visible decrease between two and four threads, but then the times remain constant.

These tests should be repeated on more powerful hardware to test the influence of more threads and test larger subsampling sizes or numbers of trees. Scalability tests were also performed in [61] on the EIF and IF implementations. Because the used structures are mostly the same with EIF and FCF, the results should be similar.

5.7.3 Novelty detection performance on toy datasets

Novelty detection is not a primary goal of this work, however it may be interesting to examine the behavior on previously unseen data and observe the anomaly score maps that these models produce. The testing involves generating synthetic data with different patterns representing possible scenarios, training the model on the data and visualizing the corresponding anomaly score maps. Evaluation is based purely on the visual representation by observing how the anomaly scores behave in regions where there are no data points are present. The anomaly scores are scaled to the interval $[0, 1]$ for each algorithm for better visibility on the heat maps. The discussion of the results follows after the score maps visualisations.

Single blob

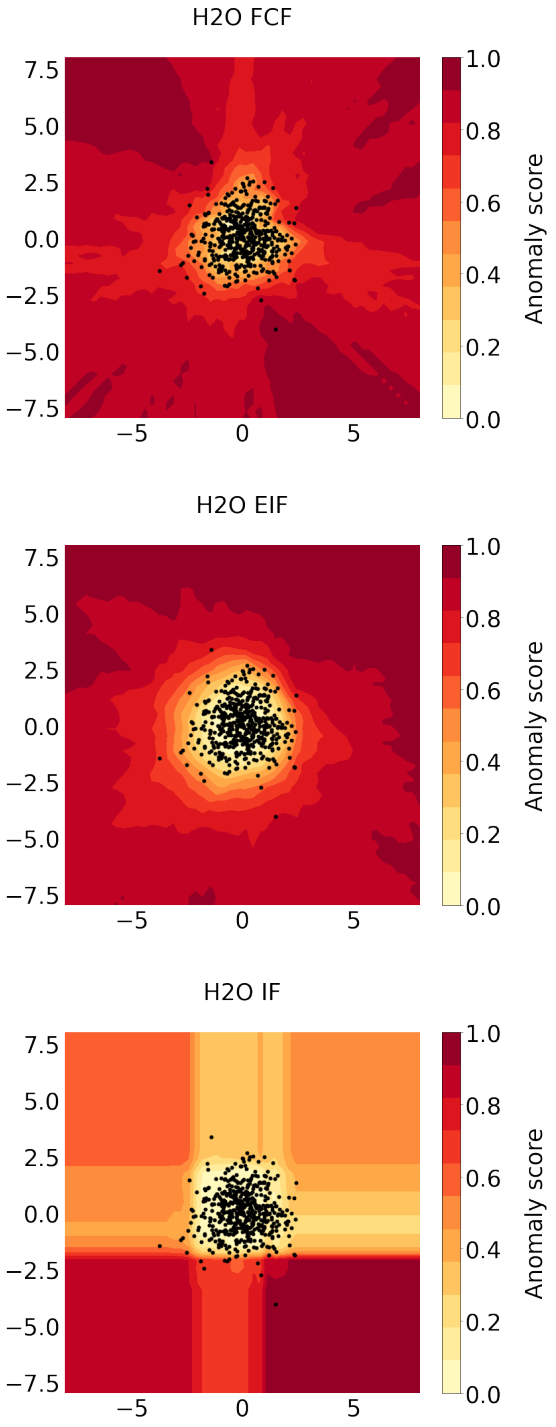


Figure 5.4: Anomaly score maps on a single blob

Double blob

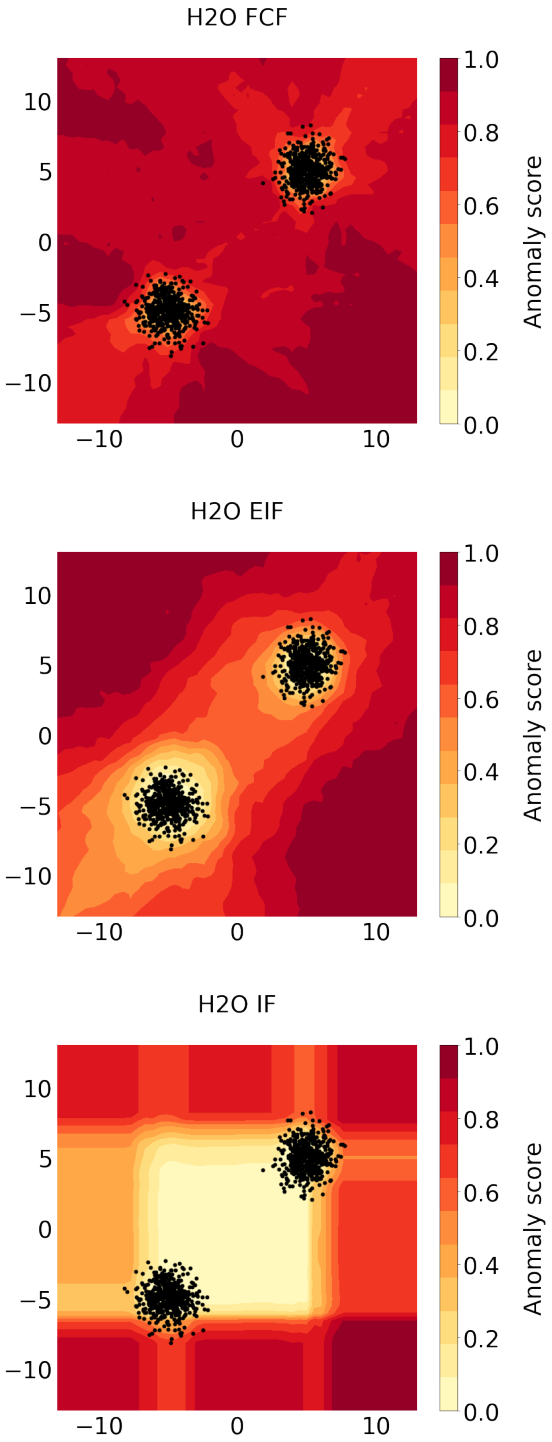


Figure 5.5: Anomaly score maps on a double blob

Single blob with overlapping anomaly cluster

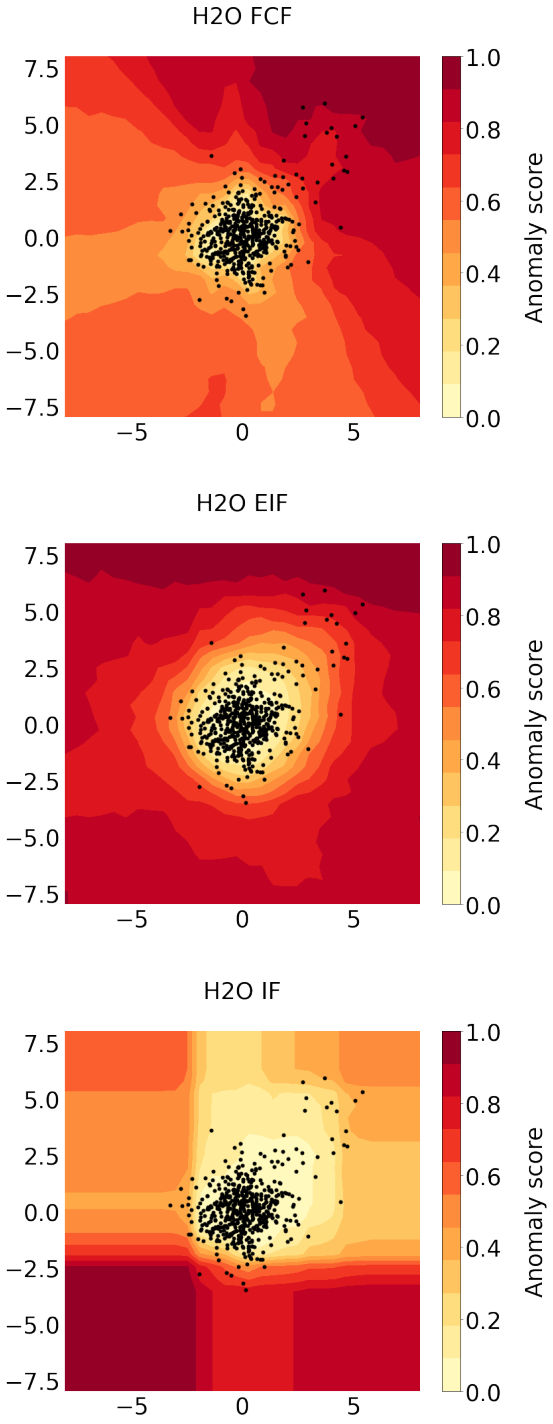


Figure 5.6: Anomaly score maps on a single blob with overlapping anomaly cluster

Moons

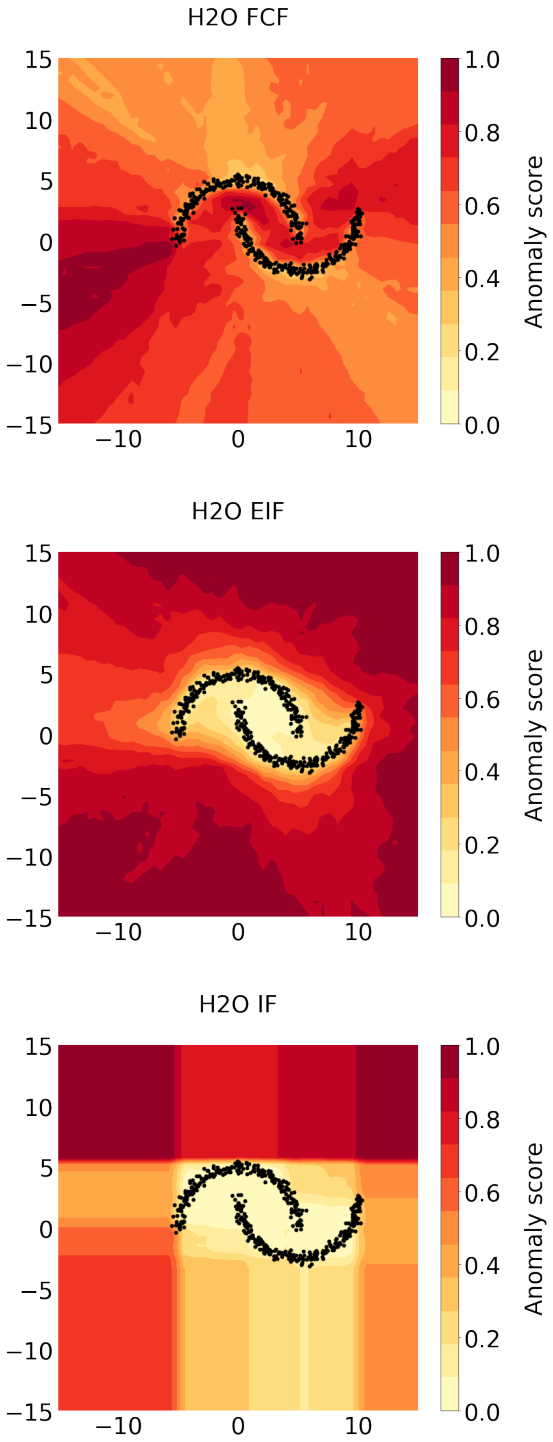


Figure 5.7: Anomaly score maps on the moons dataset

From the heat maps, it is clear that the H2O Isolation Forest implementation does not produce relevant results and the effect of ghost regions is strongly present in each of the tested scenarios. What is more interesting is the difference between the Extended Isolation Forest, where the splits are generated uniformly at random and the Fair Cut Forest, where the point that maximizes the pooled gain metric is selected. In the case of Fair Cut Forest, it is apparent that the splits are almost always performed in the regions with the highest data density, resulting in good separation between close clusters (e.g. the area between the two semicircles in the "moons" dataset or the area between the clusters in the double blob).

On the other hand, the regions without any points have almost no splits, so the low anomaly score zones are extended outward, creating a similar effect to the ghost regions. This is most visible on the single blob with overlapping anomaly cluster. The Extended Isolation Forest is able to create a tight outer boundary around the distributions, but may have difficulty in the areas between the clusters of points. This comparison makes the Extended Isolation Forest perhaps a more favorable option for the use in novelty detection, but at the same time shows the strength of the splits created by the Fair Cut Forest in the anomaly detection setting.

5.8 Possible improvements

5.8.1 Speed

The Isolation Forest is very fast, especially compared to other anomaly detection methods. This is mainly achieved by the random aspect of the algorithm with little computation in each step – only finding the minimum and maximum value of the subset. The method can also be well parallelized and distributed thanks to the tree ensemble that can be built in parallel.

With adding a function maximization problem to each step of the tree building, it is no surprise that the training times are significantly prolonged. The Optimized Computational Framework for Isolation Forest, which was studied in this thesis proposed the *gradFindSplit* method. In my tests, training times using method were similar to those using a exhaustive search. However, this could be different on very large datasets where there is motivation to use a large subsampling size, which can be tested in the future if needed. An alternative approach to speed up the training could be to skip some of some values. This approach is also used in decision trees to evaluate the gain in case of continuous variables in the dataset. Another way to speed up the training on very large datasets could be the use of distributed tree building.

5.8.2 Memory

The extension level parameter determines how many non–zero components are present in the normal vector – how many features are used in the linear combination to create the hyperplane. The vector is stored in each of the tree nodes. In addition, the means and standard deviations of the features used in the hyperplane, which are used for data standardization, are also stored. If the extension level is low (e.g. 0, equivalent to the standard Isolation Forest) but the dimensionality of the dataset is high, the space allocated for these arrays will remain unused, wasting memory. This can be optimized to use a sparse representation of the arrays, using only as much space as necessary to save memory. However, according to my observations, using the full extension gives the best overall quality of detection as measured by the AUROC and AUPR metrics. In this case, the sparse representation may slow down the computation and use even more memory. This can be a point of future upgrades of the implementation. A possible solution is to use a sparse representation if the extension level is below a certain limit (e.g. less than half of the features are used to create the hyperplane) and a dense representation otherwise.

5.8.3 Prediction phase

The thesis dealt with the improvement of the Isolation Forest, focusing only on the training phase, but all discussed algorithms share the procedure for the predictions of the anomaly score after the tree was built. This may be an area for future research to further improve the method. There are already some proposals on how to improve the computation of anomaly score in order to increase the predictive performance [62][63], but only a few compared to those dealing with the training of the model.

5.8.4 Novelty detection

As the novelty detection analysis showed, the Fair Cut Forest model is able to create high anomaly score regions even in between data clusters, unlike other methods. On the other hand, the splits follow the metric, resulting in too few splits in areas with no data points and low anomaly scores in those regions. This would lead to false negative predictions for new data in these regions.

My idea of how to achieve more splits even in these regions is to use a random step in the algorithm – similar to the GSAT algorithm. With a given probability, the algorithm makes a random split similar to the one in the Extended Isolation forest instead of using the metric guiding. This in result can combine the two methods and the predictions may benefit from this. The probability can be constant in each step, but a better option may be to decrease the probability as the depth increases, as the data subset is smaller with increasing depth, reducing the probability of splits targeting the empty regions. This idea seemed interesting, so I ran a small experiment using the following equation to calculate the probability of random split for each depth:

$$P * \frac{1}{depth + 1},$$

where P is a set constant. When using $P = 1$, the split at depth 0 is always random, at depth 1 with probability 0.5, at depth 2 with probability 0.25 etc. The decrease in probability is quick, but the sample size has a default value of 256, which means the maximum tree depth will be $\log_2(256) = 8$. I visualized the effect of different probability settings on the single blob dataset with overlapping anomaly cluster, which was the most extreme case in the preceding evaluation.

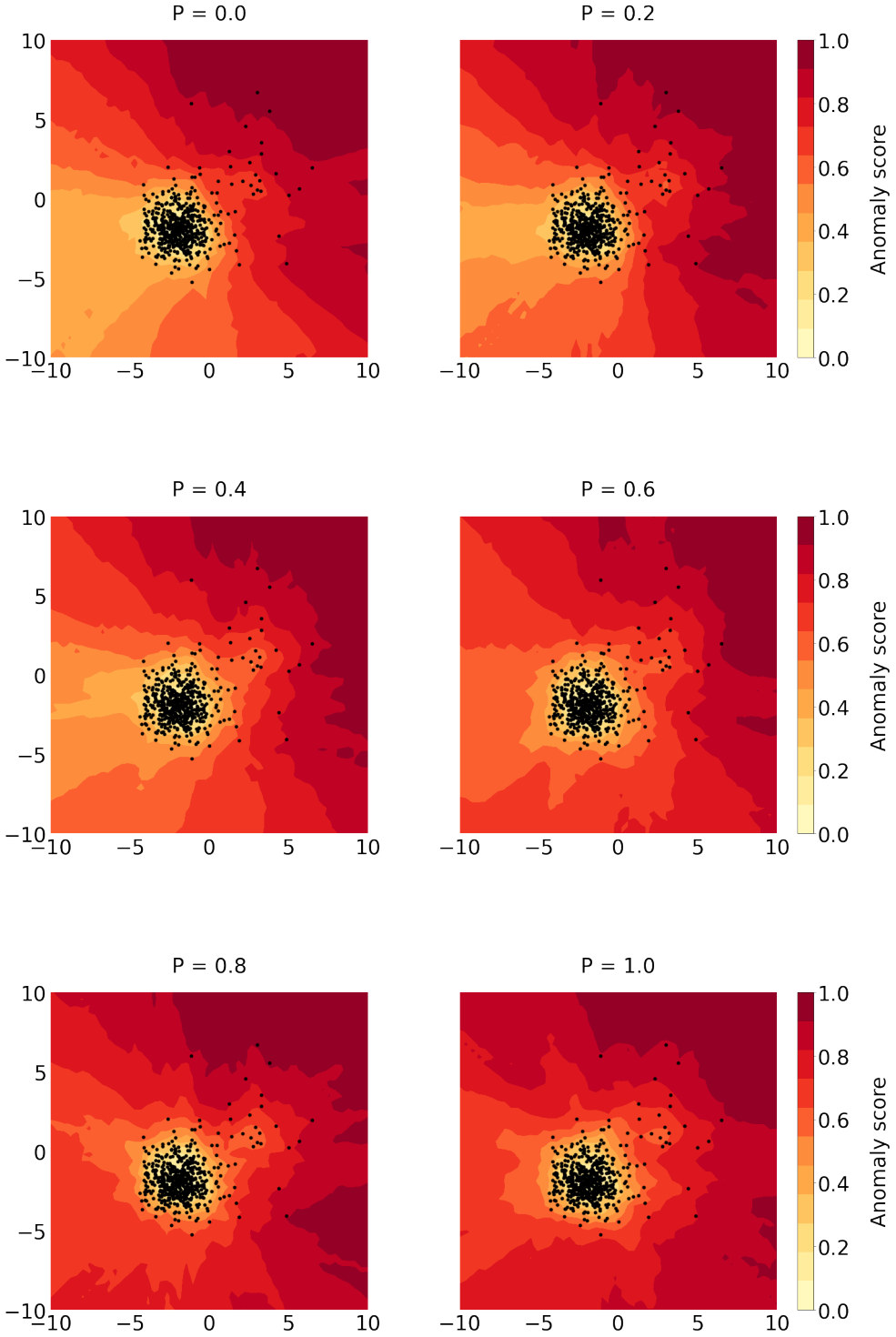


Figure 5.8: Anomaly score maps of FCF with random step

It is clear that as the probability of a fully–random split increases, the regions with lower anomaly scores tighten towards the normal cluster, reducing the unwanted effect without significantly affecting the boundary between the anomalous and normal cluster. This also affects the results of anomaly detection (AUROC and AUPR on labeled datasets), where a slight decrease in detection quality was measured. The conclusion of this experiment is that it may be relevant to use more randomized models like the Extended Isolation Forest or the modified Fair Cut Forest with random step for the task of novelty detection. Novelty detection is not the focus of this thesis or the Fair Cut Forest algorithm. This modification is just an idea how to make the model more general and usable in different scenarios.

Conclusion

The goal of this thesis was to select and implement an extension of the Isolation Forest that replaces the uniformly random selection of the split points with an optimization criteria.

The first two chapters introduced the problem of anomaly detection in general and the methods commonly used in this field. The next chapter focused on the fundamentals of the Isolation Forest algorithm. The weak spots are discussed, followed by the description of the Extended Isolation Forest, in which the splits are not made by a single feature, but by using a separating hyperplane – a combination of multiple features. Another extension is presented, being the Optimized Computational Framework for Isolation Forest (OIF), which proposes a metric to quantify the quality of the split with the idea that using splits that maximize this metric could lead to better anomaly detection performance.

The practical part of the thesis starts with the implementation of the OIF for which no publicly available implementation has been found. The method is evaluated and the influence of the hyperparameters, the split guiding metric and the method for fast search of the metric optimum is discussed. Experiments with combining the use of oblique splits from the Extended Isolation Forest with the split point guiding metric from the OIF follow. The metric was adapted to work in multiple dimensions instead of single feature, but this approach was computationally slow and the results did not show a significant improvement.

I examined and implemented two other methods, the SCiForest and the Fair Cut Forest, which work on the same principle, but the split guiding metric works on one–dimensional data, i.e. the data points projected on the separating hyperplane. This approach is more computationally efficient and results

in faster training times. These methods were evaluated along with a standard Isolation Forest, the OIF and the models from my experiments. Based on the evaluation and also its theoretical properties, the Fair Cut Forest was selected for implementation into the H2O–3 machine learning platform.

The newly implemented algorithm was able to outperform existing implementations from the H2O–3 library in an empirical evaluation on datasets commonly used to evaluate anomaly detection models. The implementation was also tested in a novelty detection scenario where it was observed that this is not the primary domain of the method. Finally, some suggestions were made on how the algorithm can be further improved in the future in various ways including speed, effective memory usage, improvements in the prediction phase and also the novelty detection performance.

Bibliography

[1] HAWKINS, Douglas M. *Identification of Outliers*. Chapman and Hall, 1980. ISBN 0-412-21900-X.

[2] BARNETT, Vic and Toby LEWIS. *Outliers in statistical data*. John Wiley & Sons Ltd, 1978. ISBN 0-471-99599-1.

[3] CHANDOLA, Varun, Arindam BANERJEE and Vipin KUMAR. Anomaly detection. *ACM Computing Surveys* [online]. 2009, **41**(3), 1-58 [cit. 2023-05-02]. ISSN 0360-0300. Available from: doi:10.1145/1541880.1541882

[4] AGGARWAL, Charu C. *Outlier Analysis*. Springer, 2016. ISBN 978-3-319-47577-6.

[5] WANG, Shanshan and Robert SERFLING. On masking and swamping robustness of leading nonparametric outlier identifiers for multivariate data. *Journal of Multivariate Analysis* [online]. 2018, **166**, pp. 32-49 [cit. 2023-05-02]. ISSN 0047259X. Available from: doi:10.1016/j.jmva.2018.02.003

[6] PIMENTEL, Marco A.F., David A. CLIFTON, Lei CLIFTON and Lionel TARASSENKO. A review of novelty detection. *Signal Processing* [online]. 2014, **99**, pp. 215-249 [cit. 2023-05-02]. ISSN 01651684. Available from: doi:10.1016/j.sigpro.2013.12.026

[7] LARSEN, Richard. J., Morris L. MARX. *An Introduction to Mathematical Statistics and Its Applications (Third Edition)*. Pearson College Div, 2000. ISBN 0-13-922303-7.

[8] MAHALANOBIS, Prasanta C. *On the generalised distance in statistics*. National Institute of Science of India, 1936.

[9] GRUBBS, E. Frank. Sample criteria for testing outlying observations. *The Annals of Mathematical Statistics*, 1950, pp. 27-58.

[10] DEAN, B. Robert and Wilfrid J. DIXON. Simplified Statistics for Small Numbers of Observations. *Analytical chemistry* [online]. 1951, **23**(4), pp. 636-638 [cit. 2023-05-02]. ISSN 0003-2700. Available from: doi:10.1021/ac60052a025

[11] ROSNER, Bernard. Percentage Points for a Generalized ESD Many-Outlier Procedure. *Technometrics* [online]. 1983, **25**(2), pp. 165-172 [cit. 2023-05-02]. ISSN 0040-1706. Available from: doi:10.1080/00401706.1983.10487848

[12] THUDUMU, Srikanth, Philip BRANCH, Jiong JIN and Jugdutt SINGH. A comprehensive survey of anomaly detection techniques for high dimensional big data. *Journal of Big Data* [online]. 2020, **7**(1) [cit. 2023-05-02]. ISSN 2196-1115. Available from: doi:10.1186/s40537-020-00320-x

[13] COVER, Thomas M. and Peter E. HART. Nearest Neighbor Pattern Classification. *IEEE transactions on information technology*, 1967, **25**(1), pp. 21-27.

[14] LI, Wenchao, Ping YI, Yue WU, Li PAN and Jianhua LI. A New Intrusion Detection System Based on KNN Classification Algorithm in Wireless Sensor Network. *Journal of Electrical and Computer Engineering* [online]. 2014, pp. 1-8 [cit. 2023-05-02]. ISSN 2090-0147. Available from: doi:10.1155/2014/240217

[15] RAMASWAMY, Sridhar, Rajeev RASTOGI and Kyuseok SHIM. Efficient algorithms for mining outliers from large data sets. *Proceedings of the 2000 ACM SIGMOD international conference on Management of data* [online]. New York, NY, USA: ACM, 2000, 2000-05-16, pp. 427-438 [cit. 2023-05-02]. ISBN 1581132174. Available from: doi:10.1145/342009.335437

[16] ZHU, Pengyun, Chaowei ZHANG, Xiaofeng LI, Jifu ZHANG and Xiao QIN. A High-dimensional Outlier Detection Approach Based on Local Coulomb Force. *IEEE Transactions on Knowledge and Data Engineering* [online]. pp. 1-1 [cit. 2023-05-02]. ISSN 1041-4347. Available from: doi:10.1109/TKDE.2022.3172167

[17] BREUNIG, Markus M., Hans-Peter KRIEGEL, Raymond T. NG and Jörg SANDER. LOF: Identifying Density-Based Local Outliers. *Proceedings of the 2000 ACM SIGMOD international conference on Management of data* [online]. New York, NY, USA: ACM, 2000, 2000-05-16, pp. 93-104 [cit. 2023-05-02]. ISBN 1581132174. Available from: doi:10.1145/342009.335388

[18] Scikit-learn. Outlier detection with Local Outlier Factor (LOF). *Scikit-learn* [online]. [cit. 2024-04-20]. Available from: `https://scikit-learn.org/stable/auto_examples/neighbors/plot_lof_outlier_detection.html`

[19] MAHTO, Paritosh. Local Outlier Factor: A way to Detect Outliers. *Medium* [online]. Sep 2020 [cit. 2023-05-02]. Available from: `https://medium.com/mlpoint/local-outlier-factor-a-way-to-detect-outliers-dde335d77e1a`

[20] ESTER, Martin, Hans-Peter KRIEGEL, Jörg SANDER and Xiaowei XU. A density-based algorithm for discovering clusters in large spatial databases with noise. *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, [online]. 1996, **96**(34), pp. 226–231 [cit. 2023-05-02]. Available from: `https://www.dbs.ifi.lmu.de/Publikationen/Papers/KDD-96.final.frame.pdf`

[21] SCHUBERT, Erich, Jörg SANDER, Martin ESTER, Hans Peter KRIEGEL and Xiaowei XU. DBSCAN Revisited, Revisited: why and how you should (still) use DBSCAN. *ACM Transactions on Database Systems* [online]. 2017, **42**(3), pp. 1-21 [cit. 2023-05-02]. ISSN 0362-5915. Available from: doi:10.1145/3068335

[22] VALETI, Dilip. DBSCAN Algorithm for Fraud Detection & Outlier Detection in a Data set *Medium* [online]. Oct 2021 [cit. 2023-05-02]. Available from: `https://medium.com/@dilip.voleti/dbscan-algorithm-for-fraud-detection-outlier-detection-in-a-data-set-60a10ad06ea8`

[23] CAMPELLO, Ricardo J. G. B., Davoud MOULAVI, Arthur ZIMEK and Jörg SANDER. Hierarchical Density Estimates for Data Clustering, Visualization, and Outlier Detection. *ACM Transactions on Knowledge Discovery from Data* [online]. 2015, **10**(1), pp. 1-51 [cit. 2023-05-02]. ISSN 1556-4681. Available from: doi:10.1145/2733381

[24] SCHÖLKOPF, Bernhard, John C. PLATT, John SHAWE-TAYLOR, Alex J. SMOLA and Robert C. WILLIAMSON. Estimating the Support of a High-Dimensional Distribution. *Neural computation* [online]. 2001, **13**(7), pp. 1443-1471 [cit. 2023-05-02]. ISSN 0899-7667. Available from: doi:10.1162/089976601750264965

[25] CORTES, Corinna and Vladimir VAPNIK. Support-vector networks. *Machine learning* [online]. 1995, **20**(3), pp. 273-297 [cit. 2023-05-02]. ISSN 0885-6125. Available from: doi:10.1007

[26] BOSER, Bernhard E., Isabelle M. GUYON and Vladimir N. VAPNIK. A training algorithm for optimal margin classifiers. *Proceedings of the fifth annual workshop on Computational learning theory* [online]. New York, NY, USA: ACM, 1992, pp. 144-152 [cit. 2023-05-04]. ISBN 089791497X. Available from: doi:10.1145/130385.130401

[27] ABDELHAMID, Djeffal, Souissi Mohamed MOUNIR and Chelouai ABDELHEY. 3D Support Vector Clustering for fingerprints segmentation *Discovery and Data Analysis (KDDA'2015)* [online]. 2015 [cit. 2023-05-02]. Available from: `https://www.researchgate.net/publication/294089330_Forest_fire_loss_estimation_using_GIS_and_data_mining_techniques_-_Case_study_The_Kroumirie_Mountains_the_North_West_of_Tunisia`

[28] TAX, David M.J. and Robert P.W. DUIN. Support Vector Data Description. *Machine learning* [online]. 2004, **54**(1), pp. 45-66 [cit. 2023-05-02]. ISSN 0885-6125. Available from: doi:10.1023/B:MACH.0000008084.60811.49

[29] SHAH-HOSSEINI, Reza, Saeid HOMAYOUNI and Abdolreza SAFARI. A Hybrid Kernel-Based Change Detection Method for Remotely Sensed Data in a Similarity Space. *Remote Sensing* [online]. 2015, **7**(10), pp. 12829-12858 [cit. 2023-05-02]. ISSN 2072-4292. Available from: doi:10.3390/rs71012829

[30] KRAMER, M.A. Autoassociative neural networks. *Computers & chemical engineering* [online]. 1992, 16(4), 313-328 [cit. 2023-05-02]. ISSN 00981354. Available from: doi:10.1016/0098-1354(92)80051-A

[31] HUANG, Peng, Jiawen SHANG, Yingjie XU, Zhihui HU, Ke ZHANG, Jianrong DAI and Hui YAN. Anomaly detection in radiotherapy plans using deep autoencoder networks. *Frontiers in Oncology* [online]. 2023, **13** [cit. 2023-05-02]. ISSN 2234-943X. Available from: doi:10.3389/fonc.2023.1142947

[32] SAKURADA, Mayu and Takehisa YAIRI. Anomaly Detection Using Autoencoders with Nonlinear Dimensionality Reduction. *Proceedings of the MLSDA 2014 2nd workshop on machine learning for sensory data analysis* [online]. New York, NY, USA: ACM, 2014, 2014-12-02, pp. 4-11 [cit. 2023-05-02]. ISBN 9781450331593. Available from: doi:10.1145/2689746.2689747

[33] LIU, Fei Tony, Kai Ming TING and Zhi-Hua ZHOU. Isolation Forest. *2008 Eighth IEEE International Conference on Data Mining* [online]. IEEE, 2008, pp. 413-422 [cit. 2023-05-02]. ISBN 978-0-7695-3502-9. Available from: doi:10.1109/ICDM.2008.17

[34] OPITZ, David and Richard MACLIN. Popular Ensemble Methods: An Empirical Study. *Journal of Artificial Intelligence Research* [online]. 1999, **11**, 169-198 [cit. 2023-05-02]. ISSN 1076-9757. Available from: doi:10.1613/jair.614

[35] KNUTH, Donald E. *The art of Computer Programming Volume 3: Sorting and Searching* 2nd ed. Upper Saddle River, NJ: Addison-Wesley, c1998. ISBN 978-0-201-89685-5.

[36] WEISSTEIN, Eric W. Euler-Mascheroni Constant. *MathWorld–A Wolfram Web Resource* [online]. [cit. 2023-05-02]. Available from: `https://mathworld.wolfram.com/Euler-MascheroniConstant.html`

[37] ANELLO, Eugenia. Anomaly Detection With Isolation Forest *Better Programming* [online]. Apr 2021 [cit. 2023-05-02]. Available from: `https://betterprogramming.pub/anomaly-detection-with-isolation-forest-e41f1f55cc6`

[38] CORTES, David. Revisiting randomized choices in isolation forests. *arXiv* [online]. 2021, [cit. 2023-05-02]. Available from: `https://arxiv.org/abs/2110.13402`

[39] HARIRI, Sahand, Matias Carrasco KIND and Robert J. BRUNNER. Extended Isolation Forest. *IEEE Transactions on Knowledge and Data Engineering* [online]. 2021, **33**(4), pp. 1479-1489 [cit. 2023-05-02]. ISSN 1041-4347. Available from: doi:10.1109/TKDE.2019.2947676

[40] WICKRAMARACHCHI D. C., B. L. ROBERTSON, M. REALE, C. J. PRINCE and J. BROWN. HHCART: An Oblique Decision Tree. *arXiv* [online]. 2015, [cit. 2023-05-02]. Available from: `https://arxiv.org/abs/1504.03415`

[41] LIU, Zhen, Xin LIU, Jin MA and Hui GAO. An Optimized Computational Framework for Isolation Forest. *Mathematical Problems in Engineering* [online]. 2018, pp. 1-13 [cit. 2023-05-02]. ISSN 1024-123X. Available from: doi:10.1155/2018/2318763

[42] HARRIS, Charles R., K. Jarrod MILLMAN, Stéfan J. VAN DER WALT, et al. Array programming with NumPy. *Nature* [online]. 2020, **585**(7825), pp. 357-362 [cit. 2023-05-02]. ISSN 0028-0836. Available from: doi:10.1038/s41586-020-2649-2

[43] RAYANA, Shebuti. ODDS Library. [online]. Stony Brook, NY: Stony Brook University, Department of Computer Science, 2016. Available from: `http://odds.cs.stonybrook.edu`

[44] DAVIS, Jesse and Mark GOADRICH. The relationship between Precision-Recall and ROC curves. *Proceedings of the 23rd international conference on Machine learning - ICML '06* [online]. New York, New York, USA: ACM Press, 2006, pp. 233-240 [cit. 2023-05-02]. ISBN 1595933832. Available from: doi:10.1145/1143844.1143874

[45] VIRTANEN, Pauli, Ralf GOMMERS, Travis E. OLIPHANT, et al. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature Methods* [online]. 2020, **17**(3), pp. 261-272 [cit. 2023-05-02]. ISSN 1548-7091. Available from: doi:10.1038/s41592-019-0686-2

[46] DEMBO, Ron S. and Trond STEIHAUG. Truncated-Newton algorithms for large-scale unconstrained optimization. *Mathematical Programming* [online]. 1983, **26**(2), pp. 190-212 [cit. 2023-05-03]. ISSN 0025-5610. Available from: doi:10.1007/BF02592055

[47] LIU, Fei Tony, Kai Ming TING and Zhi-Hua ZHOU. On Detecting Clustered Anomalies Using SCiForest. *Machine Learning and Knowledge Discovery in Databases* [online]. Springer, 2010, pp. 274-290 [cit. 2023-05-02]. Available from: doi:10.1007/978-3-642-15883-4_18

[48] CORTES, David. Imputing missing values with unsupervised random trees. *arXiv* [online]. 2019, [cit. 2023-05-02]. Available from: `https://arxiv.org/abs/1911.06646`

[49] H2O.AI. Welcome to H2O 3. *H2O documentation* [online]. 2023 [cit. 2023-05-02]. Available from: `https://docs.h2o.ai/h2o/latest-stable/h2o-docs/welcome.html`

[50] H2O.AI. Case Studies. *H2O.ai* [online]. 2023 [cit. 2023-05-02]. Available from: `https://h2o.ai/case-studies`

[51] H2O.AI. H2O Driverless AI. *H2O.ai* [online]. 2023 [cit. 2023-05-02]. Available from: `https://h2o.ai/platform/ai-cloud/make/h2o-driverless-ai`

[52] H2O.AI. H2O Architecture. *H2O documentation* [online]. 2023 [cit. 2023-05-02]. Available from: `https://docs.h2o.ai/h2o/latest-stable/h2o-docs/architecture.html`

[53] H2O.AI. Isolation Forest. *H2O documentation* [online]. 2023 [cit. 2023-05-02]. Available from: `https://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science/if.html`

[54] H2O.AI. DTree. *H2O documentation* [online]. 2023 [cit. 2023-05-02]. Available from: `https://docs.h2o.ai/h2o/latest-stable/h2o-algos/javadoc/hex/tree/DTree.html`

[55] VALENTA, Adam. *Anomaly detection using Extended Isolation Forest*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020. Available from: `https://dspace.cvut.cz/handle/10467/87988`

[56] H2O.AI. Extended Isolation Forest. *H2O documentation* [online]. 2023 [cit. 2023-05-02]. Available from: `https://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science/eif.html`

[57] H2O.AI. Frame. *H2O documentation* [online]. 2023 [cit. 2023-05-02]. Available from: `https://docs.h2o.ai/h2o/latest-stable/h2o-core/javadoc/water/fvec/Frame.html`

[58] DEAN, Jeffrey and Sanjay GHEMAWAT. MapReduce. *Communications of the ACM* [online]. 2008, **51**(1), 107-113 [cit. 2023-05-03]. ISSN 0001-0782. Available from: doi:10.1145/1327452.1327492

[59] KHAWLA, Tadist, Mrabti FATIHA, Zahi AZEDDINE and Najah SAID. A Blast implementation in Hadoop MapReduce using low cost commodity hardware. *Procedia Computer Science* [online]. 2018, **127**, pp. 69-75 [cit. 2023-05-02]. ISSN 18770509. Available from: doi:10.1016/j.procs.2018.01.099

[60] KRAMÁR, Maroš. Fair Cut Forest implementation. *H2O.ai Github repository* [online]. 2023 [cit. 2023-05-03]. `https://github.com/h2oai/h2o-3/pull/6728`

[61] VALENTA, Adam. PUBDEV-7138 – Extended Isolation Forest. *H2O.ai Github repository* [online]. 2023 [cit. 2023-05-03]. `https://github.com/h2oai/h2o-3/pull/5246`

[62] MENSI, Antonella a Manuele BICEGO. Enhanced anomaly scores for isolation forests. *Pattern Recognition* [online]. 2021, **120** [cit. 2023-05-03]. ISSN 00313203. Available from: doi:10.1016/j.patcog.2021.108115

[63] BUSCHJÄGER, Sebastian, Philipp-Jan HONYSZ a Katharina MORIK. Randomized outlier detection with trees. *International Journal of Data Science and Analytics* [online]. 2022, **13**(2), pp. 91-104 [cit. 2023-05-03]. ISSN 2364-415X. Available from: doi:10.1007/s41060-020-00238-w

Appendix A

Abbreviations

API Application programming interface

AUPR Area under precision–recall curve

AUROC Area under receiver operating characteristic curve

DBSCAN Density–Based Spatial Clustering of Applications with Noise

EIF Extended Isolation Forest

FCF Fair Cut Forest

HDBSCAN Hierarchical Density-Based Spatial Clustering of Applications with Noise

HDFS Hadoop Distributed FileSystem

kNN k–Nearest Neighbors

LOF Local Outlier Factor

MOJO Maven Old Java Object

ODDS Outlier Detection DataSets

OIF Optimized computational framework for Isolation Forest

OCSVM One–class support vector machine

PDF Probability density function

SCiForest Isolation Forest with Split–selection Criterion

SQL Structured query language

SVDD Support Vector Data Description

SVM Support vector machine

Appendix B

Contents of the attached repository

- `h2o-implementation` ..... source code of H2O FCF implementation
- `jupyter-notebooks` ..... jupyter notebooks used for evaluation
- `measurements` ..... evaluation measurements in .csv format
- `prototypes` ..... Python prototypes from chapter 4
- `text` ..... text of the thesis
  - `src` ..... LaTeX source codes of the thesis
  - `thesis.pdf` ..... the Diploma thesis in PDF format
- `readme.txt` ..... repository contents description