MARSHALL PLAN RESEARCH REPORT

Anomaly Detection in Smart Grids with Imbalanced Data Methods

performed for the

Marshallplan-Jubiläumsstiftung
Austrian Marshall Plan Foundation

in collaboration with

Bowling Green State University

Fachhochschule Salzburg University of Applied Sciences

submitted by

Christian Promper, BSc

Supervisor (BGSU):	Prof. Robert C. Green II, Ph.D.
Supervisor (FHS):	FH-Prof. DI Mag. Dr. Dominik Engel

Thalgau, Oktober 2017

ACKNOWLEDGEMENT

I would first like to thank my supervisor FH-Prof. DI Mag. Dr. Dominik Engel, head of the network technologies and security department of the information technology and systems management degree course at the Salzburg University of Applied Sciences. Without his support and faith in me the admission to the Marshallplan scholarship and as a result the semester abroad at the Bowling Green State University in the Unites States of America would not have been possible. Additionally, I am gratefully indebted for his permanent guidance and very valuable comments on this research project.

I would also like to acknowledge Prof. Robert C. Green II, Ph.D. of the Computer Science Department at the Bowling Green State University as my supervisor abroad. Beside his warm welcome in the Unites States of America and at the Bowling Green State University, the door to Prof. Robert Green's office was always open whenever I ran into a trouble spot or had a question about my research. He consistently allowed this research project to be my own work, but steered me in the right direction whenever he thought I needed it.

Finally, I must thank my family and my partner for providing me with unfailing support throughout my years of study and especially the six months abroad. This accomplishment would not have been possible without them. Also, this research project would not have been possible without the granted scholarship from the Austrian Marshall Plan Foundation. Thank you.

Christian Promper

General information

Name and surname:	Christian Promper, BSc
University:	Salzburg University of Applied Sciences
Degree Course:	Information Technology & Systems Management
Title of the Research project:	Anomaly Detection in Smart Grids with Imbalanced Data Methods
Keywords:	Smart Grid Anomaly-based Intrusion Detection Python Imbalanced Data Cost-sensitive learning Under- and Over-Sampling
Supervisor at FHS:	FH-Prof. DI Mag. Dr. Dominik Engel
Supervisor at BGSU:	Prof. Robert C. Green II, Ph.D.

Abstract

The research of anomaly-based intrusion detection within smart grids is a current topic and is investigated by many researchers. Thus, little experience is available on how to address the problem of detecting anomalies in smart grids. Another problem emerges when we try to use common approaches of pattern recognition. Through the occurring imbalance in the data distribution, which means that there are much more data instances belonging to normal behavior than to attack data, the common approaches cause a low detection rate for the minority class. Therefore, various methods to overcome this drawback will be investigated by using two different datasets. To test the performance of the investigated methods for smart grids, a three-layer hierarchical smart grid communication system using an intrusion detection system at each layer will be built. For this purpose, the two-class ADFA-LD will be used. This dataset includes contemporary attacks and is well-known for evaluating the performance of anomaly-based intrusion detection systems. Finally, the performance of common approaches is compared to the performance of the imbalanced data methods.

List of Contents

List of Abbreviations

1-NN	One-Nearest Neighbor
ACC	Accuracy
ADASYN	Adaptive Synthetic
ADFA-LD	Australian Defense Force Academy Linux Dataset
AMI	Automated Meter Infrastructure
AMR	Automated Meter Reading
AUC	Area Under Curve
C-I-A	Confidentiality-Integrity-Availability
CNN	CondensedNearestNeighbor
CSV	Comma-Separated Values
CV	Cross Validation
DDoS	Distributed Denial of Service
DoS	Denial of Service
DSL	Digital Subscriber Line
DT	Decision Tree
ENN	Edited Nearest Neighbor
FN	False Negative
FNR	False Negative Rate
FP	False Positive
FPR	False Positive Rate
FTP	File Transfer Protocol
GPRS	General Packet Radio Service
GSM	Global System for Mobile communication
HAN	Home-Area Network

HIDS	Host-Based Intrusion Detection System
ICS-PSD	Industrial Control System Power System Dataset
ICT	Information and Communication Technology
ID	Identification
IDE	Integrated Development Environment
IDS	Intrusion Detection System
IED	Intelligent Electronic Device
IEEE	Institute of Electrical and Electronics Engineers
IHT	InstanceHardnessThreshold
IP	Internet Protocol
IPv6	Internet Protocol version 6
IT	Information Technology
k-NN	k-Nearest Neighbor
KDD	Knowledge Discovery and Data Mining
LoWPAN	Low-Power Wireless Personal Area Network
M2M	Machine-to-Machine
MAC	Medium Access Control
MLP	Multilayer Perceptrons
MSE	Mean Squared Error
NAN	Neighbor-Area Network
NCR	NeighborhoodCleaningRule
NIDS	Network-Based Intrusion Detection System
OSS	OneSidedSelection
PCA	Principal Component Analysis
QDA	Quadratic Discriminant Analysis

PMU	Phasor Measurement Unit
PKI	Public Key Infrastructure
PLC	Power Line Communication
RBF	Radial Basis Function
RENN	Repeated Edited Nearest Neighbor
ROC	Receiver Operating Characteristics
ROS	RandomOverSampler
RUS	RandomUnderSampler
SCADA	Supervisory Control and Data Acquisition
SGDIDS	Smart Grid Intrusion Detection System
SMOTE	Synthetic Minority Over sampling Technique
SSH	Secure Shell
SUN	Smart Utility Network
SVM	Support Vector Machine
TCP	Transmission Control Protocol
TN	True Negative
TNR	True Negative Rate
TP	True Positive
TPR	True Positive Rate
UDP	User Datagram Protocol
WAN	Wide-Area Network
WiMAX	Worldwide Interoperability for Microwave Access
WLAN	Wireless Local Area Network

List of Figures

List of Tables

List of Listings

1 Introduction

Already over 100 years ago, the world's largest engineered system, the electric grid, was built. The electric grid consists of many different systems, components and owners but was not built for the requirements of the 21st century. Thus, the electric grid struggles with a lot of weaknesses. Since it is difficult to match the energy generation to the demand, energy utilities need to over-generate electricity to ensure a complete supply. Nonetheless, power outages can occur and these outages are usually recognized only after a customer complaint. Additionally, due to the unidirectional architecture of the grid, it is difficult to integrate renewable energy power plants (e.g., wind farms or photovoltaic systems) into the electric grid. To overcome these shortcomings, the so-called "smart grid" has emerged. Within a smart grid, intelligent communication and information systems are used, for instance, to flatten peak demands, to predict demands to balance the power generation or to transmit price information so that intelligent devices can be activated automatically. This intelligent system comprises lots of sensors and communication flows between all components, utility providers and customers of the grid. This leads to various algorithms for estimation, control and pricing. Unfortunately, through the integration of such systems a lot of vulnerabilities arise [1]. Therefore, it is suggested in [2] to use methods from data analytics to monitor the communication in a smart grid to detect potential anomalies. But since most of the data is associated to normal behavior and not to disturbances or attacks, we must deal with an imbalanced data problem. For several reasons, it is more challenging for common approaches to predict classes in imbalanced datasets. Thus, a data imbalance makes classification of either normal data or anomalies more difficult. However, methods exist to overcome these shortcomings.

Within this project, these improved methods for the classification of imbalanced data sets will be investigated comprehensively. All tests will be executed with publicly available datasets and as a 2-class classifier problem. Then, the effectiveness of imbalanced data methods is tested for an anomaly-based intrusion detection system for smart grids and is also compared to common approaches. So, the main goal is to explore the methods of classifying imbalanced data sets and then to investigate how to implement them in a smart grid anomaly detection system. Since the data in smart grids is imbalanced by nature, it is expected that the imbalanced data methods will outperform the common pattern recognition approaches.

To accomplish this, the research project will start with an introduction about the current power grid. Then, the basic concepts of a smart grid will be explained and the current grid will also be compared to a smart grid. After that, the evolution of the smart grid is presented. After a brief introduction to communication technologies, the chapter about smart grid is then finished with security concerns including a description of the intrusion detection system itself and also a concept of how to implement such a system in smart grids.

The next chapter is about pattern recognition and will explain the fundamental concepts and the general recognition process. Then, different common models and approaches for the classification task will be described. Since we must deal with imbalanced data, the different existing approaches and methods will be explained next. To conclude the chapter, the common process and metrics of evaluating the performance of a classifier will be presented. Additionally, further metrics, especially for the imbalanced data problem, will be introduced.

After this theoretical part, the used datasets will be introduced. For each dataset, the structure will be described, the source will be stated and the processing steps will be explained. Finally, former achieved performances based on a literature review will be presented. In the next chapter, the development environment, which is used for all tests, will be explained.

Now, all conditions for the tests are fulfilled (theoretical foundations, data sets and the development environment) and an extensive method screening process will be executed. The performance of the datasets will be evaluated with some single classifiers first. Then, a grid search to tune the single classifiers and some newly added ensemble learners will be executed. Next, the tuned classifiers will be used to evaluate the performance of the dataset at first without any adaptions and then with all different imbalanced data methods. The performance will then be compared among each other.

Finally, the best method will be chosen, which will be used for the implementation in the smart grid intrusion detection system. So, this research project will be completed with the creation of a prototypical smart grid intrusion detection infrastructure in which the performance of common approaches is compared to the performance of the best imbalanced data method which was chosen during the method screening process.

2 Smart grid networks

The current power grid consists of four different power layers. These layers are categorized by voltage and the type of power generation. Therefore, one can distinguish between the extra-high-voltage grid, the high-voltage grid, the medium-voltage grid and the low-voltage grid. Within the extra-high-voltage grid, large conventional power plants, large hydropower or pumped-storage plants or even large renewable energy power plants like on- or offshore wind farms generate electricity. This structure continues along with the associated size of the high-voltage and the medium-voltage grid with the exception that there are different renewable energy power plants. While the high-voltage grid receives the renewable electricity from onshore wind farms or photovoltaic power plants, the renewable energy part of the medium-voltage grid is powered by freestanding photovoltaic power plants, photovoltaic roof systems, biomass power plants and onshore wind farms, too. Finally, the low-voltage grid receives the electricity from small, decentralized power plants like cogeneration units and also from various renewable energy producers such as individual photovoltaic roof-systems on buildings or onshore wind farms as well [3] [4].

If the generated power is not consumed directly on-site, it is necessary to transmit the electricity somewhere else. If the power is not yet on extra-high-voltage level, it is important to transform the power to that level since the transmission within this voltage level has the lowest electrical losses. Basically, the higher the voltage, the lower the electrical losses. Now, the electricity is transmitted over the extra-high-voltage grid and is transformed with voltage transformation substations to the next lower voltage level. Within the high-voltage grid the electricity is either used by industries or cities with a very high power demand or is converted again to the next lower voltage level. Inside the medium-voltage grid, the electricity is either used by smaller industries or cities or is converted a last time to a lower voltage level. The low-voltage grid has the lowest voltage level and powers private households or very small companies. Altogether, each of these voltage grids are separated by voltage transformation substations and the electricity needs to be converted for each voltage shift [3]. This structure was built more than 100 years ago and was designed for a unidirectional flow only, but yet must handle bidirectional transmissions. More and more renewable energy power plants are built whereby a decentralized distribution system arises, which causes a lot of power imbalances within the current power grid [4].

Beside the distribution power imbalances within the whole power grid, an imbalance occurs also on a daily basis. While there is a very high energy demand on the morning, around noon and after work time, there is only a medium load to handle during the rest of the day. During the night, just the base load remains. This causes different demands for the grid throughout the day. Furthermore, there is also an imbalance for different seasons (e.g., there is an essential higher demand during hot summers due to electric air conditioning systems). However, only half of the power plants in the United States are generating electricity around the clock. The other half is only activated if there is a higher demand. This imbalance occurs due to an insufficient efficiency of storage systems on a large scale [5].

Additional shortcomings for the current grid are for example slow response times, a lack of control and visibility, decreasing fuel reserves and resilience problems. So, to satisfy the requirements of the 21st century and to remedy the major shortcomings of the current power grid, it is important to evolve a new, modern power grid. Consequently, so-called smart grid networks, or shortly smart grids, emerged [6].

Now, a smart grid should overcome all the shortcomings and provide full visibility and pervasive control, which is accomplished by a bidirectional communication path. This two-way communication network is realized by converging modern communication and information technologies with traditional power systems [7].

Current Power Grid	**Smart Grid**
Electromechanical	Digital
Unidirectional	Bidirectional
Centralized Generation	Decentralized Generation
Hierarchical	Network
Few Sensors	Sensors Throughout
Blind	Self-Monitoring
Manual Restoration	Self-Healing
Failures and Blackouts	Adaptive and Islanding
Manual Check/Test	Remote Check/Test
Limited Control	Pervasive Control
Few Customer Choices	Many Customer Choices

Table 1: Comparison between the current grid and a smart grid (adopted from [7])

So, a smart grid is supposed to heal itself (e.g., choose different distribution routes), to be more resilient to different system anomalies, to smoothly include renewable energy sources and to allow remote checks and tests. A comparison list between the current power grid and a modern smart grid is stated in Table 1 [7]. Now, considering an emerged smart grid, various power plants, energy storage systems, transmission and distribution substations, industries, households and various other components of a power grid are linked together with modern communication paths and sensors (illustrated in Figure 1). These connections allow interoperability between all the members of the grid [6]. Additionally, the emerge of smart grids will build an intelligence layer over the whole power grid which further enables the development of new applications and business processes [7].

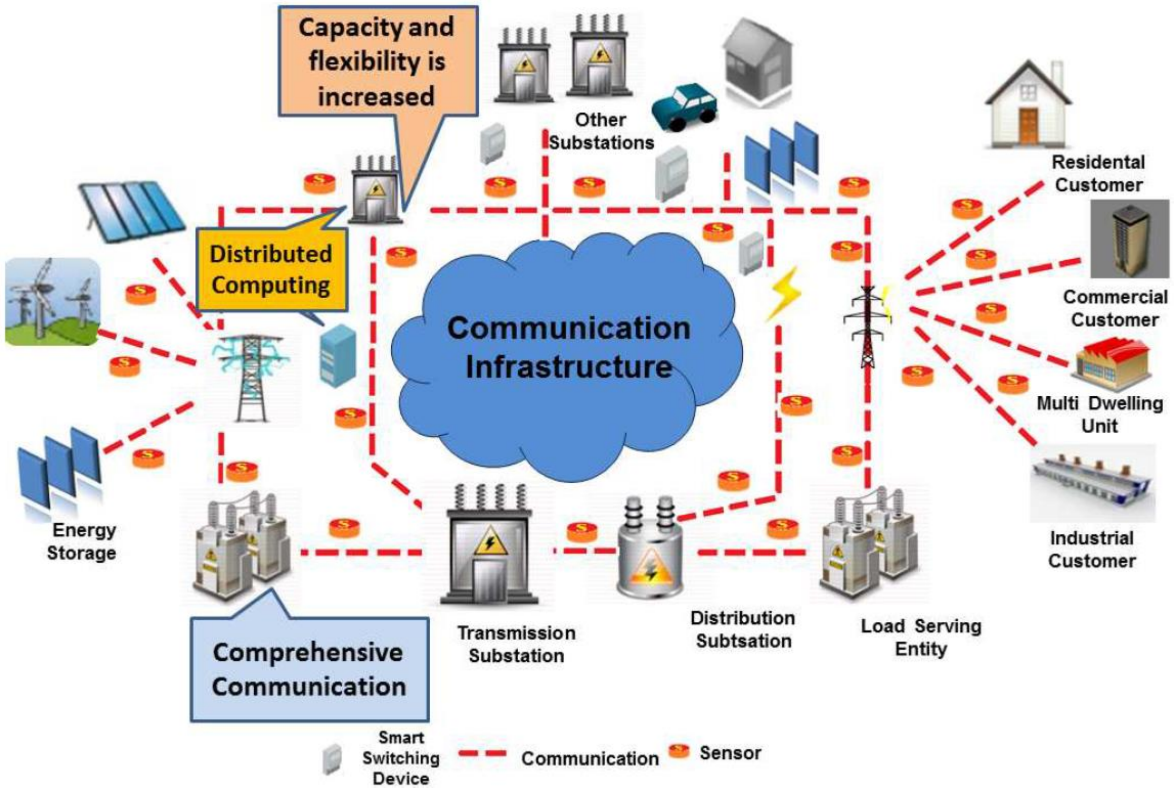


Figure 1: Smart grid sample topology according to [6]

However, the change from the current power grid to an intelligent power network cannot be done for the entire system at once. Instead, gradually strategic implementations into the existing grid across various locations are expected to arise. Then, these implementations can grow and take over more and more of the system’s load from the current power grid. All in all, there will be a coexistence between the current power grid and smart grid networks until the whole power grid network is replaced by smart devices [7].

For a more detailed explanation of the smart grid emergence, the following chapters will outline first of all the evolution of smart grids and their architecture. Then, various potential communication technologies for the data exchange within a smart grid will be introduced briefly. Afterwards, the chapter will be concluded with security concerns. More detailed, after a brief introduction in general security concerns a well-known security system is explained and considered for the smart grid domain.

2.1 Evolution and architecture of smart grids

Since most power outages (almost 90%) occur in the distribution network, the evolution of the smart grid started with improving this part of the grid. The first step was to replace the existing electromechanical meters with so-called unidirectional Automated Meter Reading (AMR) systems. AMR systems can automatically collect consumption records, alarms and status from customers and allow therefore an automated billing. However, with AMR systems, there is no demand side management possible and utilities cannot take corrective actions. Therefore, they are incapable for the usage within a smart grid. So, the next step was to use bidirectional Automated Metering Infrastructure (AMI) systems [7].

AMI systems consist of a communication network including smart meters, monitoring systems, computer hard- and software, data management systems and lots of sensors [6]. With an AMI system, it is now possible to manage the demand side. This management comprises various customer services such as variable price models, the remote addition or removal of services and also remote maintenance. Further customer benefits are, that they have more choices about services, a higher reliability, more transparency and through lower utility costs also lower energy bills. Benefits for utilities are, that they can assess equipment health, maximize asset utilization and life, optimize maintenance, pinpoint grid problems and improve grid planning [8]. However, single AMI systems need a backbone and therefore a command-and-control system. This command-and-control system must incorporate the different systems of generation, transmission and distribution since a smart grid consists of the interaction of all these different components. So, a smart grid can be simply denoted as a power grid under pervasive control of an intelligent command-and-control system with the possibility of ad hoc integrations of various components, subsystems or functions [7].

The next section will now describe the architecture of smart grids.

Architecture

It is expected, that the smart grid grows gradually and therefore a lot of independent so-called microgrids will emerge. These microgrids will contain their own control, storage, power generation and distribution systems. Thus, they can either work independently or can be linked to the entire connected smart grid network with plug-and-play integration. The interconnection of different microgrids will be performed by separate command, data and power communication lines [7].

Additionally, each smart grid or microgrid can be divided into Home-Area Networks (HAN) respectively Business-Area Networks, Neighbor-Area Networks (NAN) and Wide-Area Networks (WAN). A NAN consists of multiple HANs and a WAN consists of multiple NANs which are finally united by a utility provider through a control center (Figure 2) [1].

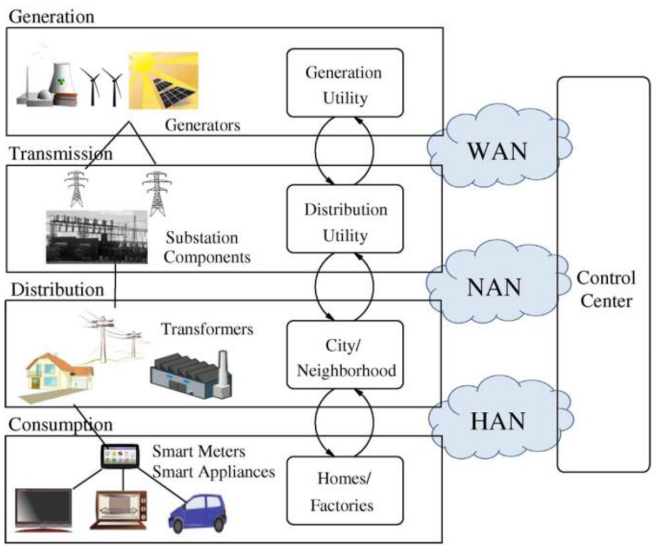


Figure 2: Smart grid architecture according to [1]

For further clarification, a HAN comprises homes or businesses. Within the boundaries of a HAN, various smart appliances (e.g., smart meters) are contained. A HAN is then connected to a NAN, which consists in general of multiple HANs. This means, multiple households and businesses are brought together within a single NAN. Additionally, other power grid components (e.g., renewable energy plants) can be incorporated with the NAN, too. Now, multiple NANs are united by a utility provided within a WAN and like a NAN, a WAN can also contain various power grid components such as power generation systems [1].

The next chapter will introduce different communication technologies to connect all the components within a smart grid.

2.2 Communication technologies

To exchange the generated data by a modern power grid (e.g., consumption info, controls, loads), a communication system is indispensable [9]. For monitoring and controlling purposes, Supervisory Control and Data Acquisition (SCADA) systems are used. But they are limited to high-voltage networks and are not suitable for large scale monitoring for the whole grid [10].

However, there are already a lot of existing technologies used in the current grid, including wired and wireless technologies. For wired data transmissions, utility providers use, for example, power line communication systems or copper-wire lines and for wireless communication they use cellular networks such as Global System for Mobile communication (GSM), General Packet Radio Service (GPRS), Worldwide Interoperability for Microwave Access (WiMAX), Wireless Local Area Network (WLAN) or Cognitive Radio [10]. But since these mediums and technologies alone are not sufficient for the use within a smart grid, Figure 3 shows an example of a hybrid smart grid architecture using various communication mediums and communication flows for the distribution domain [9].

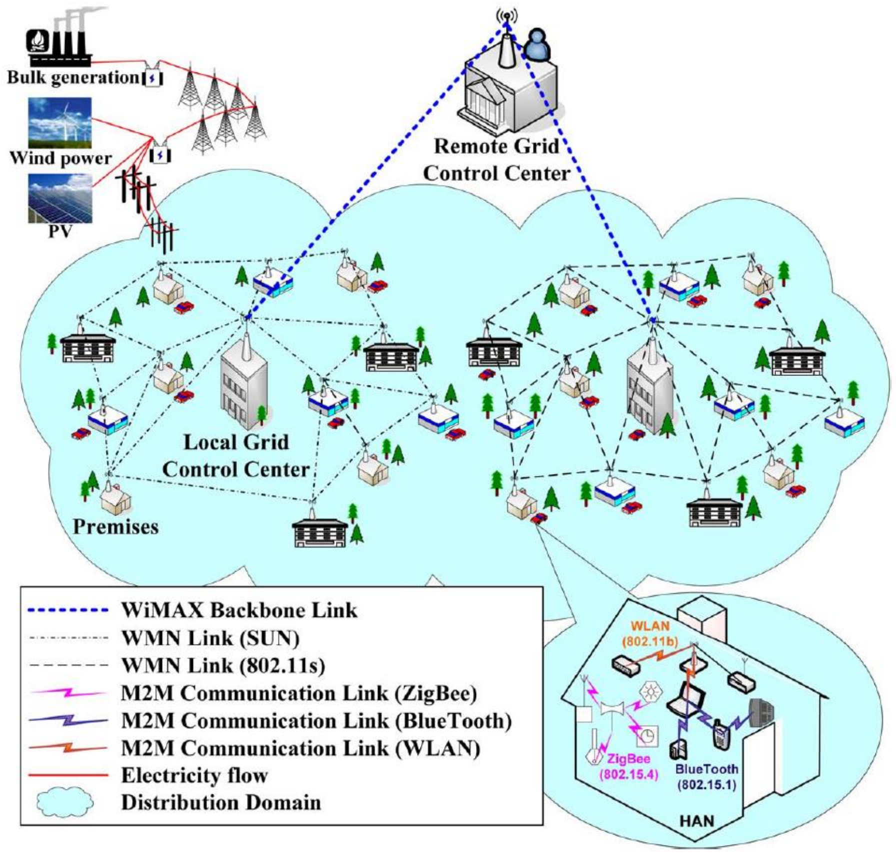


Figure 3: Hybrid smart grid network topology for the distribution domain by [9]

So, in addition to a distinction between wired and wireless communication mediums, one can distinguish also between two different communication flows. The first data flow occurs when sensors and electrical appliances send their collected data to smart meters and the second flow occurs between smart meters and electrical utilities and their data centers [6].

The different communication flows including the appropriate communication mediums and technologies for both wired and wireless transmissions will be explained subsequently in the following sections.

Communication between sensors and smart meters

As previously stated, the first data flow occurs when sensors and electrical appliances send their collected data to smart meters [6]. This flow is also denoted as Machine to Machine (M2M) communication. For example, wired power line communications (PLC) and various wireless transmission technologies are suggestions to enable M2Ms [9].

But the communication flow between sensors and smart meters in the current grid is limited since there is a missing internet protocol (IP) based network architecture. Therefore, different wireless technologies such as ZigBee, 6LoWPAN and Z-Wave as an addition to Bluetooth and WLAN were developed to connect low-power devices to the internet. Another intention of these developments was to build low-cost, reliable and scalable communication protocols [10].

Communication between smart meters and utility providers

The second flow occurs between smart meters and electrical utilities and their data centers. To transmit data between these two nodes, cellular technologies or the internet can be used [6]. The reason, why PLC is usually not used for transmissions between consumers and utilities, is that the frequency range from 10 to 20 megahertz cannot be used for a reliable communication over large distances, since feeder cables were not developed to transmit data and so they are prone to interferences [10].

Beside the use of existing technologies such as WiMAX or various cellular network standards for the communication between smart meters and the utility providers, specific mesh networks have emerged. Wireless mesh networks bridge the gap between a HAN and a WAN whereby WiMAX can be used to increase the backbone network capacity to improve the performance and to reach long distances [9].

2.3 Security

The emergence of the smart grid, containing all these previously stated different communication technologies, brings a lot of vulnerabilities along. The large scale and the complexity of a smart grid causes a lot of troubles to provide security over the network. The modern grid will be operated by many different entities and systems such as generation facilities, the distribution network and the according communication networks. Additionally, the accompanying control and pricing algorithms might also add even more vulnerabilities. The motivation to even attack smart grids might have several reasons. Maybe one wants to decrease his electricity bills, wants to play tricks on utility providers, intends to threaten people or just shows the possibility to invade a smart grid communication network [1].

Generally, the goal of information technology (IT) security is to protect assets such as financial data, hardware, software and networks from getting exploited. Therefore, each countermeasure should consider the so-called Confidentiality-Integrity-Availability (C-I-A) triangle. Confidentiality intends to protect data from third-parties so that no personal information is obtained by unauthorized users. Integrity ensures that information has not been modified during the transmission and finally availability intends to guarantee access to information whenever requested [11].

The requirements for smart grid data such as price information, meter data, control commands and software in relation to the C-I-A triangle can be found in Table 2.

	Confidentiality	**Integrity**	**Availability**
Price Information	Not critical (if public knowledge)	**Critical** (simultaneous device activations)	**Critical** (financial and legal implications)
Meter Data	Important (personal activity information)	Important (Revenue losses)	Not Critical (data can be extracted later)
Control Commands	Not critical (if public knowledge)	Important (Revenue losses)	Important (no state changes for meters)
Software	Not critical (No security through obscurity)	**Critical** (control of devices and components)	N/A

Table 2: C-I-A triangle for smart grid data and software (created from [1])

The statement in the brackets either explains the reason why a protection is unnecessary or it states the impact of a security breach. The encryption of price information (confidentiality) is not inevitably necessary since this information might be public knowledge anyway. So, protective measures for integrity (authentication) are more important than for confidentiality or availability [1].

Anyway, various threats and countermeasures exist. More information about threats and countermeasures can be found in [1]. Further, it is learned from IT security that a comprehensive security system needs to include monitoring systems, too [2]. Therefore, monitoring is considered carefully within this research project. Thus, the next section explains the concept of an intrusion detection system (IDS) and then how an intrusion detection system can be implemented into a smart grid network.

Intrusion detection

Beside a lot of entry points and attack possibilities, there are further shortcomings for typical defense mechanisms such as firewalls and antivirus software. While firewalls mostly protect against malicious packets from outside, antivirus software is always running after the newest signatures and the software also cannot avoid zero-day attacks. Therefore, an IDS can monitor all devices and the whole traffic within a network (ingoing, outgoing and communication between hosts) and is also able to avoid zero-day attacks [12].

An IDS consists of one or more sensors for real-time monitoring of traffic and a management console to operate and monitor the sensors and to display warnings [12]. But, an IDS is not an active system which can stop attacks directly. In fact, it basically detects malicious traffic. After a detection, an IDS can report the attack to the management system, reconfigure network devices (e.g., a firewall) to block the malicious traffic or to send a TCP reset command to the traffic source to terminate the connection [14].

To achieve this, it is for example possible to connect an IDS sensor to a central switch as illustrated in Figure 4. Then, the switch broadcasts a copy of the passed traffic through a so-called mirror port to the sensor. Other functions are, for example, to audit configurations and vulnerabilities of a system, to assess the integrity of critical systems and files, to statistically analyze incidents of known attacks and to check the operating system. Beside the surveillance of attacks, it is also possible to monitor the compliance of policies [13].

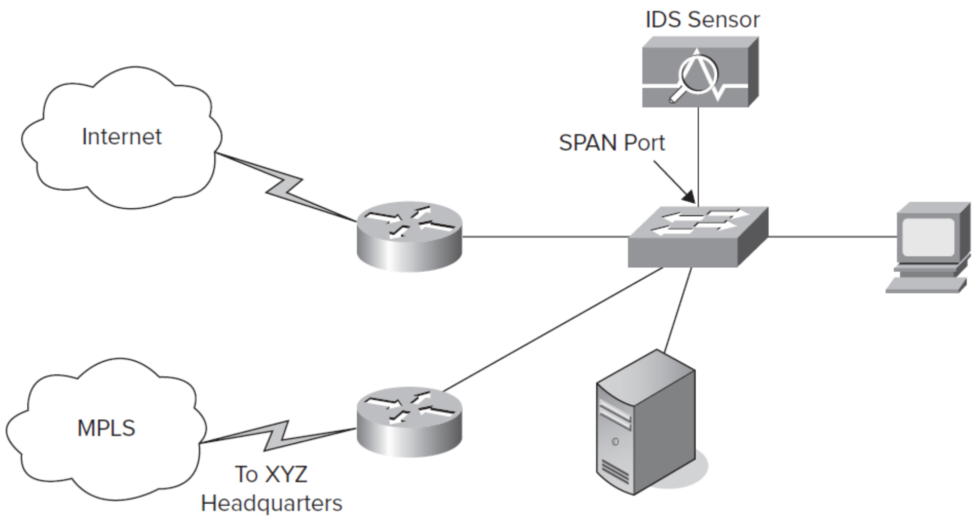


Figure 4: Intrusion detection system topology by [14]

However, there are two diverse types of IDS. One can distinguish between host-based intrusion detection system (HIDS) and network-based intrusion detection systems (NIDS). A HIDS has only a single sensor (e.g., software on a host) to monitor critical system functions on the host. The inspection is only done on system level (e.g., logs or events) and not for network packets. On the other hand, a NIDS uses one or multiple sensors spread within the network to monitor the whole network traffic. Each sensor has two interfaces whereby one interface is the management interface to control the sensor and to inspect warnings and the other interface is configured in monitor mode to capture and analyze the traffic [12] [13]. There are various pros and cons for both host and network based systems which are stated in detail in [12]. However, it is suggested to use both systems for a more comprehensive protection.

To detect anomalies at all, four different approaches exist. These approaches are signature-based detection, rule-based detection, anomaly-based detection and the usage of honeypots. Honeypots are systems with built-in vulnerabilities to lure blackhats. They can be deployed both within or outside the firewall perimeter. The reason to use honeypots is to gain information about attack methods to finally prepare the real network and systems for these attacks. A rule-based IDS will monitor the behavior and traffic for specific criteria. For example, a NIDS could monitor a special port and if a determined threshold (e.g., the quantity of scanned systems for this port) is exceeded, then the NIDS would send a warning message to the management system. It is possible to define rules for any parameter and threshold as long as the criteria meet the policies.

The signature-based approach tries to find patterns for single or multiple packets. Therefore, sequences are compared to a database with known attacks and if a sequence matches with an entry in the database then it is malicious behavior respectively malicious traffic. Although this method is very fast, the detection of zero-day attacks is not possible. So, the anomaly-based detection approach can detect zero-day attacks and learns therefor the behavior of a clean system respectively network or uses already defined rules. From now on, any deviation from this trained behavior triggers an alarm. But unfortunately, it is very difficult to define and train the normal behavior correctly [14].

Smart grid intrusion detection

To deploy an IDS within a smart grid, one must consider the requirements (e.g., encryption and real-time transmissions) and constraints (e.g., topology and bandwidth) of smart grid communication systems. These considerations can help to define impacts and limitations on functionalities and security for the communication architecture and the monitoring system. Due to the fact that an AMI will consist mainly of wireless (mesh) networks with a lot of nodes, it is on the one hand more vulnerable for network-related attacks and unauthorized physical access and on the other hand it is more difficult to monitor such topologies. Additionally, a constraint for an implementation is a high detection rate including zero-day attacks while causing only a low overhead. While both network and host-based sensors are required to monitor a whole smart grid network, host-based sensors for smart devices are still in research due to low performance devices such as smart meters. Now, it is the challenge to apply the knowledge of intrusion detection systems to smart grids to cover the related threats and yet to consider industry strengths [2].

The main limitation of a traditional IDS architecture is to make it scalable for the size of a smart grid network since the processing of the data from millions of nodes on a central system would be too inefficient. To circumvent this problem, it would be for example possible to outsource some of the processing load directly to the sensors whereby the central management station is only responsible for coordinating sensors and collecting high-level alerts. Another requirement is the robustness against failures and attacks. So, the system is supposed to operate even when a subset of sensors or the management station are unavailable or compromised. While sensors can be protected through virtualization or by using a separate hardware, the approach for management stations is to use redundant systems.

To detect compromised systems, various methods exist (e.g., a reputation system or a distributed proof system). Finally, it is suggested to use separated communication networks between sensors and management servers [2].

Existing approaches and concepts to implement an intrusion detection system within smart grids are for example a Model-Based IDS [15], a Behavior-Rule-Based IDS [16], an IDS with Domain Knowledge [17] and the Smart Grid Intrusion Detection System (SGDIDS) [18]. Since the SGDIDS is based on an anomaly-based approach, this concept will be used as reference model later. Basically, the SGDIDS works with a three-layer (HAN, NAN and WAN) network detection architecture with a top-down and vice versa communication and information flow (see Figure 5).

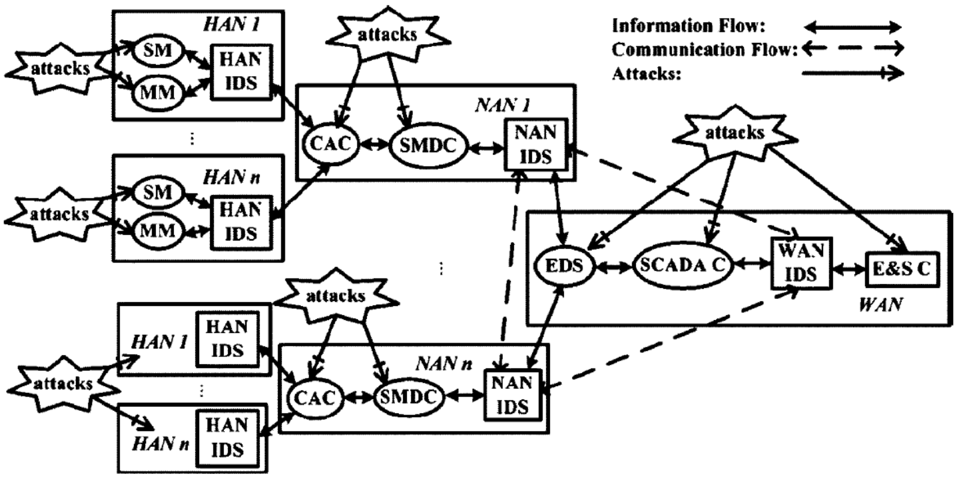


Figure 5: SGDIDS three-layer network architecture by [18]

However, this research project will investigate the anomaly-based detection approach and will therefore outline pattern recognition methods in the next chapter. This includes the pattern recognition process itself, various approaches and methods, models and classifiers and the imbalanced data problem. Then, various performance evaluation methods will be presented.

3 Pattern Recognition

Pattern recognition is the process of assigning a class, category or value to a given raw data input. While this is a naturally process for humans (e.g., face recognition), it is a sophisticated and complex task for machines [19]. In general, there are four different well-known approaches for pattern recognition. These are template matching, the statistical approach, the syntactic or structural approach and neural networks.

The template matching approach is one of the earliest approaches and is very simple. Basically, a template respectively a prototype of points, curves or shapes is available and stored. Then, an unseen pattern is compared with this template also considering translations, rotations and scale changes.

The statistical approach will be used within the scope of this research project and is based on having an amount of d features, which are represented in a d-dimensional matrix. The effectiveness of the approach is dependent on finding and extracting the correct features to distinguish between different classes, which is a very complex task. In general, it is the goal to find decision rules or boundaries within the feature space to distinguish between classes.

If more complex patterns are involved, it is more appropriate to use a syntactic approach. For this purpose, a hierarchical structure is used where a pattern consists of multiple sub patterns and each of these sub patterns consists also of multiple sub patterns and so on.

Finally, neural networks are based on the human nature and consist of a large amount of small processing stations which are interconnected. This creates the ability to build complex nonlinear relationships, to use sequential training procedures and to evolve dependent on the input data. Even if there are a lot of differences between neural networks and the statistical approach, there are also a lot of links between them with equivalent/similar methods [20].

As mentioned above, statistical pattern recognition will be used within the scope of this research project. So, the statistical approach and some of its concepts, techniques and procedures will be explained in the next chapter. Then, so-called models, which are used to build decision rules or boundaries within the feature space, will be described thoroughly. After an introduction to the imbalanced data problem, state-of-the-art solutions respectively concepts for this problem will be presented. Finally, the chapter is concluded with procedures, definitions and metrics for a comprehensive performance analysis.

3.1 Statistical pattern recognition

The whole statistical pattern recognition cycle is illustrated in Figure 6. While the recognition cycle always starts with a specific problem and the design of experiments, the next step is usually to preprocess data and to select and extract features from the experiment object. After these steps, the object is then stored as vector which equals the best possible representation for other processes within the cycle. Now, depending on which recognition problem is present, there are different learning techniques to use, namely supervised learning, unsupervised learning and reinforcement learning. All these learning techniques have different approaches and methods to solve a given problem. Then, after completing the recognition process, the performance of the recognition is finally evaluated [21].

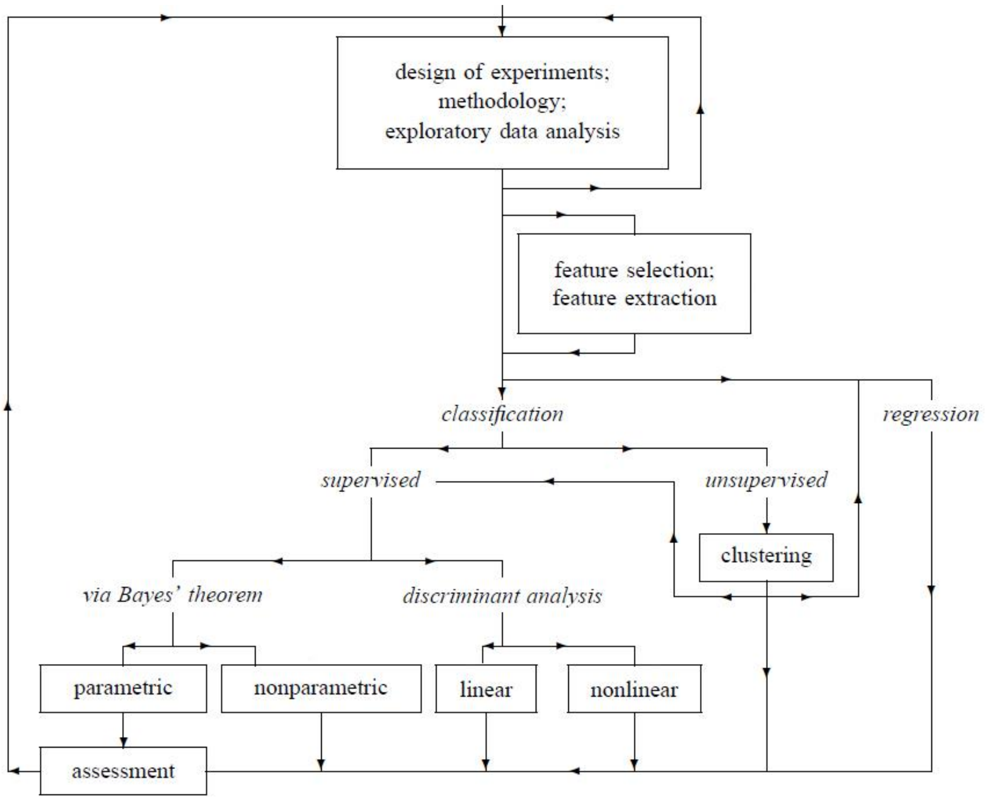


Figure 6: Statistical pattern recognition cycle by [21]

First of all, the different learning approaches will be explained briefly since this is a crucial distinction how a pattern recognition problem will be solved. Afterwards, an example based on one of these approaches will be presented since this approach is representative for a common pattern recognition problem. This approach will be used within the scope of this research project, too. Finally, the methods used within this approach and the mentioned preprocessing and feature selection methods will be explained to complete this subchapter.

3.1.1 Learning techniques

As mentioned above, pattern recognition can be distinguished by three different learning techniques. The first learning technique is called *supervised learning*. Basically, if a given recognition problem has a training dataset with assigned targets (correct labels of the related classes), this problem falls into the category of supervised learning. Furthermore, one can distinguish between two subcategories of supervised learning, namely *classification* and *regression*. We talk about a classification problem, if only a finite number of categories to classify (e.g., negatives/positives) is given. On the other hand, if the output is a continuous variable (e.g., the age including decimals), it is considered a regression problem. If the training data do not have assigned targets, then it is about *unsupervised learning*. With unsupervised learning, one can discover groups (*clustering*), determine the distribution (*density estimation*) or project data from high dimensionality to two or three dimensions (*visualization*). The last learning technique is called *reinforcement learning* and addresses the problem of finding the best action for a given situation. The goal thereby is to interact with the environment in which a sequence of actions is available and to finally maximize a reward (e.g., a score for a game). Instead of finding a correct output, the algorithm improves itself by a process of trial and error. An example for such a problem might be a backgammon game in which a learning algorithm plays millions of games to improve its algorithm [22].

Since the intrusion detection datasets, which will be used within the scope of this research project, have correctly assigned labels and also finite distinct classes, the supervised learning and classification problem is considered for all following chapters.

3.1.2 Methods and steps for supervised learning classification

To explain the concepts of the fundamental methods for supervised classification, an example from [22] is used. This example tries to recognize handwritten digits as illustrated in Figure 7. Each digit consists of 28 x 28 pixels and is stored as a vector denoted as x. So, this vector x consists of 784 values. Based on the principle, that pattern recognition assigns a class, category or value to a given raw input, it is now the challenge to build a machine that takes the vector x as input (raw data) and then to classify the digit as output. That implies, that the machine should calculate a digit between 0 and 9 based on the given input vector x. Again, this might be easy for a human but it is a nontrivial problem for a machine due to many several types of handwriting [22].

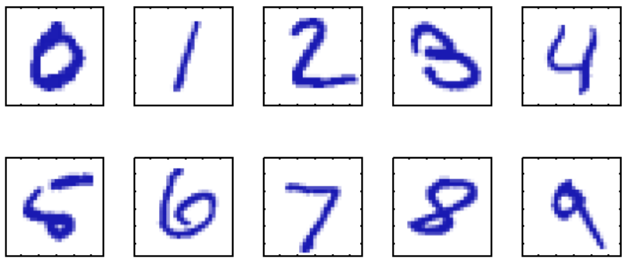


Figure 7: Hand-written digits for pattern recognition by [22]

To build such a machine model, a large set of so-called *training data* is used. This training data consists of many vectors of digits. The related digit for each vector is known previously (e.g., by inspecting and hand-labelling) and is stored as *target vector* denoted as t. So, each vector x respectively digit image has a related (correct) digit value from the target vector t assigned. Now, to assign digits to new image vectors, the machine model must be trained. This process is called *training phase* respectively *learning phase*. During this process, a function $y(x)$ is calculated based on the training data. After the learning phase, the trained model can determine digits for a given *test data* set. Now, each new digit image is processed by the trained machine model respectively the function $y(x)$ which results in a new vector with predicted labels. This predicted target vector shows the ability to classify completely unseen and different digit images. If this prediction on unseen images has a good performance, it is called *generalization*. This is the central goal in pattern recognition [22]. Even though this process covers the basic tasks for supervised classification, there are some optimization methods to improve the performance in terms of both correct recognition and computation speed, namely preprocessing and feature extraction [21].

So far, the single processes of a classification task were explained. To put all the pieces together, Figure 8 illustrates the two different mentioned recognition modes and how they do work together, to finally classify an unseen pattern. The first mode is the training (learning) mode and the second mode is the classification mode. During the training mode, the input data is preprocessed and features are extracted. Now, a machine model is trained in order to partition the feature space. Then, during the classification mode, the input data must be preprocessed exactly like the training data and the same features must be extracted. Finally, the trained machine model assigns the new unseen data to one of the given classes based on the measured features [20].

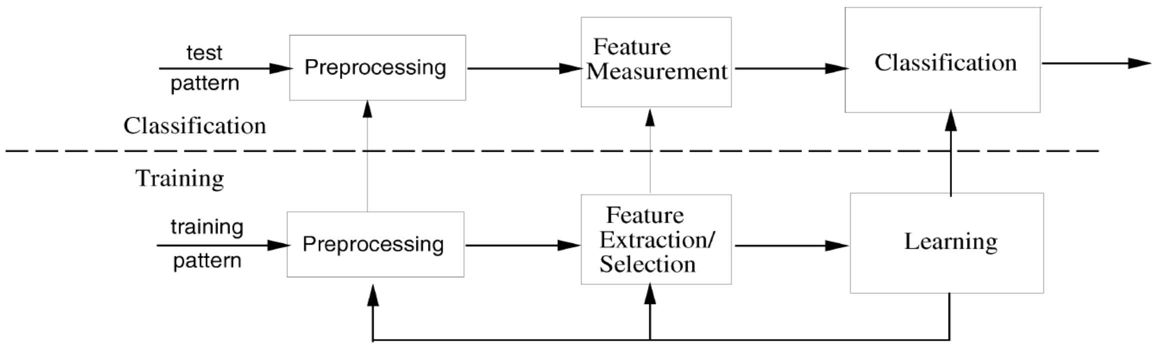


Figure 8: Statistical pattern recognition classification process by [20]

So, a classification task is accomplished in the same way all the time. Raw input data is preprocessed and features are extracted, then the data is split into training and test sets. Afterwards, a machine model is trained by the training data and finally the test data, preprocessed in the same way as the training data, is classified by the trained model [20]. All these steps will be explained subsequently. First of all, the optimization methods preprocessing and feature extraction will be explained and then two data splitting methods to create training and test data sets are introduced to conclude this subchapter. Then, the next chapter is completely dedicated to models and their tasks (learning and classification).

Preprocessing and feature extraction

To optimize the pattern recognition process, *preprocessing* and *feature extraction* might be used. Preprocessing transforms input variables into a new space of variables [22]. The reasons to use preprocessing are to reduce noisy data (outliers), to get a common resolution for images and videos, to optimize images and signals (e.g., edge detection) and to use scaling and normalization to have the data on a common range of values. In relation to scaling and normalization, the three different methods interval fit, z-score scaling (standardization) and arctangent scaling exist. The interval fit method fits the input data in each column usually to an interval [0,1] or an interval in any other range. The z-score scaling results in a zero mean and unit variance of the data while the arctangent scaling extends the standardization by putting the data in the interval [-1,1]. In that way, outliers are still within the interval range and data near the mean is scaled almost linearly [19]. Now, in relation to the digit recognition example, each image would be translated and scaled to the same format and size, which leads to easier processing and distinction through smaller variabilities. To predict unseen image vectors correctly, the test data must be preprocessed equally [22].

This recently mentioned preprocessing stage is also called feature extraction. Another example of feature extraction is related to face recognition. Since it is infeasible to process high resolution images in real-time, the process of feature extraction tries to find only features of a face which are fast to compute but preserve enough information for a correct distinction. So, instead of the whole image vector, only the extracted features are used as input data. This procedure is also related to *dimensionality reduction* [22]. However, the feature extraction process is application dependent but yet there exist a few methods which can be applied to any input vector. A popular method for independent dimensionality reduction is the principal component analysis (PCA). The goal of the PCA is to identify a smaller number of uncorrelated variables, so-called principal components. This smaller number of variables are supposed to map a maximum amount of variance of the whole input data [19].

Data splitting

Training data is used to train the model and test data is used to evaluate the performance of the model. Sometimes a third part (validation data) is extracted, too. The usage of this part will be explained in the appropriate chapter since it belongs to a specific problem.

Now, to divide the data accordingly, different splitting techniques exist. During the so-called *random sampling,* a randomly permuted data set is created. This data set serves as index array and has the same length as the input data which is supposed to get sampled. Based on a given split percentage (e.g., for the training set), this percentage of the randomly permuted array is taken and the indices are used to select the related indices from the input data for the training data. The remaining input data is used for the test data set. Optionally, a third split percentage can be used for the validation data set.

The *stratified sampling* procedure follows the exact same routine but with one exception. While the existing classes during the random sampling procedure will be split randomly (e.g., most of the digits between 0 to 4 might be in the training set while the rest of the digits remain in the test set), stratified sampling splits the amount of data instances related to the classes equally to training, test and validation data set. For example, if a stratified sampling set for the digit vectors with a 50:50 split would be created, the training set would consist of half of the amount of all digits from 0 to 9 and so would the test set, too. Nevertheless, both datasets are still randomly sampled by the randomly permuted index array [19] [22].

3.2 Models and classifiers

In another example, this time from [19], a classifier tries to distinguish between a salmon and a sea bass and therefore extracts two features for the input vector. The two-dimensional feature-space consists of the lightness and the width from both salmon and sea bass. In general, the features and so the differences between a salmon and a sea bass can be viewed as different models. It is a challenging task to find the most appropriate features to classify a class as good as possible and to have a robust, insensitive model while using a minimum of features. However, the task for a classifier is to create a decision rule or boundary to decide either if it is a salmon or a sea bass. As illustrated in Figure 9, a learned decision rule from the test data (black lines in both images) would classify a given unseen (input) feature vector as sea bass if the data point is above the decision line and as salmon if the data point is below the decision line [19]. So, a classifier is looking for a mathematical or algorithmic mapping between features and classes to create a decision rule that partitions the feature space in as many regions as classes exist [21].

For a model, there are learnt parameters during a training phase to define the boundary of the classifier. But, a classifier might also have so-called hyper-parameters which can be defined freely and used to tune a model respectively to adapt the decision boundary. To tune a model and to find the best performance, a search within the hyper-parameter space is recommended. Therefore, one approach is to define a set of various hyper-parameters and to try each possible combination. This approach is called grid search and is used commonly. The second approach is to execute a randomized search for various defined hyper-parameter to decrease the computational cost [23].

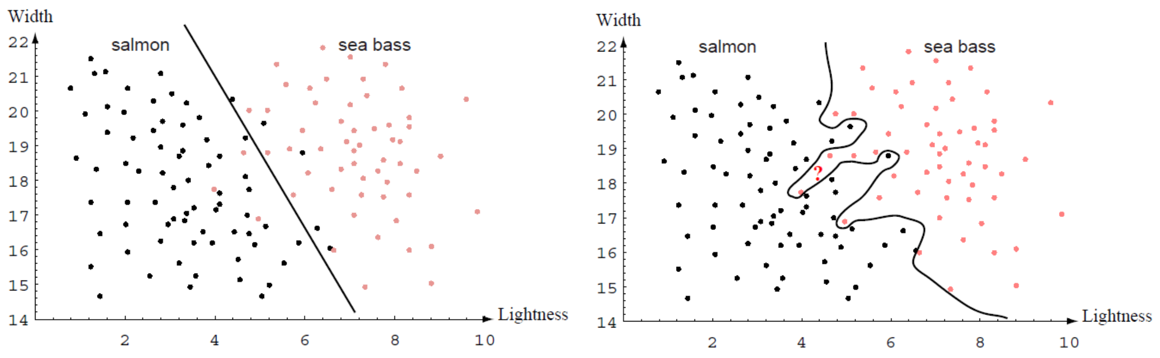


Figure 9: Model decision boundaries for a two-dimensional feature-space by [19]

However, the classifier used in the left picture of Figure 9 is very simple but this simple decision boundary will achieve a decent performance in terms of generalization. The more complex model in the right picture of Figure 9 might classify the test data perfectly but might not perform very well in terms of generalization since the decision rule is overly optimized for the given training data. This problem is known as *overfitting*. To avoid overfitting and to find a decision boundary with the ideal trade-off between performance on unseen data and simplicity of a classifier, one can split the given training set into training, test and validation data as described previously. Then, the validation data is used to compare the so-called error rate with the training data. If the error rate of the training data is very small while the error rate of the validation data is high, overfitting is indicated and the model must be adapted [19].

Now, to create a decision rule or boundary, two different approaches exist. The first approach is to use Bayes' decision rule and to estimate the class-conditional densities. The second approach uses discriminant functions for the classification. Both approaches will be explained in the following two chapters including classifier examples.

Additionally, a third subchapter describes different approaches how to combine different classifiers with the aim to improve the overall performance.

3.2.1 Bayes decision rule (stochastic)

Based on a probabilistic approach, the optimal Bayes decision rule assigns a pattern to the class with the highest posteriori probability. To calculate the posteriori probability, it is necessary to have knowledge about the probability density function of each class. Since in most cases the real probability density function is unknown, one can build an estimated density function based on a given training data set. These estimated densities are either parametric or non-parametric. Commonly used parameter models are multivariate Gaussian distributions for continuous features, binomial distributions for binary features and multinormal distributions for integer-valued and categorical features. Commonly used non-parameter models are the k-nearest neighbor (k-NN) rule and the Parzen classifier. While the k-NN rule operates similar to the one-nearest neighbor decision rule (explained in the following chapter), the Parzen classifier replaces the class-conditional densities by estimates using the so-called Parzen window approach. Both classifiers need to calculate the distance of an unseen pattern to all patterns within the training set to make a decision [20].

Then the goal is to build decision rules with the motive to either minimize the average error (minimum error) or to minimize the average cost of classification (minimum risk) [21].

Rather than using probabilities to create decision boundaries, the next chapter will use discriminant function to determine decision rules.

3.2.2 Discriminant functions (deterministic)

A discriminant function with a given pattern/vector x basically leads to a classification rule. If we assume a two-class classifier problem, a discriminant function denoted as $h(x)$ and a constant/threshold k, then the incidental classification rule is

$$\begin{aligned} h(x) > \mathrm{k} \;&\Rightarrow\; \mathrm{x} \in \omega_1 \\ h(x) < \mathrm{k} \;&\Rightarrow\; \mathrm{x} \in \omega_2 \end{aligned} \tag{3.1}$$

If $h(x) = \mathrm{k}$, then the class is assigned randomly. In case of a multiclass problem with N discriminant functions $g_i(x)$, the function for the classification rule is

$$g_i(x) > g_j(\mathrm{x}) \;\Rightarrow\; \mathrm{x} \in \omega_i \quad with\, j = 1, \ldots, N\; and\; j \neq i \tag{3.2}$$

Basically, the pattern x is assigned to the class ω_i with the largest discriminate. So, the form of a discriminant function is specified and not accompanied by its distribution as described in the previous chapter. The used discriminant function might be either chosen through prior knowledge or a functional form is adapted during the training phase [26].

However, the discriminant approach can be distinguished by linear functions (e.g., minimum distance classifier, nearest neighbor rule), kernel-based approaches to build nonlinear functions (e.g., radial basis functions, support vector machines), projection-based methods to build nonlinear functions (e.g., multilayer perceptrons) and tree-based approaches [21]. So, in the following sections, linear discriminant functions, including the minimum distance classifier, the nearest neighbor rule and the k-NN rule, will be explained. After an introduction to the geometric approach, which finally leads to multilayer perceptrons, the support vector machine is described. Finally, the concept of decision trees will be presented.

Minimum distance classifier and nearest neighbor rule

A linear discriminant function divides the feature space by a hyperplane and creates convex decision regions. The orientation of the hyperplane is calculated by a weight vector and the distance from the origin is calculated by a weight threshold [21].

A piecewise linear discriminant function creates non-convex and disjoint decision region. Particular cases for a piecewise linear discriminant function are the minimum distance classifier and the nearest neighbor classifier [21]. The minimum distance classifier respectively nearest mean classifier is represented by multiple so-called prototypes. This classifier simply calculates the mean vector of all data points for each class and then each mean vector represents a single prototype. So, a new data vector is assigned to the class with the lowest Euclidean distance to a prototype respectively mean vector. While this is a very simple classifier, (learning) vector quantization and data reduction methods (e.g., editing, condensing) are more advanced techniques to calculate prototypes. The data reduction methods are used for example for the one-nearest neighbor (1-NN) classifier, which assigns an unseen data point based on the Euclidean distance (or other distance metrics) to the nearest neighbor. The previously introduced k-NN rule considers then instead of one nearest neighbor the k nearest neighbors and assigns the pattern to the class with the highest occurrence within the k neighbors [20].

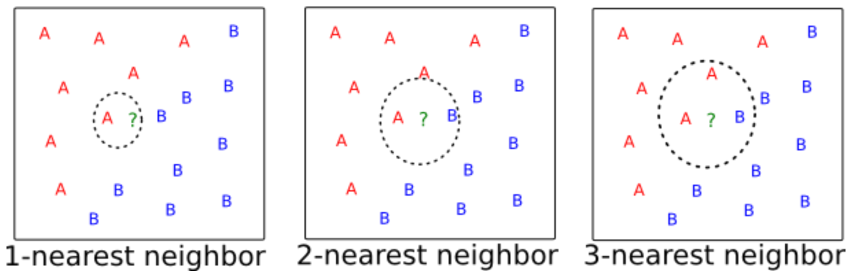


Figure 10: k-NN classifier example with k=1,2,3 (adopted from [24])

An example of the one-nearest neighbor and the two and three nearest neighbors (k=1,2,3) is illustrated in Figure 10. The one-nearest neighbor classifier (left image) decides for A, the 2-nearest neighbor classifier (middle image) decides randomly and the 3-nearest neighbor classifier (right image) decides for A since there are more A's than B's as neighbors.

Geometric approach

Now, the geometric approach tries to minimize a criterion during the training procedure. This criterion might be the classification error or the mean squared error (MSE) between classifier output and a preset target value [20].

Examples for this approach are the Fisher's linear discriminant, which minimizes the MSE between classifier output and the stated labels, and the single-layer perceptron, which iteratively updates the separating hyperplane by the distances between the misclassified patterns from the hyperplane. A similar behavior as in other linear classifiers is achieved if the MSE is implemented together with the sigmoid function. An example therefor are feed-forward neural networks respectively multilayer perceptrons (MLP). A neural network can lead to different classifiers and by including hidden layers, a neural network can also lead to nonlinear decision boundaries. In addition of classifying an input vector, the MLP classifier can also approximate the posteriori probabilities. This leads to the possibility of rejecting a pattern in case of doubt [20].

Support vector machines

A support vector machine (SVM), as illustrated in Figure 11, uses the width of the margin between two classes as optimization criterion to define the decision function. The margin is an empty area around the optimal decision hyperplane. This area is then limited by so-called support vectors (patterns) of each class, which are calculated from the training data [20].

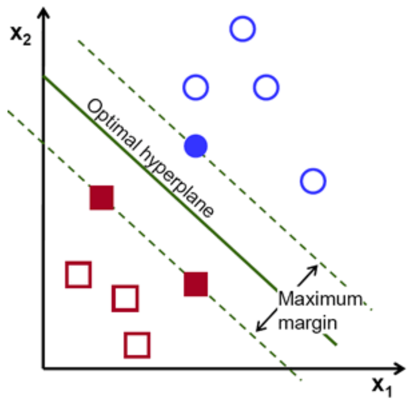


Figure 11: Support vector machine example by [25]

To accomplish this and to classify unseen patterns, it is necessary for the decision rule to use a so-called kernel function. The kernel function is a similarity function to find the similarity between two inputs. The simplest form (linear kernel) is just the dot product between the unseen pattern and a support (vector) set. On the other hand, nonlinear kernels consist of polynomial classifier (e.g., the dot product plus one and squared) or gaussian radial basis functions (RBF kernel). Now a SVM provides the ability to train even a small training data set with a high dimensionality space with good generalization. On the other hand, for large training sets only a small support set is selected to minimize the computational power [20].

Decision trees

Finally, tree-based methods or specifically a decision tree is represented by the most striking features of the whole feature space. The training phase comprises an iterative selection of these features. For the tree generation and as criteria for the feature selection the information content, the node purity or Fisher's criterion are considered. This implies, that feature selection is explicitly built-in for tree based methods [20]. An example of a decision tree is illustrated in Figure 12. The figure shows a common binary tree, where each *node* (circle) is a single feature with a variable/threshold. The decision process itself is based on multiple stages until a final *leaf* (square), which represents the decided class label, is reached [21].

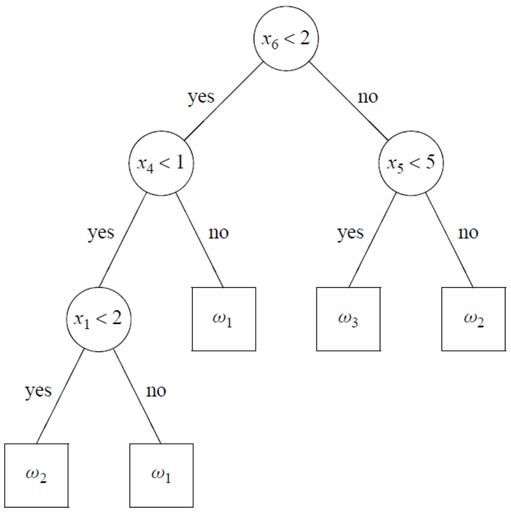


Figure 12: Decision tree example by [21]

Now, let us assume, that we have built a decision tree similarly to Figure 12 and we want to classify a new pattern $x = (5,4,6,2,2,3)$. So, we start at the root node. This node decides based on the feature number 6 and a threshold smaller than 2. Since the feature number 6 from our pattern is the value 3, the condition (< 2) is false and so we continue with the right child, which is another node. The decision for this node is based on a threshold smaller than 5 and the feature number 5. The value of feature 5 in our pattern is 2. Thus, we proceed with the left child and have finally reached a leaf node. This node assigns our pattern to class 3. This example simplifies a complex procedure but basically it explains the main concept, which is to break up the final decision in multiple smaller and simpler decisions [21].

After the explanations, how decision boundaries with probabilities and decision rules with discriminant functions are created, the next chapter will explain various methods how single classifiers can be combined to improve the overall performance.

3.2.3 Ensemble methods

Ensemble methods combine multiple so-called base learners to a single decision rule, which generalization ability is often much better than the generalization ability of a single classifier. Another great advantage of ensemble methods is, that they can boost weak learners (slightly better than a random guess) to strong learners for accurate predictions. A common architecture for ensemble methods is illustrated in Figure 13.

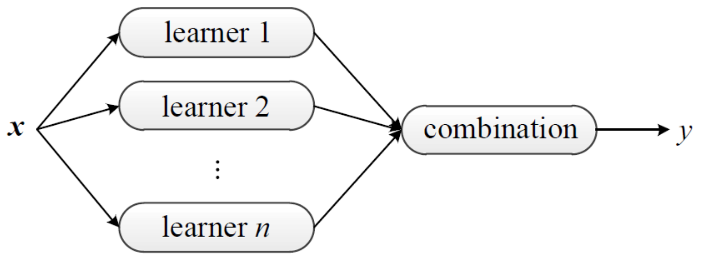


Figure 13: Common architecture for ensemble methods by [26]

Basically, each single base learner is an algorithm which is created based on the training of the training data by a k-NN, MLP, SVM, decision tree or any other classifier. While most ensemble methods use homogenous base learners (e.g., only decision trees), heterogenous ensembles consist of different basic learner types [26].

However, state-of-the-art ensemble learning approaches are *boosting* and *bagging*. These approaches are directly related to the previously described combination of weak learners to build a strong single learner. More detailed, boosting is a sequential ensemble method (sequential generation of base learners) and bagging is a parallel ensemble method (parallel generation of base learners). Another approach is the combination of various classifiers through averaging or voting [26].

The different approaches for weak learners, namely boosting, bagging and the combination techniques, will be explained subsequently. Additionally, state-of-the-art classifiers will be introduced in each category.

Boosting

To combine multiple weak learners, the general boosting algorithm changes the distribution of a training set iteratively. Let us assume, that a data space x is composed of three parts x_1, x_2 and x_3 whereby the data is equally distributed and the training set is drawn randomly by a distribution D. Since we have only one weak learner available, the goal is to generate a strong learner [26].

So, this weak learner, denoted as h_1, is trained by the distribution D and has correct classifications for x_1 and x_2 but wrong classifications for x_3 (classification error of 1/3). Now, to correct the error from h_1, boosting derives a new distribution D' from D which is more focused on the data from x_3. Then, our weak classifier, now denoted as h_2, is trained by the distribution D'. This classifier has accurate classifications for x_1 and x_3 but wrong classifications for x_2. It would be possible to combine the classifier h_1 and h_2 at this moment. Then, the combined classifier would classify x_1 correct but might have some errors within x_2 and x_3. Therefore, another distribution D'' is derived, to focus on the errors from classifier h_2. If we train the weak learner with the new distribution, the new classifier h_3 has correct classifications for x_2 and x_3 but wrong classifications for x_3. Finally, by combining h_1, h_2 and h_3, a strong learner is created, which is able to classify all sets correctly [26].

```
Input: Sample distribution $\mathcal{D}$;
       Base learning algorithm $\mathfrak{L}$;
       Number of learning rounds $T$.
Process:
1.  $\mathcal{D}_1 = \mathcal{D}$.    % Initialize distribution
2.  for $t = 1, \ldots, T$:
3.      $h_t = \mathfrak{L}(\mathcal{D}_t)$;    % Train a weak learner from distribution $\mathcal{D}_t$
4.      $\epsilon_t = P_{\boldsymbol{x} \sim D_t}(h_t(\boldsymbol{x}) \neq f(\boldsymbol{x}))$;    % Evaluate the error of $h_t$
5.      $\mathcal{D}_{t+1} = Adjust\_Distribution(\mathcal{D}_t, \epsilon_t)$
6.  end
Output: $H(\boldsymbol{x}) = Combine\_Outputs(\{h_1(\boldsymbol{x}), \ldots, h_t(\boldsymbol{x})\})$
```

Figure 14: General boosting procedure by [26]

The previously described process is shown as pseudo code in Figure 14 whereby the function f is basically the ground truth function respectively expresses the correct labels. However, this process outlines just the basic sequence and is no real algorithm, since neither the combination of the classifier nor the adjustment of the distribution is functionally declared. So, the AdaBoost algorithm was the first classifier which instantiated this process. AdaBoost is still the most famous boosting algorithm [26].

Bagging

Beside the fact that bagging methods can achieve a good generalization performance, this method can also be used for parallel computing, which is a serious benefit nowadays. However, the name bagging is a combination of Bootstrap and AGGregatING and therefore bootstrapping and aggregation is used during the bagging algorithm [26].

The goal of bagging methods is to reduce the error by combining independent base learners. Unfortunately, one cannot obtain real independent learners since they are at least trained by the same training set. A possibility would be to sample several non-overlapped data subsets and train each learner with a different sample but since we do not have unlimited training data, this approach is more visionary. So, randomness during the learning process is used to achieve more independence between the base learners [26].

Therefore, bootstrap sampling is used to generate different subsets to train the base learners. Basically, through sampling with replacement, which is executed T times, a dataset with m instances leads to T subsets with also m instances. So, some subsets might contain double entries of instances but might not contain other instances at all. Finally, the base learners are trained by these sampled subsets and the aggregating strategy voting is used to classify a test pattern (averaging is used for regression problems). Therefore, the output of each base learner is collected and the label with the highest occurrence is predicted. In case of the same amount of votes the label is chosen randomly [26].

The described algorithm is described as pseudo code in Figure 15.

```
Input: Data set $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_m, y_m)\}$;
       Base learning algorithm $\mathfrak{L}$;
       Number of base learners $T$.
Process:
1. for $t = 1, \ldots, T$:
2.     $h_t = \mathfrak{L}(D, \mathcal{D}_{bs})$  % $\mathcal{D}_{bs}$ is the bootstrap distribution
3. end
Output: $H(\boldsymbol{x}) = \underset{y \in \mathcal{Y}}{\arg\max} \sum_{t=1}^{T} \mathbb{I}(h_t(\boldsymbol{x}) = y)$
```

Figure 15: General bagging procedure by [26]

Bagging can be used for each classifier but is very popular for decision trees since a more flexible decision boundary can be created and a satisfactory performance is achieved. However, the so-called random forest ensemble is the state-of-the-art method, which is an extension of bagging but is also based on decision trees [26].

Combination methods

To combine different outputs, the methods averaging, stacking and voting exist. While averaging is used for regression problems, stacking can be considered as general framework for the generalization of many ensemble methods or as specific combination method [26].

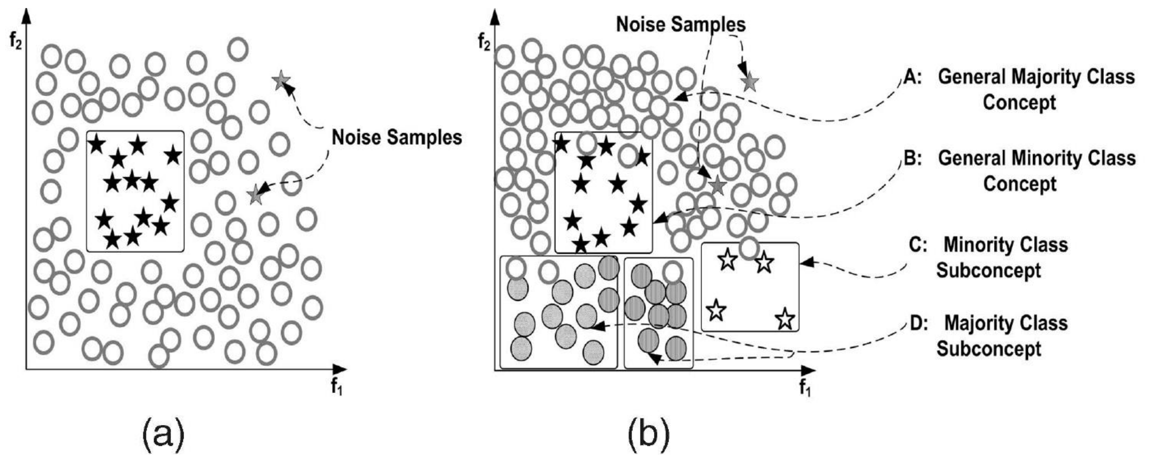
The preferred combination method for classification problems is voting. Voting is used to combine multiple label outputs to a single decision label. Additionally, voting can also be used to make a final decision for probability outputs. Basically, a voting classifier takes multiple inputs (nominal or probability outputs from single classifiers) and creates a final decision through voting. Therefore, the four different voting types majority voting, plurality voting, weighted voting and soft voting exist. As the most popular voting method, majority voting simply counts the occurrences of each class label and the class with more than half of the votes is predicted. In case that no class gets more than half of the votes, the majority voting method makes no decision (rejection). On the other hand, the plurality voting predicts simply the class with the most votes. In case of a tie, the class is chosen randomly. If we assume that the classifiers for the voting have different performances, the weighted voting might be used since this voting classifier assigns more power to better classifiers. Therefore, weights based on the performance for each classifier are calculated and then this weight is multiplied with the vote. Finally, the class with the highest value is predicted. If a classier generates a probability output, soft voting is the way to go. Again, if all classifiers have equal performances, the simple soft voting classifier averages all individual outputs. If the outputs should be combined with different weights, weighted soft voting is used [26].

After the introduction of ensemble methods by concluding the chapter about models and classifiers, the next chapter will introduce the imbalanced data problem. After a definition of the problem, various methods to optimize the performance for pattern recognition with imbalanced data will be shown.

3.3 The imbalanced data problem

Basically, an imbalanced data problem means that a dataset has an unequal distribution between the classes. The issue thereby is to achieve the same performance as for balanced datasets since common algorithms or classifiers are only optimized for balanced datasets or equal misclassification costs. In case of a two-class problem, an imbalanced data problem means that one class has significantly more instances than the other class. This is known as between-class imbalance. As example, we consider the real-world medicine problem of detecting cancer with the two occurring classes healthy (negative) and cancerous (positive). This domain has imbalanced data in its nature since more healthy than cancerous patients exist [27].

For example, the real-world “Mammography Data Set” contains 10.923 negative examples, denoted as the majority class, and only 260 positive examples, denoted as minority class. Usually, we want to achieve a high classification performance for both classes. But with such a dataset the performance might be very high for the majority class (close to 100% correct classifications) and tends to be bad for the minority class (e.g., between 0% and 10% correct classifications). This implies, that from the 260 cancerous patients 90% to 100% would be classified as healthy. Since within the medical domain it is costlier to classify a cancerous patient as healthy than vice versa, it is important to improve the accuracy of the minority class. This problem can be assigned to many other domains such as fraud detection or network intrusion. However, it is furthermore important to distinguish between relative imbalance and imbalance due to rare instances respectively absolute rarity. If we assume that the mammography dataset would consist of 100,000 majority instances and 1,000 minority instances, the relative imbalance might be high (1:100) but the minority class is with 1,000 instances not rare in an absolute perspective. While an absolute rarity might lead to a deficient performance for the minority class, some researches have shown that usual classifiers can achieve a satisfactory performance for the minority class for relative imbalances without absolute rarity. But these researches have also shown that the degree of imbalance is not the only factor which affects the performance. Instead, the most affecting factor is the complexity of the dataset and the addition of imbalance just worsens these impacts. The complexity of a dataset comprises for example overlapping, lack of representative data and small disjuncts. To explain these problems, an example imbalanced dataset is illustrated in Figure 16 [27].



f2
f1
Noise Samples
(a)
f2
f1
Noise Samples
A: General Majority Class Concept
B: General Minority Class Concept
C: Minority Class Subconcept
D: Majority Class Subconcept
(b)

Figure 16: Illustrative imbalanced datasets by [27]

While the stars represent the minority class, the circles represent the majority class. Both datasets consist of a relative imbalance. While Figure 16a has no overlapping instances and only a single applied concept for each class, Figure 16b has overlapping instances and multiple applied concepts for each class. Figure 16b shows also the difference between relative imbalance (all stars vs. all circles) and absolute rarity (sub-concept C with a lack of representative data). Additionally, Figure 16a illustrates the previously described between-class imbalance and Figure 16b introduces now the within-class imbalance. A within-class imbalance means that the representative data of a single class is distributed unequally over multiple (sub)concepts. So, cluster B within Figure 16b represents the dominant part (main concept) of the minority class while cluster C represents a sub-concept of the minority class. Since the sub-concept contains less instances than the main concept, this is called within-class imbalance. Cluster D represents two sub-concepts of the majority class and concept A represents the main concept of the majority class. Again, the sub-concepts contain less instances than the main concept and so also the majority class has a within-class imbalance [27].

Now, a lot of different solutions to address the imbalance data problem exist. They can be categorized by data-level solutions, algorithm-level solutions and finally ensemble solutions [28]. These different approaches will be explained subsequently.

3.3.1 Data-level solutions

The approach for data-level solutions is to change the distribution of an imbalanced dataset to build a (more) balanced set. A sampled dataset is then used for the learning procedure and then the classifier might achieve better classification results. In a lot of studies, it was proved that some classifiers achieved a better overall performance with a sampled and (more) balanced dataset [27].

In general, one can distinguish between under-sampling and over-sampling. While under-sampling removes data from the majority class in the original imbalanced data set, the over-sampling algorithm adds data to the minority class. So, it seems that both sampling techniques achieve the same result, namely change the imbalanced dataset to a more balanced set. However, both sampling techniques have their own pros and cons. While under-sampling might lose some important concepts through removing instances from the majority class, over-sampling might lead to overfitting since some instances might be simply duplicated through the randomness [27].

Therefore, some intelligent approaches exist to overcome these shortcomings. In addition to various under- and over-sampling approaches, which will be explained subsequently, hybrid-sampling approaches exist, too. Basically, hybrid sampling combines under- and over-sampling in diverse ways to improve the performance [27].

Under-sampling

Different approaches how to apply under-sampling will be explained in written form and also graphically. First, the RandomUnderSampler (RUS) simply chooses and removes majority samples randomly until the classes are balanced [29]. An example is shown in Figure 17.

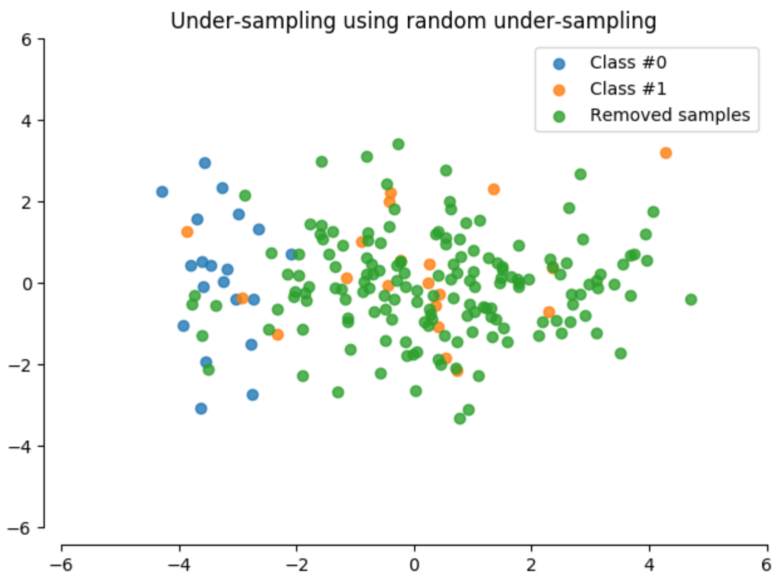


Figure 17: Under-sampling example using RandomUnderSampling by [30]

To start with the more intelligent approaches, the CondensedNearestNeighbour (CNN) is based on the nearest neighbor rule. The basic concept for the nearest neighbor rule was introduced previously. However, a shortcoming of this method is that the classifier must store all training instances. So, CNN under-sampling is an improved method of the NN-rule which needs finally less space for storing [29].

An example is shown in Figure 18.

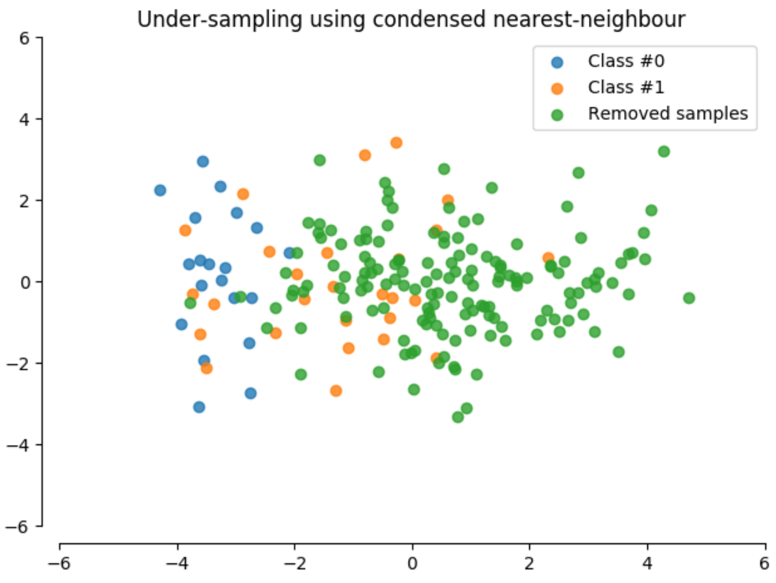


Figure 18: Under-sampling example using CondensedNearestNeighbour by [30]

The next three under-sampling techniques, namely EditedNearestNeighbours (ENN), RepeatedEditedNearestNeighbours (RENN) and All-KNN, are all quite similar and are shown together in Figure 19.

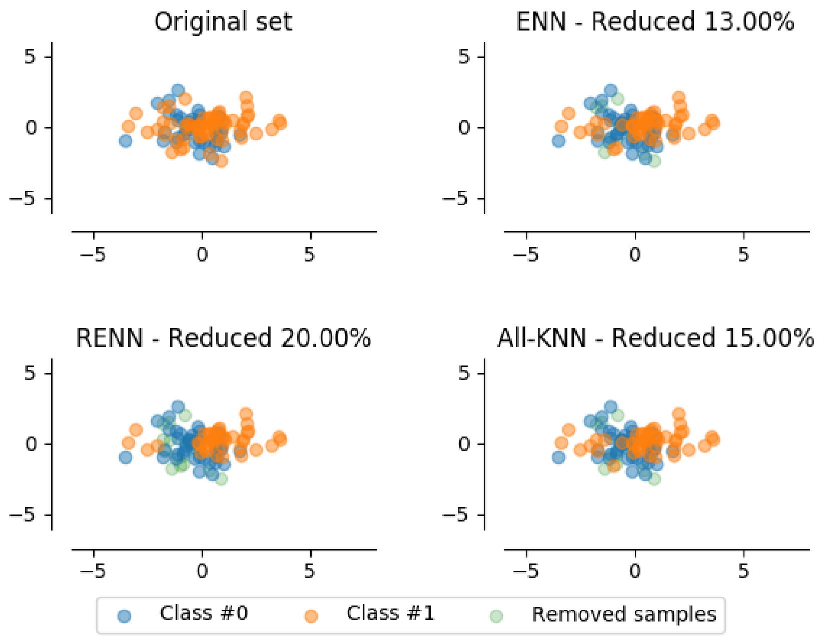


Figure 19: Under-sampling example using ENN, RENN and All-KNN by [30]

ENN is based on the k-nearest neighbor rule. Basically, under-sampling performed by ENN creates a more balanced data set distribution by accepting only instances which were correctly classified by the k-NN rule [29].

RENN works identically as ENN. The only change is that the process of removing wrongly classified instances is repeated infinite times respectively as long as no more eliminations are possible. However, this method has no proof of performance improvement in comparison to the ENN under-sampling [29]. The next under-sampling approach, All-KNN, iterates from $k = 1$ to n over a given distribution of a dataset. For each round, the k-nearest neighbors are calculated and then each instance within the distribution is classified. If most of the n predictions for an instance are wrong, then this instance will be removed [31].

The InstanceHardnessThreshold (IHT) assumes a value denoted as hardness for each instance within a dataset. This value indicates the probability of misclassification. Now the IHT under-sampling method comprises an algorithm to measure the hardness to filter the instances based on a given threshold [29]. Examples for various thresholds are shown in Figure 20.

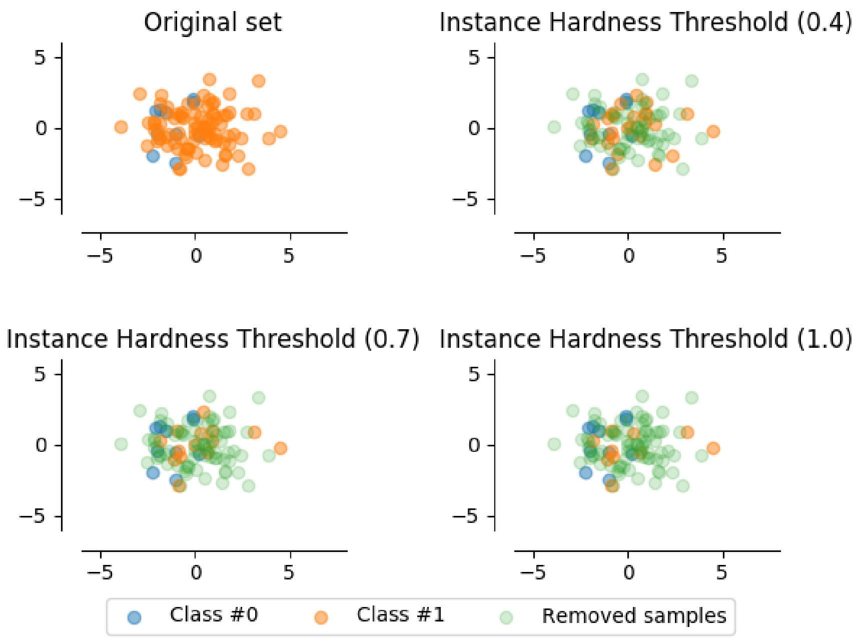


Figure 20: Under-sampling example using InstanceHardnessThreshold by [30]

NearMiss consists of three different versions, but all focus on the relation between minority and majority class. While version 1 selects instances with the lowest average distance between majority instances and three minority instances, version 2 calculates the distance to all minority instances and selects then the instances with the average distance to the three farthest minority examples. Finally, version 3 selects majority instances which are surrounded by minority instances [29]. The three versions are shown in Figure 21.

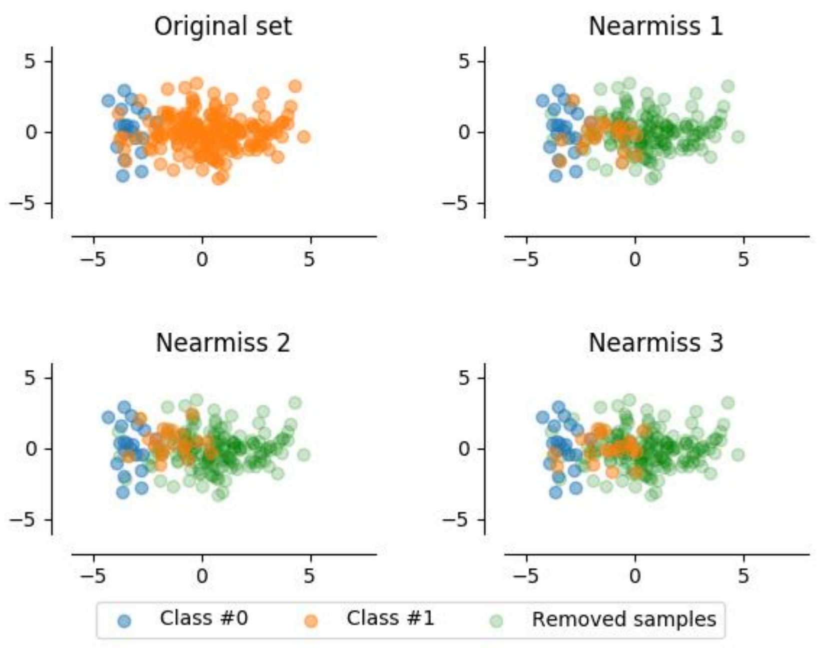


Figure 21: Under-sampling example using NearMiss version 1-3 by [30]

The under-sampling technique TomekLinks is based on CNN. Since CNN might have some shortcomings (e.g., random selection of instances at the beginning of the algorithm, which might lead to a disregarding of boundary instances), the TomekLinks algorithm uses two modifications for an increased consideration of boundary instances (Figure 22) [29].

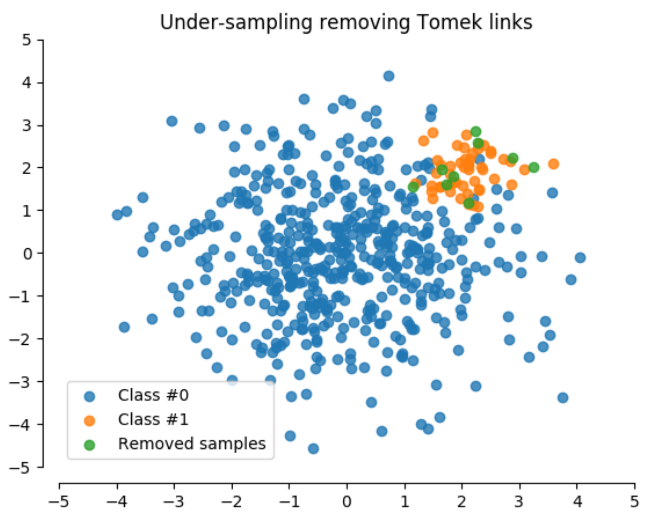


Figure 22: Under-sampling example using TomekLinks by [30]

The OneSidedSelection (OSS) method creates subsets of all minority instances and only a single majority instance. Then, the original dataset is reclassified by the 1-NN rule and the misclassifications are added to the generated subset. Finally, TomekLinks under-sampling is used to remove noisy and borderline instances of the majority class [29]. An example is shown in Figure 23.

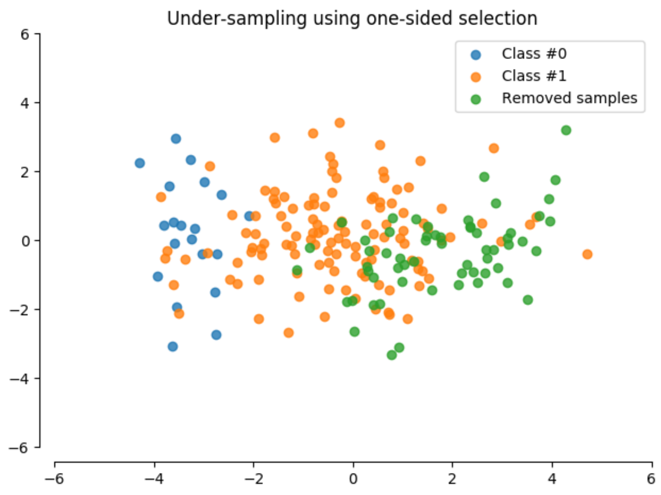


Figure 23: Under-sampling example using OneSidedSelection by [30]

Finally, the NeighbourhoodCleaningRule (NCR) works like OSS but changes the 1-NN rule since the rule might be too sensitive to noise in the data. So, NCR under-sampling uses ENN under-sampling for the majority class to remove noisy instances. Then, misclassified instances are removed from both the minority and the majority class with the 3-NN rule [29]. An example is shown in Figure 24.

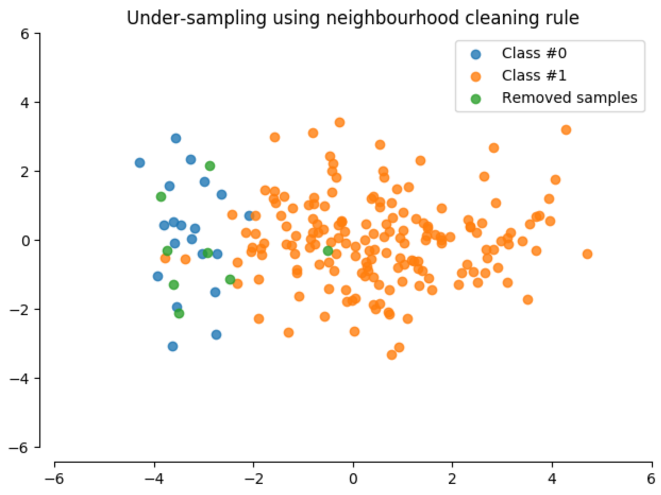


Figure 24: Under-sampling example using NeighbourhoodCleaningRule by [30]

Over-sampling

Now, various over-sampling methods will be introduced. First, the RandomOverSampler (ROS) is simply the reversed version of the RandomUnderSampler. ROS replicates minority instances randomly until the dataset is balanced [29]. An example is shown in Figure 25.

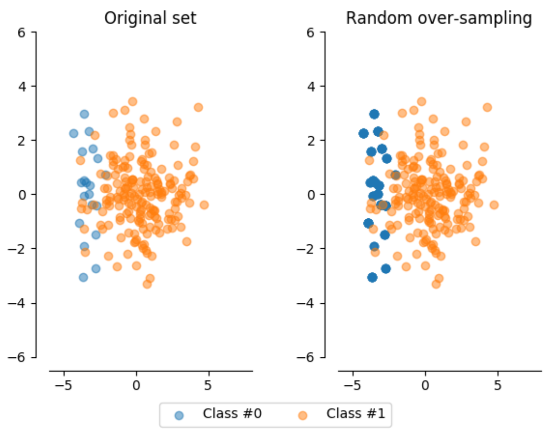


Figure 25: Over-sampling example using RandomOverSampler by [30]

The Synthetic Minority Over sampling TEchnique (SMOTE) uses synthetic instances to achieve more balance. The regular version calculates the distance between an instance and the nearest neighbor and then multiplies this distance with a random number between 0 and 1. SMOTE borderline 1 and 2 assume that borderline instances are more likely to get misclassified. Thus, they are more important and so these over-sampling methods try to synthetize only borderline instances. Finally, SMOTE SVM focuses on the borders of the minority and majority class [29]. Examples for all variations are shown in Figure 26.

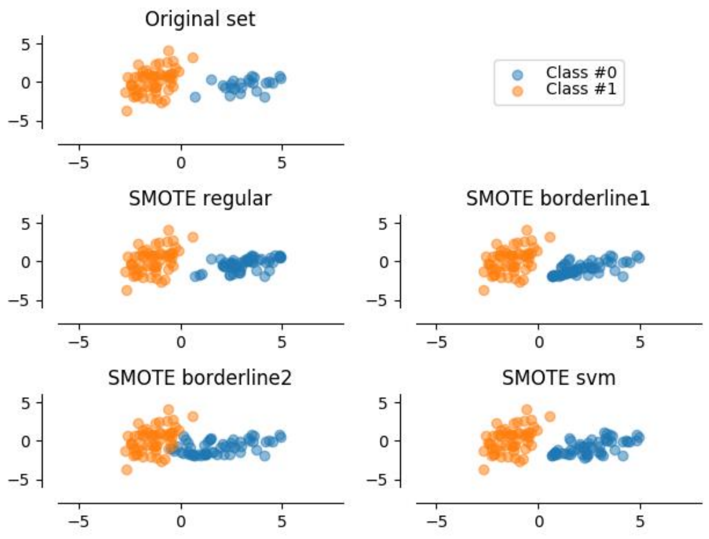


Figure 26: Over-sampling example using SMOTE by [30]

Finally, Adaptive Synthetic (ADASYN) over-sampling is based on SMOTE. The key difference is that ADASYN uses the k-nearest neighbors of an instance from the majority class and decides then, based on a weighting algorithm, how many minority instances the algorithm should synthetize. This is done with the intention to reduce bias through imbalance and to shift boundaries towards harder examples [29]. An example is shown in Figure 27.

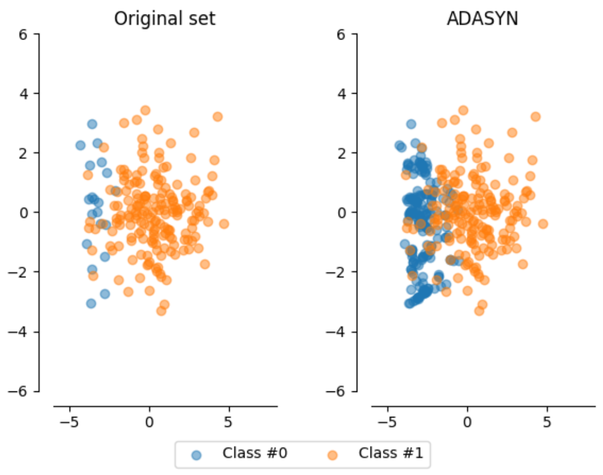


Figure 27: Over-sampling example using ADASYN by [30]

Hybrid-sampling

The last sampling category introduces two different approaches for hybrid-sampling methods. The first approach, SMOTETomek, starts with over-sampling the dataset using SMOTE and then uses Tomek to under-sample the dataset. Since both under- and over-sampling have their shortcomings, the idea is to improve the results with a combination of both methods [29]. An example is shown in Figure 28.

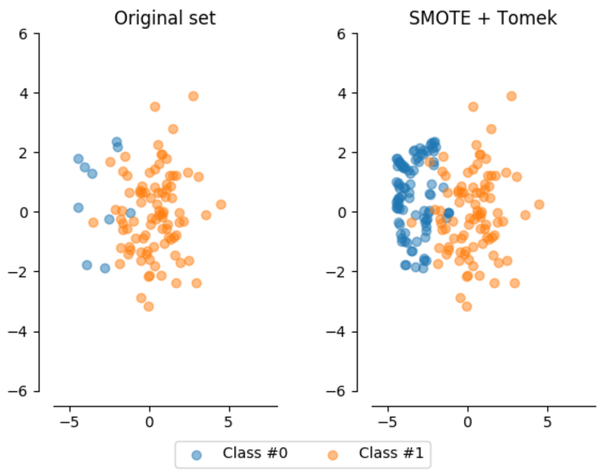


Figure 28: Hybrid-sampling example using SMOTETomek by [30]

The other hybrid-sampling approach, SMOTEENN, performs a similar procedure like SMOTETomek except using ENN to remove samples after the SMOTE over-sampling process. Since ENN is used instead of Tomek, this might lead to more removed instances which might further lead to a better performance [29]. An example is shown in Figure 28.

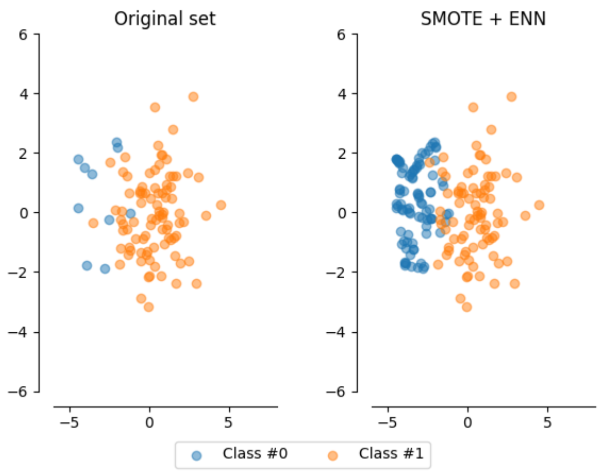


Figure 29: Hybrid-sampling example using SMOTETENN by [30]

3.3.2 Algorithm-level solutions

Even though this research project considers only cost-sensitive learning methods within the algorithm-level solution space, other approaches on the algorithm-level exist, too. These approaches are the kernel-based learning framework, one-class learning and active learning. Kernel-based methods solve a lot of today's recognition problems and therefore they are also used for imbalanced data problems. One-class learning tries to train the classifier only by the instances of a single class and active learning (usually related to unsupervised learning) uses instead of the whole dataset only a small subset for each iterative step [27] [28].

Cost-sensitive learning

As previously stated, a misclassification might be associated with different costs. Since studies showed that there is a direct connection between cost-sensitive learning and imbalances, the algorithms for cost-sensitive learning can be directly applied to imbalanced datasets without any change. To apply the following methods, a so-called cost-matrix is needed. A cost-matrix contains numerical values with costs/penalties for misclassifying a pattern. If the actual cost values are unavailable, a common way to build a cost matrix for imbalanced data problems is to assign the imbalanced ratios inversely.

This leads to costs of $C(Min, Maj)$ for misclassifying the majority class and to costs of $C(Maj, Min)$ for missclassifiying the minority class whereby in general $C(Maj, Min) > C(Min, Maj)$. However, there are usually no costs assigned for classifying a class correctly. As soon as the cost matrix is built, the goal for cost-sensitive learning is to minimize the overall costs for the training set. To achieve this, the Bayes conditional risk method is usually applied [27].

However, cost-sensitive learning can be distinguished by three distinct categories. These categories are dataspace weighting, meta-techniques and classifiers with built-in cost-sensitive functions or features.

Data-space weighting

To apply cost-sensitive learning through data-space weighting, the misclassification costs are used to change the training data distribution. This approach is strongly based on the theoretical foundations of the Translation Theorem in [32]. So, the training distribution is changed to minimize the costs and to get the best possible distribution by multiplying each case by its relative cost. This can be performed either as transparent box or black box. The transparent box passes the cost-matrix directly to the classifiers while the black box performs a re-sampling with the same cost-matrix before handing the data over to the classifier. However, this method might lead to overfitting [33].

Meta-techniques

The second category is built on theoretical foundations of the MetaCost Framework by [34]. In contrast to data-space weighting, a meta technique does not sample the data distribution and is also called non-sampling cost-sensitive meta-learning. With cost-sensitive meta-learning it is possible to convert cost-insensitive classifiers into cost-sensitive classifiers without modifying them. This is done either with pre-processing the training data or post-processing the output. However, this category can be further divided into the subcategories relabeling, weighting and threshold adjusting. The first subcategory, relabeling, changes the classes of single instances by the minimum expected cost criterium. Relabeling can be either done for the training data or the test data. The next method, weighting, basically assigns a given weight (based on the cost-matrix respectively misclassification costs) to classes and so classes with higher weights get more consideration [35].

The last method, threshold adjusting or also referred to as thresholding, investigates the output probabilities and optimizes the threshold to minimize the total misclassification costs based on a given cost-matrix. In general, the output probabilities from the training instances are used to calculate a new optimal threshold. Then, the new calculated threshold is used as decision criterion to classify the output probabilities from the test instances. If the probability of a pattern is above the new threshold, the instance is predicted as positive and if the probability is lower than the new threshold then the instance is labelled as negative. So, for all classifiers which can produce probability estimates for each instance, thresholding can be used. Additionally, this method avoids overfitting [35].

Built-in cost-sensitive functions

The last category integrates cost-sensitive learning methods directly into various classifiers. Since the way how functions are integrated or features are changed are very different, no unifying framework is available [27].

Classifiers used for such modifications are for example decision trees [27] [36], neural networks [27], random forests [36], bagging classifiers [36], pasting classifiers [36] and random patches classifiers [36].

3.3.3 Ensemble solutions

The following stated methods and classifiers will not be used within this research project. Nevertheless, they are added to the picture for completeness. In general, ensemble solutions combine either data-level or algorithm-level solutions with ensemble learning and can be distinguished by the four types bagging, boosting, random forests and hybrids. Other ensemble solutions are based on ensemble selection or ensemble pruning [28].

Bagging-based ensembles

For this approach, sampling techniques are combined with bagging. So, the training instances are sampled in several ways and then they are used to train the single learners of the ensemble method. UnderBagging and OverBagging are examples which use either random under- or over-sampling respectively to balance the dataset for the learners. SMOTEBagging on the other hand uses a bootstrap sample from the majority class and a sample of the minority class created through SMOTE over-sampling [28].

Boosting-based ensembles

Boosting-based ensembles combine either preprocessing or cost-sensitive learning with a boosting procedure. While cost-sensitive boosting keeps the general learning framework of boosting but includes weights within the weighing procedure, preprocessing in combination with boosting exploits the sampling method before the classifier generation step. Examples for cost-sensitive boosting ensembles are AdaC1, AdaC2, AdaC3, CSB1 and CBS2. They only differ in the way how they change the weighting procedure. On the other hand, SMOTEBoost, which combines SMOTE over-sampling and boosting, and RUSBoost, which combines random under-sampling and boosting, are examples for data-level-based boosting ensembles [28].

Random forest based ensembles

Different adapted random forest ensemble methods exist for both sampling and cost-sensitive learning. For example, the balanced random forest draws randomly the same number of (under-sampled) instances of the minority and majority class for each reputation of the random forest approach. The weighted random forest just adds heavier penalties for processes such as node-splitting within the random forest algorithm and is therefore related to cost-sensitive learning [28].

Hybrid ensembles

The well-known EasyEnsemble and BalanceCascade approaches combine bagging and boosting for data-level solutions. EasyEnsemble uses bagging as main ensemble learning method but AdaBoost is used instead of a single classifier to train a random under-sampled balanced dataset. BalanceCascade removes majority instances in each bagging iteration after they are correctly classified by an iteratively trained AdaBoost classifier [28].

3.4 Performance analysis

To complete this chapter, different evaluation methods and performance metrics will be introduced, since we want to know how well the trained model will perform on unseen test data. In short, we want to observe the generalization ability. To do that in practice, a given dataset is split into a training set and a test set. Then, a model is trained with the training data and finally the performance is evaluated by using the test data [20]. The methods of splitting a dataset and the performance measure process will be explained subsequently.

3.4.1 Splitting a dataset

If we want to evaluate the performance of a test data set, the question is how to split a given dataset into a training and test set optimally. While a small training set would result in a not very robust classifier and a bad generalization ability, a small test set would lead to a low confidence of the evaluated performances. So, various data splitting methods exist (stated in Table 3) to overcome these problems as good as possible. Nevertheless, there is no perfect solution for splitting a dataset since different (random) splits will always lead to different performances. However, the holdout method, the leave-one-out method and the rotation method (common method) fall into the cross validation (CV) approach [20].

Splitting method	Description
Resubstitution method	All data is used for training and testing. This leads to a very optimistic estimate.
Holdout method	Half of the data is used for training and the other half is used for testing. This leads to a pessimistic estimate and different splits will lead to different estimates.
Leave-one-out method	If we assume n data instances, a classifier is trained by $n-1$ instances and then the performance is evaluated by the remaining single test instance. This procedure is repeated n times. Even if the estimate is unbiased, it has a large variance and requires a lot of computational power.
Rotation method (k-fold cross validation)	This is a happy medium between holdout method and leave-one-out method. With a defined fold size k ($1 \leq k \leq n$), the available data instances are divided into k subsets. Now $k-1$ subsets are used to train the classifier and the remaining subset is used for the performance evaluation. This procedure is then repeated k times, so each subset is used for evaluation one time.
Bootstrap method	This method creates N bootstrap sets by sampling the data with replacement. This might lead to lower variance than the leave-one-out method but is computationally even more demanding and is therefore only useful for small data sets.

Table 3: Data splitting methods (adapted from [20])

3.4.2 Performance metrics

The performance of a two-class problem can be generally represented with a so-called confusion matrix (Figure 30) including True Positives (TP), False Positives (FP), False Negatives (FN) and True Negatives (TN). Relating to Figure 30, the $\{p, n\}$ labels are the true positive and negative class labels and $\{Y, N\}$ are the predicted positive and negative class labels respectively. Finally, $\{P_c, N_c\}$ are the total positives and negatives respectively [27].

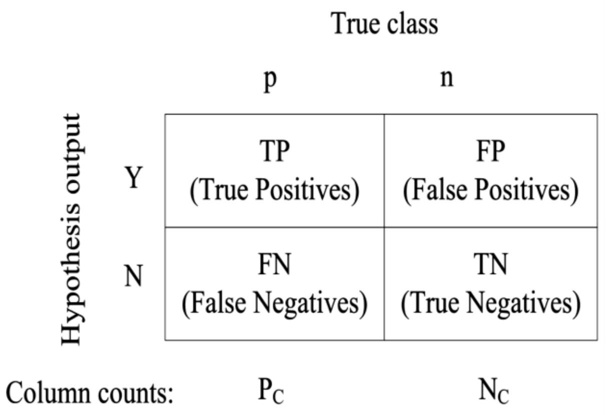


Figure 30: Confusion matrix by [27]

Let us return to the two-class problem with healthy and cancerous patients. Instead of distinguishing only between healthy or cancerous diagnosed patients, two different distinctions, namely healthy patients diagnosed as cancerous and cancerous patients diagnosed as healthy, will be of interest. Now, let us denote healthy as negatives and cancerous as positives. If a healthy patient (negative class) is diagnosed correctly ($\{n, N\}$), then it is a TN. If a cancerous patient (positive class) is diagnosed correctly ($\{p, Y\}$), then it is a TP. If a healthy patient is diagnosed as cancerous ($\{n, Y\}$), then it is a FP. Finally, if a cancerous patient is diagnosed as healthy ($\{p, N\}$), then it is a FN. In case of intrusion detection, it is usual to denote normal behavior as negatives and intrusions as positives, so this example is directly adaptable.

Metrics

The most common metrics to evaluate the performance are the accuracy (ACC) and the error rate. While the error rate is just $1 - ACC$, the accuracy is expressed as

$$ACC = \frac{TP + TN}{TP + FN + TN + FP} \tag{3.3}$$

Basically, the accuracy equation (3.3) expresses the correct classification rate over all instances and is calculated by adding up all correct classifications and then dividing them by all instances. Since we are evaluating performances for imbalanced data, this metric is not very meaningful. Let us assume that we have an imbalanced ratio of 1:99, which means that the minority class consists only of 1% of all data instances and the majority class consists of the remaining 99% instances. So, if we achieve an accuracy of 99%, that could mean that we have classified all majority instances correctly but all minority instances wrongly [33].

To overcome this shortcoming, various other evaluation metrics exist, which are more suited for the imbalanced domain [20] [33]. These metrics are

$$Precision = \frac{TP}{TP + FP} \tag{3.4}$$

$$True\ Positive\ Rate\ (TPR)\ or\ Recall/Sensitivity = \frac{TP}{TP + FN} \tag{3.5}$$

$$True\ Negative\ Rate\ (TNR)\ or\ Specifity = \frac{TN}{TN + FP} \tag{3.6}$$

$$False\ Positive\ Rate\ (FPR) = \frac{FP}{TN + FP} \tag{3.7}$$

$$False\ Negative\ Rate\ (FNR) = \frac{FN}{TP + FN} \tag{3.8}$$

In general, all metrics are based on TP, FP, FN and TN. First and foremost, Precision measures the exactness which means how many of all predicted positives are classified correctly. On the other hand, TPR or Recall/Sensitivity measures the completeness which means how many instances of all real positives are predicted correctly [20].

Intuitively, TNR computes how many instances of all real negatives are predicted correctly. Now, FNR and FPR have an inverse relationship to TPR and TNR respectively. FNR states how many instances of all real positives are predicted wrongly and FPR calculates how many instances of all real negatives are predicted wrongly. However, further metrics are

$$F - Measure = \frac{(1 + ß^2) * Recall * Precision}{ß^2 * Recall + Precision} \tag{3.9}$$

$$G - mean = \sqrt{sensitivity * specifity} \tag{3.10}$$

The $F - Measure$ combines Recall and Precision as weighted ratio to represent the effectiveness of the classifier. The weight is based on the ß parameter. Usually, this parameter is set to 1 and so a balanced weight of Precision and Recall is achieved. Even though this metric gives more insight than the accuracy metric, it is still sensitive to imbalanced data distributions. The $G - mean$ finally calculates the ratio of positive accuracy and negative accuracy which represents the degree of inductive bias [20].

Receiver operating characteristics curves

The receiver operating characteristics (ROC) curve plots the TPR against the FPR. Each point within this graph represents a single classifier at a specific data distribution. This means, that such a graph yields to a visual representation between benefits (TPR) and costs (FPR) for various data distributions. An example is shown in Figure 31. The Points $\{A, B, C, D, E, F, G\}$ originate from hard-type classifiers which are only able to produce a single $\{TPR, FPR\}$ pair. So, Point A represents a perfect classification (100% TPR, 0% FPR) while Point B represents the worst possible classification (0% TPR, 100% FPR). The separation line between the white and the grey space stretched between Point C and D expresses a random classifier and therefore Point E represents a random guess. So, each Point within the grey lower right triangle performs worse than random guessing (e.g., Point F) and on the other hand each Point within the white upper left triangle performs better than random guessing (e.g., Point G) [27].

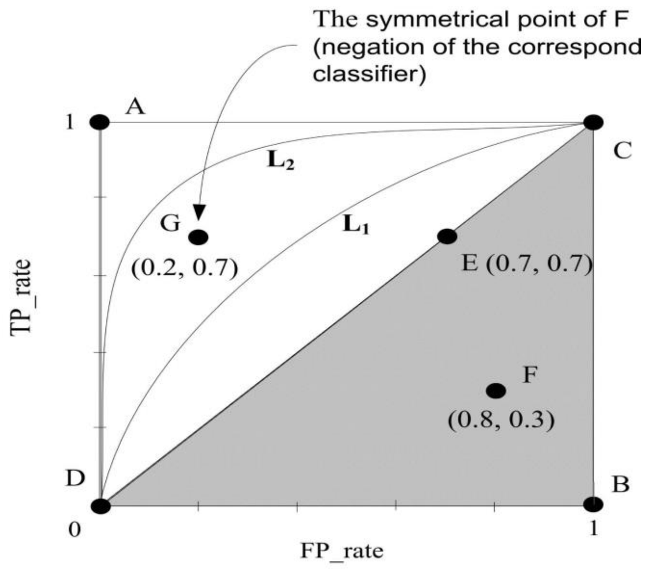


Figure 31: ROC curve example by [27]

On the other hand, a series of ROC points produced by a threshold can generate full-featured ROC curves. So, the curves $\{L_1, L_2\}$ in Figure 31 are provided by soft-type classifiers, which can output continuous numeric values representing the confidence of an instance. To compare the average performance of different classifiers, the area under curve (AUC) is calculated. For example, the area under the curve L_2 is greater than the area under the curve L_1. This means, that the classifier which has generated the curve L_2 has a better average performance than the classifier which has generated the curve L_1 [27].

4 Datasets

For all following tests, two different datasets are used. The first dataset is the Australian Defense Force Academy Linux Dataset (ADFA-LD) [37] which consists of normal behavior traffic and different attacks against an Ubuntu Linux server [38]. This dataset belongs to traditional network data which could occur in an analogous way in the information and communication technology (ICT) part of a smart grid network. The second dataset, belonging to the energy part of smart grids, is the Industrial Control System Power System Dataset (ICS-PSD) [39] [40] [41] [42] which contains different event types. These event types are No Events, Natural Events and Attack Events. Additionally, both ADFA-LD and ICS-PSD include multiple classes to distinguish different attack types respectively events [37] [39]. Nonetheless, this research project deals only with the binary classification problem and will not investigate the multiclass classification problem for these datasets.

In the following subsections, both datasets will be explained. This explanation will contain a detailed description of the datasets, the dataset structure and quantities, the imbalance ratio, a processing description and a literature review on previous performances.

4.1 ADFA-LD

The ADFA-LD was made for anomaly based intrusion detection systems and created due to missing datasets containing contemporary attack protocols. One example of an outdated dataset is the Knowledge Discovery and Data Mining (KDD) dataset [43] which was generated in 1998 and was historically the most used dataset for IDS research. Prior to the generation of the ADFA-LD, there were some other not so successful attempts to generate and establish a contemporary and standard dataset for IDS research (e.g., the UNM dataset [44]). However, to generate the ADFA-LD, an Ubuntu Linux Server Version 11.04 was used as operating system. To allow different attacks, Apache Version 2.2.17 [45] with PHP Version 5.3.5 [46] and MySQL in Version 14.14 [47] were installed and started. The File Transfer Protocol (FTP) [48] and the Secure Shell (SSH) [49] services were enabled, too. To add additional vulnerability, TikiWiki Version 8.1 [50] was installed and started. After the full installation of the software and the installation of all available patches, different payloads to attack the operating system were generated [37] [51].

These payloads include password brute forcing, adding new superusers, a Java Based Meterpreter, a Linux Meterpreter Payload and a C100 Webshell. The vectors used for the password brute force attack were FTP by Hydra and SSH by Hydra. A client-side poisoned executable vector was used for both adding new superusers and to transfer the Linux Meterpreter Payload. To get a Java Based Meterpreter session, a TikiWiki vulnerability exploit was sent to the server. Finally, for the C100 Webshell payload a PHP Remote File Inclusion vulnerability was exploited. Altogether, these payloads and attack vectors represent current practices and tools to exploit a system. Considering the preparation of the server, a realistic defense environment was provided, too. Through several tests with different algorithms and the comparison with the KDD dataset, the ADFA-LD was validated as challenging and representative dataset for current cyber-attacks [37] [51].

Data structure

During normal operations like web browsing or document operations 833 traces of system calls for normal training data and 4373 traces of system calls for normal validation data were collected. The normal training data traces contain only traces with a file size between 300 Bytes and 6 Kilobytes while the normal validation data traces contain traces with a file size between 300 Bytes and 10 Kilobytes. The separation was done as trade-off between data fidelity and processing time. Since the goal of this research project is to gain the best detection rate, all normal data traces will be combined. This results in a total of 5206 normal behavior traces which will be classified as class 0 respectively negatives. For the generation of the attack data, ten attacks were executed for each attack vector, which results in totally 746 attack data traces denoted as class 1 respectively positives. Consequently, the imbalanced ratio is approximately 1:7 [37]. The whole dataset is offered as a download at [52]. The downloaded file contains a separate subfolder for attack data, training data and validation data. Furthermore, the attack data folder contains subfolders for all different payloads and attack vectors. In all these subfolders are different amounts of text files contained. However, each single data example is an individual file and equals a system call trace whereby the term ‘trace’ refers to a sequence of single system calls for a privileged process. Each different system call has a different unique system call identification (ID). So, a sequence of system call IDs is saved for each system call trace and is therefore a single data example [51]. More information about the system call extraction can be found in [53].

Processing

Since system call traces have different lengths, it is not possible to process them directly with a machine learning algorithm. Different solutions to bypass this problem consider trace lengths, the usage of common patterns or to count the frequencies. In [51] the author stated, that the trace length is not an effective way to find anomalies. Whereas common patterns like consecutive system call IDs are effective but highly time-consuming.

Thus, a frequency based counting to gain a common sample length for system call traces is used. Therefore, the adapted dataset will consist of the same number of features as system call IDs. In case of the ADFA-LD, the highest system call ID is 340. Consequently, each data instance has 341 features, starting with the system call ID 0. For each file respectively system call trace, a row with 341 zero value entries is created. Now each occurrence of a system call ID in the trace increases the according (system call ID) feature by 1. After the processing of a trace, the row is added either to a matrix containing normal data or to a matrix containing attack data. So, two $m\ x\ n$ matrices were created with m as the number of either normal or attack data instances and n as the number of features.

Through further investigations, it was found that some system call IDs never occur within all system call traces. Hence, both matrices were combined and the system call IDs respectively feature IDs with column sums equal 0 were extracted. Afterwards the found system call ID columns were removed from both the normal and attack data matrices. This adjustment has no impact to the detection rate but results in a higher performance due to the reduced feature space. Finally, the created matrices are stored to the local machine to save processing time. During the runtime of the program the data is stored in dictionaries.

Previous performance

Regarding to [54] the highest percentage achieved for the ADFA-LD measured by the area under the curve for a ROC curve is 95.32%. This value was achieved by a classification with a semantic Extreme Learning Machine [55]. In comparison, 88.93%, 76.22% and 86.87% were achieved with a semantic SVM, a syntactic Hidden Markov Model [56] and the Sequence Time-Delay Embedding 10 method [54] respectively. More results with different approaches can be found in [57], [37] and [51] in form of ROC plots without attached values.

4.2 ICS-PSD

Disturbances in power systems occur not only by natural events, but additionally by man-made events. Even though power systems were built with redundancies, computer security was not considered as relevant firstly. Through the emergence of smart grid networks, a lot of different intrusion detection techniques were developed. However, all the different approaches are limited to individual devices or attacks or they are very expensive to implement. So, with the ICS-PSD and the used network for the dataset generation it was intended to have a possibility to test various anomaly detection methods for power systems and to detect both natural and malicious events [39].

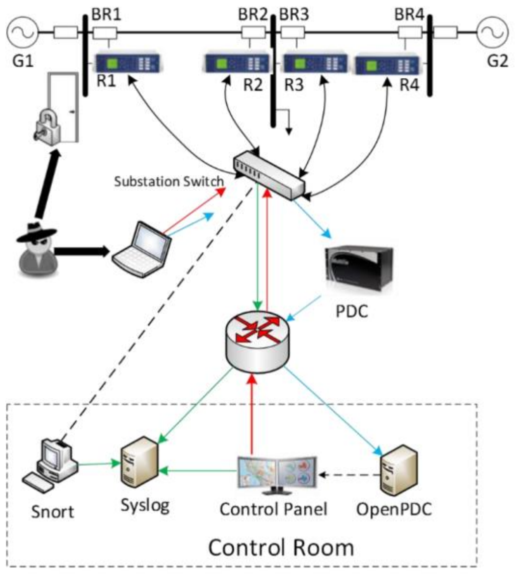


Figure 32: Network diagram for the ICS-PSD [39]

Figure 32 shows the power system framework with different device types used to generate the ICS-PSD. For example, the components G1 and G2 represent power generators. The components BR1 to BR4 are breakers with a single line between BR1 and BR2 and another line between BR3 and BR4. The Intelligent Electronic Devices (IEDs) R1 to R4 can switch these breakers on or off automatically. Additionally, the IEDs have a distance protection scheme to trip a breaker once a fault is detected. Finally, it is possible to trip breakers via the IEDs manually. For all state changes, the IEDs submit information to the control system through a substation switch and a router. Beside the control system, there are different network monitoring systems such as a Snort [58] and Syslog [59] server and also data acquisition systems. However, all following attack scenarios assume that an attacker has already gained access to the substation switch [39].

To generate disturbances, five different scenarios were performed. These scenarios are a short-circuit-fault, a line-maintenance, a remote tripping command injection, a relay setting change and a data injection. While the short-circuit-fault, which is a short in a power line, and the line maintenance belong to the natural events, the other scenarios belong to the attack events. During a remote tripping command injection, the attacker sends a command to an IED to open a breaker. With the relay setting change the attacker disables the distance protection scheme for the IEDs in what way they cannot trip for valid commands or faults. Finally, the data injection simulates a valid fault by changing different values which causes a black-out [39].

Data structure

The dataset is offered at [60] as multi-class dataset with 8 different natural events, 28 different attack events and a single class with no events. Additionally, the dataset is offered as Ternary-class dataset with natural events, attack events and no events and finally as Binary-class dataset where the normal and natural events were grouped together to one class and the attack events represent the other class [39].

As previously stated, this research project covers only the binary classification problem. Nonetheless, the Ternary-class dataset was downloaded and grouped to two different binary datasets. The first dataset equals the Binary-class dataset from [60] and groups the normal and natural events to class 0 respectively negatives while the attack events represent class 1 respectively positives. The second dataset groups the natural and attack events to class 1 respectively positives while the normal events represent class 0 respectively negatives. The first binary dataset is denoted as ICS-PSD-NNvA and the second one as ICS-PSD-NvNA.

Now the downloaded archive contains 15 different comma-separated values (CSV) files with thousands of individual samples for each distinct event. In total, there are 4405 normal events, 18309 natural events and 55663 attack events. To work with a similar data amount as the ADFA-LD for better comparisons and a decent processing time, just the first CSV file is used. This file contains 173 normal events, 927 natural events and 3866 attack events. In case of the ICS-PSD-NNvA, class 0 consists of 1100 samples and class 1 of 3866 samples consequently. So, the imbalanced ratio for this dataset is approximately 1:3.50. In contrast, class 0 of the ICS-PSD-NvNA consists of 173 samples while class 1 consists of 4793 samples. This results in an imbalanced ratio of approximately 1:27.70.

Each of the datasets has 129 columns and 128 features with 116 different measurements from four phasor measurement units (PMUs). The remaining 12 features consist of control panel logs, Snort alerts and relay logs for the PMUs. The last entry is a marker for the event type to classify if it is a normal event, a natural event or an attack event.

Processing

For the simple reason that the dataset is a single CSV file and each data entry has the same number of features, it is sufficient to iterate over each data entry in this file. Just the last column entry must be separated to distinguish between the event type. However, there are a few entries with an infinite value. Since the machine learning algorithms are not able to deal with any data types but numbers, it is necessary to substitute these values. There is no difference whichever number is substituted. The only two important things to consider are that firstly the substituted number remains the same for each substitution and secondly the number differs from any other containing number. Since all values are greater than or equal to zero, the easiest way is to choose a negative number. In this case minus one was chosen. Now, three different matrices, one for each event type, are created and each data entry is stored in the related matrix. Finally, the created matrices are stored in the local machine exactly like the ADFA-LD and will be handled as dictionaries during the runtime, too. Furthermore, the stored matrices can be loaded either as ICS-PSD-NNvA or as ICS-PSD-NvNA during runtime and will be accordingly grouped during the import process.

Previous performance

Despite a literate review, no prior classification results for the generated ICS-PSD-NNvA were found. However, for the ICS-PSD-NvNA there are different performances in [39] available. The highest F-Measure percentage for the ICS-PSD-NvNA is approximately 92% (estimated from a plot) and was achieved with the AdaBoost+JRipper classifier. However, there was a higher percentage of 95.5% achieved for the Ternary-class dataset. Apart from [39], no other performance evaluations were found.

The next chapter briefly describes the development environment (e.g., programing language, development tools, project structure).

5 Development environment

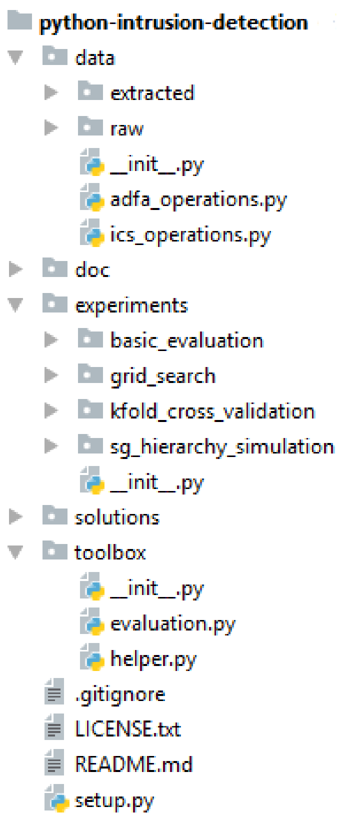
The data processing as well as any other pattern recognition procedures such as preprocessing, classification/prediction and evaluation are implemented with the Python programing language [61] in version 3.6.0 which is for now one of the latest versions. Since there are many different packages used, the Anaconda Distribution [62] is installed. The Anaconda Distribution is a package manager, an environment manager and a Python distribution, including over 720 open source packages. The Jetbrains PyCharm [63] integrated development environment (IDE) is used for the source code development.

The necessary packages to run the full program are stated in the following list:

- NumPy [64] >= 1.11.2
- SciPy [65] >= 0.18.1
- Scikit-Learn [66] >= 0.18.1
- Matplotlib [67] >= 2.0.0
- Imbalanced-Learn [68] >= 0.2.1
- CostsensitiveClassification [36] >= 0.5.0

NumPy is a useful and efficient structure for numerical works with arrays and to manipulate matrices. So, NumPy arrays are used to store all the data and targets from the different datasets and for various calculations as well. SciPy contains modules for scientific work like linear algebra, integration, interpolation and image processing. Both NumPy and SciPy are required to install Scikit-Learn, which is used as the main library for all pattern recognition tasks. Scikit-Learn is a very powerful machine learning library featuring for example classification, regression, clustering, dimensionality reduction, model selection and preprocessing. With the Matplotlib module it is possible to generate diverse types of plots. For this research project, Matplotlib is used to generate ROC curves. Finally, Imbalanced-Learn and CostsensitiveClassification are both libraries to execute different tests with the imbalanced datasets. The Imbalanced-Learn package supports various models for under-sampling, over-sampling and hybrid-sampling. The CostsensitiveClassification library offers models for cost-sensitive classification problems with methods featuring for example cost-sensitive weighting, thresholding and classifiers with built-in cost-sensitive algorithms [64] [65] [66] [67] [68] [36].

The program itself contains three different main modules which are denoted as “data”, “experiments” and “toolbox” and additionally two folders which are denoted as “doc” and “solutions”. The whole structure is illustrated in Figure 33. The “data” module contains one file for all operations for the ADFA-LD and one file for both datasets of the ICS-PSD. Furthermore, this module contains a subfolder for the downloaded raw data and a subfolder for the converted data sets which are stored there after the import process. The “experiments” module contains four submodules, one module for each experiment type. The different experiments are basic evaluations, a grid search, k-fold cross validations and a smart grid hierarchy simulation. For each of these experiments and for each dataset type an own file is created to set various parameters and to start the tests. All these experiments will be explained explicitly in the next chapters. The “toolbox” module contains two python scripts. The “helper” script contains various functions for easier processing such as calculating and saving different metrics, generating and saving ROC plots and also a setup of different tests. The “evaluation” script contains different core functionalities for the experiments.



```
python-intrusion-detection
▼ data
    ▶ extracted
    ▶ raw
      __init__.py
      adfa_operations.py
      ics_operations.py
▶ doc
▼ experiments
    ▶ basic_evaluation
    ▶ grid_search
    ▶ kfold_cross_validation
    ▶ sg_hierarchy_simulation
      __init__.py
▶ solutions
▼ toolbox
      __init__.py
      evaluation.py
      helper.py
  .gitignore
  LICENSE.txt
  README.md
  setup.py
```

Figure 33: Python project structure

Finally, the “doc” directory contains a project documentation built with Sphinx [69] and the “solution” folder contains all achieved rates during the experiments.

6 Method screening

The main goal of this research project is to implement anomaly detection methods for imbalanced data sets in a smart grid hierarchy and to evaluate their performance. So, the first task is to investigate the mentioned imbalanced data methods for the previously described datasets. This task will be described within this chapter from beginning to end. Additionally, this chapter is divided into three parts. The first section will describe the process of testing different classifiers and preprocessing (scaling) methods with a simple stratified sampling to get a first impression of the performance of the three datasets ADFA-LD, ICS-PSD-NvNA and ICS-PSD-NNvA. After this process, the best scaling method will be chosen for all further tests. Then, in the second section, ensemble classifiers will be added to the classifier set and a grid search with k-fold cross validation in the hyper-parameter space for each individual classifier will be executed to find the best individual performance. Finally, in the last section, the methods sampling and cost-sensitive learning for imbalanced data will be tested and evaluated with k-fold cross validation. In this last subchapter, a method will be chosen, which will then be used for the smart grid hierarchy test implementation.

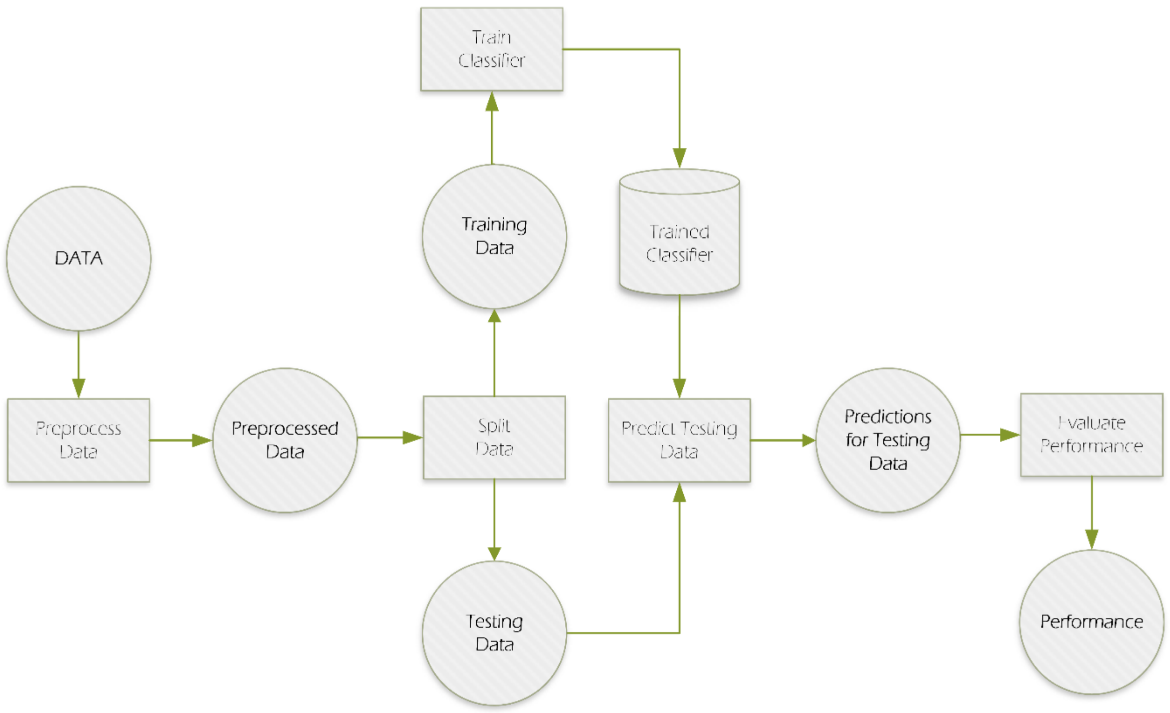


Figure 34: Basic method screening process

If we consider the steps in the pattern recognition cycle, the first steps are feature extraction and preprocessing. Since the feature extraction process was described in the chapter about the datasets, all steps performed within these three sections will start with preprocessing and will end with performance evaluations. These steps are illustrated in Figure 34. So, after the dataset is loaded and the data is preprocessed, the next step is to split the data in training and test data. Once the classifier is trained with the training data, the test data is predicted with the trained classifier. Finally, the predictions for the testing data will be evaluated with the introduced performance metrics. The steps from splitting to prediction will be repeated n times for a more robust performance. Anyway, the described cycle must be executed for each single preprocessing, splitting, sampling and also classifier.

However, to execute the tests for each subcategory, two files are used. The first file is located in the experiments module and the related experiment. This file is named after the dataset and contains the basic setup for the tests, such as the number of iterations, the fold size, scaling methods, the used classifiers and so on. This file enables an easy adoption of the test setup. The second file, namely evaluations, is located in the toolbox module and provides all the necessary tasks for the related subchapter purpose.

6.1 Classifier and sampling tests

First of all, before any test can be executed, it is necessary to define a set of classifiers, which will be used for all tests and comparisons. The following single classifiers were chosen:

- Decision trees (DT)
- k-nearest neighbor (k-NN)
- Quadratic Discriminant Analysis (QDA)
- Multilayer Perceptrons (MLP)
- Support vector machines with a linear kernel (linSVM) and a RBF kernel (rbfSVM)

For the first tests, all the classifiers will be used with their default (hyper-)parameters. As previously stated, the first step once the data is loaded is to preprocess the data. To find the best possible method, all different scaling methods, namely interval fit in the range of $[0,1]$, z-score scaling (standardization) and arctangent scaling were tested and compared to the performance of the unscaled data for each dataset. The interval fit method and the z-score scaling methods are directly implemented in the Scikit-Learn module “preprocessing”.

The arctangent scaling is not directly implemented and must be programmed. To get the arctangent scaled data (denoted as a) the standardized data (denoted as s) is scaled by

$$a = \frac{2}{\pi} * \tan^{-1}(s) \tag{6.1}$$

Then, for each dataset the data is split into training and test data and targets with stratified sampling with a 70:30 training/test split. To execute this task, the built-in function "train_test_split" from the module "model_selection" in Scikit-Learn is used. Once the data is split, the training data and targets are used to train the classifiers and then the trained classifier is used to predict the test data split. The related functions are directly built-in in the classifiers and are denoted as "fit" and "predict". The parameters for the "fit" function are the training data and the training targets and the only parameter for the "predict" function is the test data set. Finally, the performance is evaluated by comparing the original respectively correct test targets with the predicted targets. For a more robust result, the process is repeated 10 times and then the performance is simply averaged.

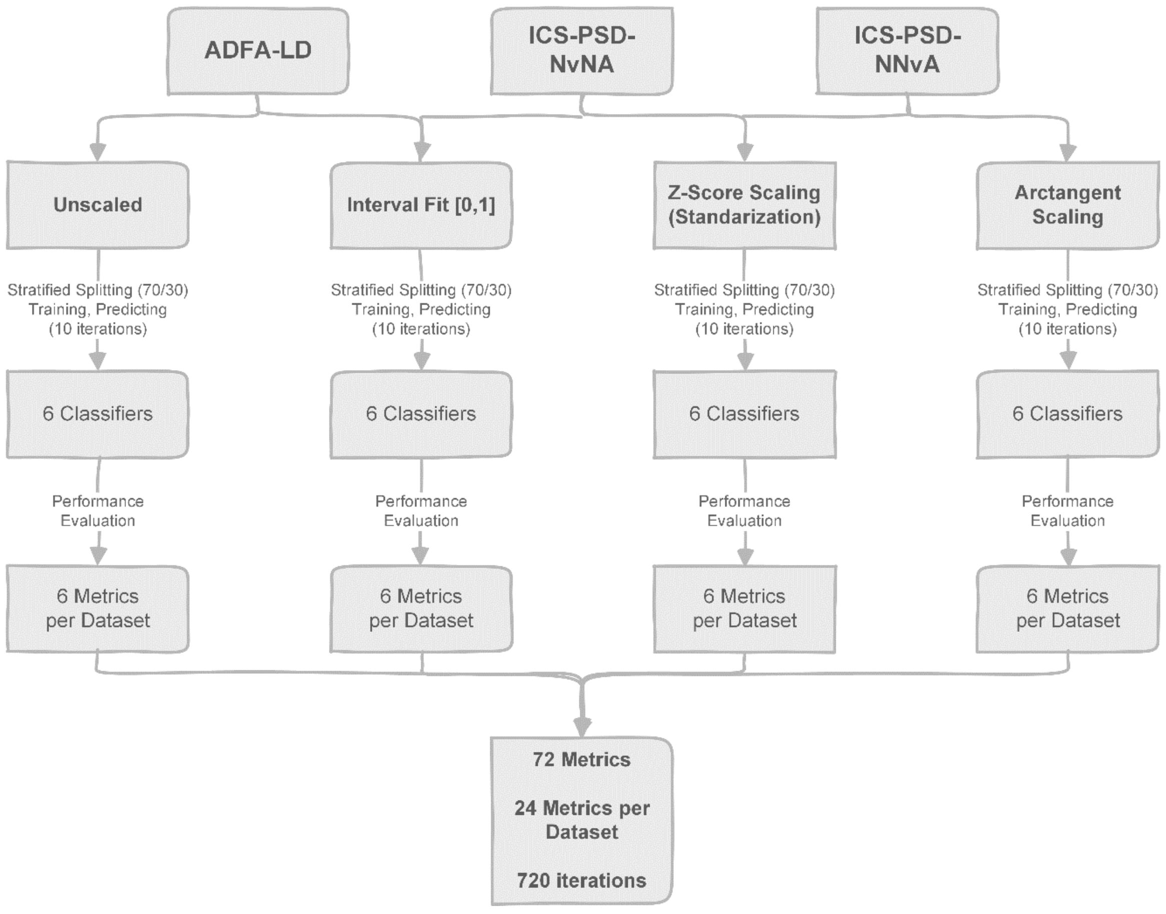


Figure 35: Test-setup for scaling and classifier tests

This test setup is illustrated in Figure 35. So, each dataset is scaled separately to an interval [0,1], is standardized and arctangent scaled. Including the unscaled dataset, this means that for each dataset four different scaling methods exist. Consequently, 12 different datasets will be tested. Since we use 6 different classifiers, this leads to 72 iterations in total to execute the whole test setup a single time. Since the test is repeated 10 times for a more robust solution, the basic test setup needs 240 iterations for each dataset and 720 total iterations to complete all tests. For further clarification, the algorithm "basic_evaluation_binary" is stated in Listing 5 in Appendix A. If all parameters are passed correctly, a dictionary for the results is created. For each classifier, a dictionary is created and within such a classifier dictionary another dictionary for all the different scaling types is created. Then, a single scaling rate is initialized as a 2x2 matrix with zero values to store the confusion matrix. Next, a for-loop executes the test n times. Within this for-loop, the first step is to split the data in training and test data and targets with stratified random sampling. Then, the data is scaled for each passed scaling type. Once we have the split and scaled datasets, the training and prediction tasks are executed for each passed classifier and scaling type. Then, the confusion matrix from the predicted targets is calculated with a built-in function "confusion_matrix" in the module "metrics" in Scikit-Learn and is added to the related rates in the previously created dictionary. Once all n iterations are finished, the rates are divided by the number of iterations to get the average performance for each classifier. Finally, the rates dictionary is returned.

Now, this test-setup is evaluated with TPR, TNR and ACC. For each dataset, an own table will show the performances for the stated classifiers DT, k-NN, MLP, QDA, linSVM and rbfSVM and each scaling type. The performance for the ADFA-LD is shown in Table 4. In general, a satisfactory performance is achieved with the out-of-the-box classifiers for the ADFA-LD. Especially the decision tree, k-NN and MLP classifiers have a high accuracy with approximately 95%. Since the positives in the ADFA-LD are the minority class and the negatives are the majority class, the performances for TPR and TNR behave as expected. The TNR (detection rate for negatives respectively for the majority class) for these three classifiers is about 97% and the TPR (detection rate for positives respectively minority class) is between approximately 79% and 84%. In case of decision trees and k-NN, the scaling of the data has little to no impact on the performance and lies within the fluctuation margin through random sampling. On the other hand, the MLP classifier shows little improvements for the interval fit and good improvements for standardization and arctangent scaling.

	unscaled			interval-fit [0,1]			z-score scaling			arctangent scaling		
	TPR	**TNR**	**ACC**	**TPR**	**TNR**	**ACC**	**TPR**	**TNR**	**ACC**	**TPR**	**TNR**	**ACC**
DT	0.8103	0.9697	0.9497	0.8147	0.9705	0.9510	0.8121	0.9702	0.9504	0.8063	0.9707	0.9501
k-NN	0.8388	0.9736	0.9567	0.8317	0.9704	0.9530	0.8433	0.9718	0.9557	0.8491	0.9778	0.9616
MLP	0.7911	0.9793	0.9557	0.8272	0.9800	0.9609	0.8545	0.9802	0.9644	0.8536	0.9832	0.9670
QDA	0.9710	0.4738	0.5371	0.9688	0.5061	0.5641	0.9656	0.4967	0.5555	0.9741	0.4652	0.5291
linSVM	0.7344	0.9360	0.9107	0.3754	0.9834	0.9072	0.8174	0.9618	0.9437	0.8183	0.9720	0.9527
rbfSVM	0.3179	0.9971	0.9119	0.0000	1.0000	0.8746	0.5330	0.9855	0.9287	0.0009	0.9999	0.8746

Table 4: Basic evaluation results - ADFA-LD

The QDA classifier steps out of the line and shows a bad accuracy of 53.71%. However, the performance for the minority class is with 97.10% very high and with 47.38% for the majority class very low. Since there are a lot more majority instances, the accuracy is decreased. This example shows perfectly the low validity of the accuracy score. A high detection rate for the majority class enables a good accuracy score while a good performance for the minority class does not lead to a high accuracy score necessarily. Nevertheless, in combination with TPR and TNR the validity is still good enough to choose the best scaling method after comparison of all performances. However, interval fit and z-score scaling show a slight improvement for the minority class detection rate for the QDA classifier while the arctangent scaling has no impact.

The SVM with linear kernel achieves an accuracy of 91.07% with a TNR of 93.60% and a TPR of 73.44%. So, the performance is significantly worse than in comparison to the DT, the k-NN and the MLP classifiers. But the z-score scaling and arctangent scaling improve the performance in such a way that the gap is nearly closed. On the other hand, by using the interval fit method, the performance for the TPR drops significantly. Finally, the SVM with RBF kernel has the best detection rate for the majority class with a TNR of 99.71% but unfortunately a very low detection rate for the minority class with a TPR of 31.79%. This leads to a total accuracy of 91.19%. This behavior is even more intense with the use of the interval fit method and arctangent scaling. With interval fit all majority examples are detected and no single minority instance is correctly predicted and with arctangent scaling nearly all majority instances and only a few minority instances are detected. With z-score scaling the performance is more balanced by achieving a good detection rate for the majority class and nearly a doubled detection rate of the minority class in comparison to the unscaled data. In general, the low performances for the SVMs are unexpected since they usually show a good generalization ability out-of-the-box. But this circumstance will be investigated in the next subchapter.

However, with the ADFA-LD alone, no statement for the best scaling method is possible but there are little advantages for z-score scaling and arctangent scaling, since they show better improvements for single classifiers. Especially the classifiers with an already good performance showed even a slight performance improvement using standardization or arctangent scaling.

	unscaled			**interval-fit [0,1]**			**z-score scaling**			**arctangent scaling**		
	TPR	**TNR**	**ACC**	**TPR**	**TNR**	**ACC**	**TPR**	**TNR**	**ACC**	**TPR**	**TNR**	**ACC**
DT	0.9945	0.8519	0.9895	0.9946	0.8365	0.9891	0.9950	0.8404	0.9896	0.9942	0.8423	0.9889
k-NN	0.9926	0.6750	0.9815	0.9954	0.8327	0.9897	0.9958	0.8385	0.9903	0.9967	0.8481	0.9915
MLP	0.9316	0.1135	0.9030	0.9946	0.1712	0.9658	0.9958	0.8385	0.9903	0.9967	0.8481	0.9915
QDA	0.9955	0.8250	0.9895	0.9955	0.8259	0.9895	0.9955	0.8250	0.9895	0.9967	0.8135	0.9903
linSVM	0.9993	0.0000	0.9644	0.9991	0.0019	0.9643	0.9892	0.2692	0.9640	0.9944	0.5423	0.9786
rbfSVM	0.9998	0.0019	0.9650	1.0000	0.0000	0.9651	1.0000	0.0000	0.9651	1.0000	0.0000	0.9651

Table 5: Basic evaluation results – ICS-PSD-NvNA

Next, the performance for the ICS-PSD-NvNA is shown in Table 5. First of all, for this dataset the positives are the majority class and the negatives are the minority class. In general, the performance for this dataset is very high. The highest accuracy of 99.15% was achieved with arctangent scaling and the k-NN and MLP classifiers with a TPR of 99.67% and a TNR of 84.81%. Since these classifiers are out-of-the-box single classifiers without hyper-parameter changes, the achieved performance is very good. Another remarkable note in terms of scaling is, that the MLP classifier achieved without scaling only an accuracy of 90.30% with a TPR of 93.16% and a TNR of 11.35%. Like the ADFA-LD, the decision tree achieved a very good performance for the ICS-PSD-NvNA. With an accuracy of 98.95% and a TPR of 99.45% and a TNR of 85.19% the DT is even the best classifier for the unscaled data. Surprisingly, the QDA classifier reached a similar performance than the decision tree although this classifier performed not very good for the ADFA-LD. The performance through scaling was not improved for neither the decision tree nor the QDA classifier. While the k-NN classifier achieved a medium performance in comparison to the best classifiers for unscaled data, the performance for the scaled datasets is much better. As previously stated, the k-NN rule achieved even the best performance with arctangent scaling. Both SVM classifiers show a similar behavior as the ADFA-LD. While they achieve nearly 100% or even 100% detection rate for the majority class, the detection rate for the minority class is smaller than 1% or even 0%. Only with z-score and arctangent scaling the detection rate for the minority class was raised while the performance for the majority class is similar. But again, the performance for both SVM classifiers seems too low.

Overall, both z-score scaling and arctangent scaling are again superior to interval fit scaling and unscaled data. However, the arctangent scaling method is even slightly better than the z-score scaling method.

	unscaled			**interval-fit [0,1]**			**z-score scaling**			**arctangent scaling**		
	TPR	**TNR**	**ACC**	**TPR**	**TNR**	**ACC**	**TPR**	**TNR**	**ACC**	**TPR**	**TNR**	**ACC**
DT	0.9453	0.8270	0.9191	0.9450	0.8209	0.9175	0.9441	0.8312	0.9191	0.9454	0.8252	0.9188
k-NN	0.9370	0.6752	0.8790	0.9407	0.7461	0.8976	0.9459	0.7355	0.8993	0.9423	0.7276	0.8948
MLP	0.9802	0.0255	0.7687	0.9731	0.0961	0.7789	0.9437	0.5424	0.8548	0.9363	0.5358	0.8476
QDA	0.3816	0.9452	0.5064	0.3838	0.9436	0.5078	0.3853	0.9224	0.5087	0.3950	0.9442	0.5166
linSVM	0.8037	0.2009	0.6702	0.9823	0.0694	0.7801	0.9357	0.1855	0.7695	0.9604	0.1564	0.7823
rbfSVM	0.9997	0.0206	0.7829	0.9450	0.8209	0.9175	0.9949	0.0506	0.7858	1.0000	0.0000	0.7785

Table 6: Basic evaluation results – ICS-PSD-NNvA

The performance for the ICS-PSD-NNvA is shown in Table 6. The ICS-PSD-NNvA has again more positive instances than negative instances. The performance for the best classifiers DT and k-NN is very good, but worse than the ICS-PSD-NvNA. This seems reasonable, since in the ICS-PSD-NNvA the normal and natural event data are combined to the negative class. For the unscaled data, the decision tree achieves an accuracy of 91.91% with a TPR of 94.53% and a TNR of 82.70% while the k-NN rule achieves an accuracy of 87.90% with a TPR of 93.70% and a TNR of 67.52%. Even for these two classifiers, the detection rate for the minority class has large varieties. However, the other classifiers have a very bad performance. Even if the MLP and rbfSVM achieve an accuracy between 76% and 79% with a TPR between 98% and 100%, the TNR for both is only between 2% and 3%. The linSVM achieves also only approximately 2% TNR and 80% TPR which results in an accuracy of 67.02%. The QDA demonstrates the same behavior than in the ADFA-LD. The detection rate for the minority class is with 94.52% very high while the detection rate for the majority class with only 38.16% is very low. This results in a total accuracy of 50.64%. The scaling behavior is similar to the other datasets. The interval fit method shows slight improvements especially for the k-NN, MLP and both SVM classifiers. The same behavior occurs for z-score scaling and arctangent scaling but again the improvements are better than the interval fit method. Still, the performance for the SVM is surprisingly low.

In general, the decision tree classifier performs very well for all three datasets but shows no improvements with the scaling methods. While the k-NN rule has also a good performance for all datasets, the performance improvements with scaling are high (especially through z-score scaling and even more through arctangent scaling). So, the k-NN rule performed overall for two out of the three datasets better than the decision tree classifier.

The same behavior occurs for the MLP classifier. Consequently, the three classifiers DT, k-NN an MLP showed the best performances. The k-NN rule and the MLP classifier achieved also a remarkable improvement through scaling, especially with arctangent scaling. On the other hand, QDA shows an odd behavior by having very high detection rates for the minority class for the datasets ADFA-LD and the ICS-PSD-NNvA and low detection rates for the majority class. Another strange behavior for the QDA classifier is that this classifier ranks simultaneously to the two best classifiers for the ICS-PSD-NvNA. Even though the scaling improved the performance for both SVM classifiers, the performance is still bad and unexpected low for all three datasets. Anyway, the performance improvements by scaling are very good. Since the best improvements were achieved by the arctangent scaling method, further test will use this method only.

This section investigated only the basic performance for the datasets and the different scaling methods. So, in the next subchapter the full range of classifiers, including ensemble learners, will be used. Additionally, to improve the performance for all classifiers, a full k-fold cross validation grid search will be executed. After this, a full test for the three datasets with k-fold cross validation and the ensemble learners will be performed. The results of this test will be the foundation for the comparison with all imbalanced data methods.

6.2 Ensembles and hyper-parameter grid-search

Now, some ensemble learners will be added to the classifier kit. Since the decision tree classifier has a stable high performance, this classifier will be used as base learner for all non-voting ensemble classifiers. Voting classifiers will be used as well but since they do not have any parameters to tune they will be used the first time in the k-fold cross validation test setup. So, the used classifiers for the grid search are

- Decision tree (DT)
- k-nearest neighbor (k-NN)
- Quadratic Discriminant Analysis (QDA)
- Multilayer Perceptrons (MLP)
- Support vector machine (SVM)
- AdaBoost with decision trees as base learners (DTBoost)
- Bagging with decision Trees as base learners (DTBagg)
- Random forest (RForest)

All of these classifiers have different hyper-parameters which will be explained subsequently. Prior to that, the basic process of a grid search CV is illustrated in Figure 36.

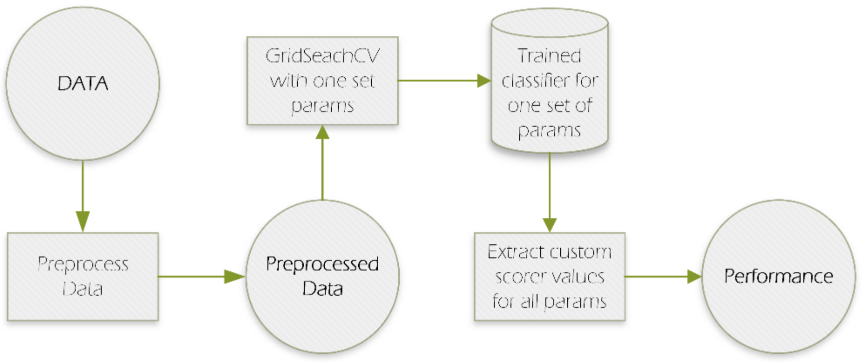

DATA
GridSeachCV with one set params
Trained classifier for one set of params
Preprocess Data
Preprocessed Data
Extract custom scorer values for all params
Performance

Figure 36: Basic grid search cross validation process

So, to execute a single iteration of a grid search cross validation, a dataset is loaded and is then preprocessed with arctangent scaling as elaborated in the previous subchapter. Then, a classifier and a set of related defined parameters is used for the cross validation with this parameter set. Luckily, Scikit-Learn offers in the “model_selection” module a function for a grid search cross validation, namely “GridSearchCV”. To use this function, one must pass the used classifiers, the parameters, the number of folds and a scoring function.

The scoring function offers different default metrics such as the accuracy score to compare the achieved performances for different parameter sets. Since no appropriate metric for imbalanced data is available, a custom scorer function was created and is stated in Listing 6 in Appendix A. This custom scorer function calculates both the F1-Measure and G-mean and averages the performance.

Back to the execution of a single grid search iteration, the next step is to extract the performances. To evaluate the performance, it is only necessary to train respectively fit the grid search classifier with each set of the chosen parameter sets. The built-in function “GridSearchCV” offers an attribute which is called “best_params_” which returns the mean performance and standard deviation for the k folds for a specific parameter set. This process is then repeated for each parameter set and for each classifier. For example, let us consider the k-NN rule which has a hyper-parameter called “n_neighbors”. This parameter simply represents the number of neighbors. So, if we want to evaluate the performance of this classifier with neighbors from 1 to 9, this single process is repeated 9 times.

The first iteration performs the cross validation with 1-NN, the next iteration with 2-NN, the next with 3-NN and so on. If we would add another parameter with two different possible values, the cross validation would be executed 18 times since each parameter is combined with each other. Once all parameter sets are evaluated, the best set can be chosen. Since k-fold cross validation is used for the performance evaluation, overfitting is avoided.

The source code for the generation of the classifiers with all related parameters for the grid search cross validation is shown Listing 7 in Appendix A. In general, a parameter set is defined as dictionary with the name of the hyper-parameter as key and a list of parameter values as value. Additionally, a dictionary for each classifier is created. While the key is the name of the classifier, the first value contains the classifier itself and the second value contains the parameter dictionary set.

First of all, the parameters chosen for the DT classifier are “criterion”, “min_samples_split”, “max_depth”, “min_samples_leaf” and “max_leaf_nodes”. The “criterion” parameter measures the quality of a split and offers the criteria “gini” and “entropy”. The default parameter is “gini”. So, both criteria will be used for the grid search. Next, “min_samples_split” defines the number of samples required to split an internal node. The default parameter is 2 and the values 5 and 10 are added to the grid search. The parameter “max_depth” defines the maximum depth of the tree. The default parameter is “None” which means that nodes are expanded until all leaves contain less than “min_samples_split” samples or until all leaves are pure. The parameter “None” is extended by the values 5, 10 and 20 for the grid search. The parameter “min_samples_leaf” defines the minimum number of samples required to be at a leaf node. The default value is 1 and is extended by 2 and 5 for the grid search. Finally, the “max_leaf_nodes” parameter grows the leaves of a tree limited by these values. The default value is “None” which means that there is no stopping criterion. For the grid search, the “None” value is extended by 20 and 50 [70].

Now, all parameters and values are defined. The parameter “criterion” has 2 values, the parameter “min_samples_split” has 3 values, the parameter “max_depth” has 4 values, the parameter “min_samples_leaf” has 3 values and the parameter “max_leaf_nodes” has also 3 values. This leads to $2 * 3 * 4 * 3 * 3 = 216$ different combinations for the decision tree classifier.

For the k-NN classifier, only the two parameters “n_neighbors” and “weights” are used. While “n_neighbors” defines the number of neighbors to use (default=5), the parameter “weights” is used to define a weight function for the prediction. Therefore, it is possible to choose between “uniform”, which weighs all points in the neighborhood equally, “distance”, which weighs the points by the inverse of their distance or a custom weight function. The default value is uniform [71]. For the grid search, this value will be extended by the “distance” weight function. Additionally, a range of neighbors between 1 and 9 will be used for the “n_neighbors” parameter. This leads to $2 * 9 = 18$ different combinations for the k-NN classifier.

This process is repeated for all other classifiers with their related parameters. The descriptions for the parameter definitions for the QDA classifier can be found in [72], for the MLP classifier in [73], for the SVM classifier in [74], for the AdaBoost classifier in [75], for the bagging classifier in [76] and for the random forest classifier in [77].

However, this leads to 11 different combinations for the QDA classifier, to $10 * 3 * 3 = 90$ different combination for the MLP classifier, to $1 * 4 * 13 + 1 * 4 = 56$ different combination for the SVM classifier and to $10 * 12 * 2 = 240$ different combination for the random forest classifier.

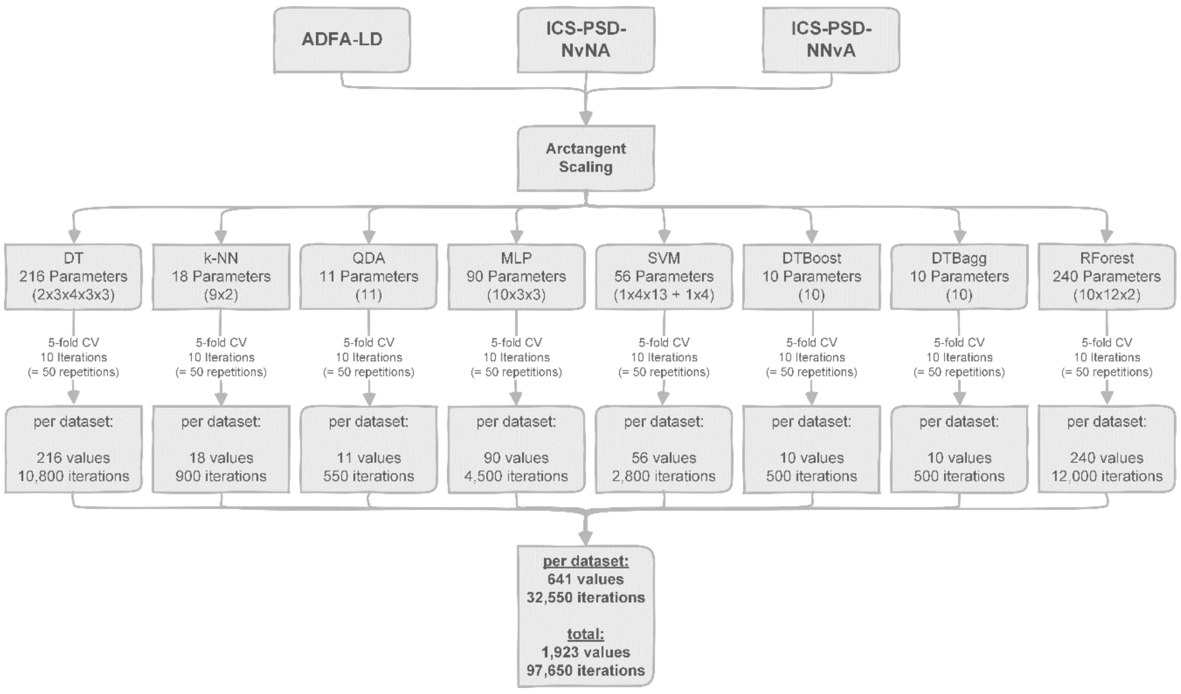

ADFA-LD
ICS-PSD-NvNA
ICS-PSD-NNvA
Arctangent Scaling
DT 216 Parameters (2x3x4x3x3)
k-NN 18 Parameters (9x2)
QDA 11 Parameters (11)
MLP 90 Parameters (10x3x3)
SVM 56 Parameters (1x4x13 + 1x4)
DTBoost 10 Parameters (10)
DTBagg 10 Parameters (10)
RForest 240 Parameters (10x12x2)
5-fold CV 10 Iterations (= 50 repetitions)
per dataset: 216 values 10,800 iterations
per dataset: 18 values 900 iterations
per dataset: 11 values 550 iterations
per dataset: 90 values 4,500 iterations
per dataset: 56 values 2,800 iterations
per dataset: 10 values 500 iterations
per dataset: 10 values 500 iterations
per dataset: 240 values 12,000 iterations
per dataset: 641 values 32,550 iterations
total: 1,923 values 97,650 iterations

Figure 37: Test-setup for the grid search cross validation

Since both the AdaBoost and the bagging classifier use the decision tree classifier as base learner, a grid search for the decision tree itself would result in redundant workload. So, the decision tree grid search was executed beforehand. The best performance for the decision tree for all datasets was achieved with a changed “criterion” parameter. The changed value is “entropy” while the rest of the parameters remain with their default values. Thus, 218 additional feature combinations can be avoided for both the AdaBoost and bagging classifiers. So, it is only necessary to investigate the number of used base learners (“n_estimators”). Instead of 2,180 different combinations, only 10 different combinations for both DTBoost and DTBagg remain.

The combination of all parameters and the used classifiers are shown in Figure 37. Since a 5-fold cross validation with 10 repetitions is executed for each parameter set, 50 iterations are necessary for each parameter set and for each classifier. This results for each dataset in 641 different parameter sets and 32,550 iterations and for all three datasets together in 97,650 iterations to execute the full grid search cross validation.

The function to execute these 97,650 iterations is stated in Listing 1.

```
def grid_search(data, targets, iterations, number_of_folds,
            number_of_threads, use_DT, use_kNN, use_QDA, use_MLP, use_SVM,
            use_DTBoost, use_DTBagg, use_RForest, save_dir=''):

    # generate grid search classifiers and params for the search
    grid_search_classifiers = __generate_grid_search_classifiers(use_DT,
                                  use_kNN, use_QDA, use_MLP, use_SVM,
                                  use_DTBoost, use_DTBagg, use_RForest)

   # arctangent scaling
    data = preprocessing.StandardScaler().fit_transform(data)
    data = (2 * np.arctan(data)) / np.pi

    for key, value in grid_search_classifiers.items():

        best_params = []
        results = []

        for x in range(0, iterations):

            # shuffle data
            data, targets = shuffle(data, targets)

            # set-up for grid search with stratified k-fold cross validation
                    and f1 + gmean comparison
            clf = GridSearchCV(estimator=value['classifier'],
                    param_grid=value['parameter'], cv=number_of_folds,
                    n_jobs=number_of_threads, scoring=make_scorer(
                             score_func=__custom_scorer_mixed_f1_gmean,
                             greater_is_better=True))

            # calculate the different parameter performances
            clf.fit(data, targets)
            # append results to the list
            best_params.append(json.dumps(str(clf.best_params_)))
            results.append(clf.cv_results_)

        # calculate averages
        all_means = []; all_stds=[];

        for result in results:
            all_means.append(result['mean_test_score'])
            all_stds.append(result['std_test_score'])

        average_means = [np.mean(x) for x in zip(*all_means)]
        average_stds = [np.mean(x) for x in zip(*all_stds)]
        params = clf.cv_results_['params']

        sorted_index = np.argsort(average_means)[::-1]
        average_means = np.array(average_means)[sorted_index]
        average_stds = np.array(average_stds)[sorted_index]
        params = np.array(params)[sorted_index]

        # print solution or save solution to file
        …
```

Listing 1: Grid search execution function

To execute this function successfully, the following parameters are necessary:

- **data [numpy matrix]:** Numpy dataset matrix (columns=features)
- **targets [numpy matrix]:** Numpy target matrix (each row = one target)
- **iterations [int value ≥ 1]:** The number of repetitions for the CV grid search
- **number_of_folds [int value > 1]:** The number of folds for a single grid search
- **number_of_threads [int value ≥ 1]:** The number of threads to use
- **use_*classifier* (any) [boolean]:** True if the individual classifier should be generated
- **save_dir [string]:** If empty, performances are printed. Otherwise the results are stored in the given path. Default = empty string

First of all, the previously described function from Listing 7 is executed to generate the dictionary with all classifiers and parameters for which the grid search cross validation should be executed. Afterwards, the data is preprocessed with arctangent scaling. Then the function iterates over a for-loop for each single classifier. For each classifier, a list with the best parameters and the results is created to store all results which will be generated in another for-loop with 10 iterations. So, for each iteration in each classifier, the grid search cross validation is executed. Before the execution, the data is shuffled with the built-in function “shuffle” from the module “utils” in Scikit-Learn to generate more randomness and to avoid overfitting. Then, the parameters with their achieved performances are appended to the previously created list.

After the 10 iterations, the performances and standard deviations are summed up and averaged. Finally, the performances are either printed or saved. This process is then repeated for each classifier and then for each dataset. Additional to the performance for each parameter set, the best parameter set in each iteration is chosen and the output shows how often a specific parameter set was chosen.

An example output for the k-NN classifier and the ADFA-LD is shown in Listing 2. This example shows, that the parameter “n_neighbors” with the value 4 and the distance weight function was chosen 6 of 10 times as the best classifier with an averaged performance over 10 iterations for F1-Measure and G-Mean of 89.20% and a standard deviation of 3%. Both the k-NN classifier with either 3 or 6 neighbors and also the distance weight function were chosen 2 of 10 times as best classifiers. This implies, that in general the distance weight function results in better performance than the uniform weight function. This behavior is confirmed if we look in the single results for each parameter set.

```
Best parameters set for KNeighbors found on development set:
"{'n_neighbors': 4, 'weights': 'distance'}": 6/10
"{'n_neighbors': 3, 'weights': 'distance'}": 2/10
"{'n_neighbors': 6, 'weights': 'distance'}": 2/10

Average grid scores on development set:
0.892 (+/-0.030) for {'n_neighbors': 4, 'weights': 'distance'}
0.890 (+/-0.029) for {'n_neighbors': 3, 'weights': 'distance'}
0.890 (+/-0.028) for {'n_neighbors': 6, 'weights': 'distance'}
0.887 (+/-0.030) for {'n_neighbors': 5, 'weights': 'distance'}
0.885 (+/-0.030) for {'n_neighbors': 7, 'weights': 'distance'}
0.885 (+/-0.029) for {'n_neighbors': 8, 'weights': 'distance'}
0.884 (+/-0.030) for {'n_neighbors': 1, 'weights': 'distance'}
0.884 (+/-0.030) for {'n_neighbors': 1, 'weights': 'uniform'}
0.882 (+/-0.028) for {'n_neighbors': 2, 'weights': 'distance'}
0.881 (+/-0.029) for {'n_neighbors': 3, 'weights': 'uniform'}
0.873 (+/-0.030) for {'n_neighbors': 5, 'weights': 'uniform'}
0.870 (+/-0.033) for {'n_neighbors': 2, 'weights': 'uniform'}
0.869 (+/-0.029) for {'n_neighbors': 4, 'weights': 'uniform'}
0.864 (+/-0.032) for {'n_neighbors': 6, 'weights': 'uniform'}
0.864 (+/-0.030) for {'n_neighbors': 7, 'weights': 'uniform'}
0.854 (+/-0.037) for {'n_neighbors': 8, 'weights': 'uniform'}
```

Listing 2: Grid search cross validation performance output

This example shows only 18 different parameter sets for one classifier and one dataset. Thus, it is not possible to show the full performance outputs for each dataset, all classifiers and each parameter combination. Hence, only the best parameters with the chosen parameters and the achieved performances for each classifier and dataset will be stated subsequently. The choices for the ADFA-LD can be found in Table 7, the choices for the ICS-PSD-NvNA are stated in Table 8 and the choices for the ICS-PSD-NNvA can be found in Table 9.

	DT	k-NN	QDA	MLP	SVM	DTBoost	DTBagg	RForest
selected times	5 / 10	6 / 10	10 / 10	1 / 10	6 / 10	3 / 10	5 / 10	2 / 10
mean performance	0.8610	0.9000	0.7590	0.9010	0.8980	0.8940	0.9010	0.9070
standard deviation	0.0370	0.0320	0.0260	0.0230	0.0310	0.0400	0.0310	0.0210
criterion	entropy	-	-	-	-	entropy	entropy	entropy
min_sample_split	2	-	-	-	-	2	2	-
max_depth	None	-	-	-	-	None	None	-
min_samples_leaf	1	-	-	-	-	1	1	-
max_leaf_nodes	None	-	-	-	-	None	None	-
n_neighbors	-	3	-	-	-	-	-	-
weights	-	distance	-	-	-	-	-	-
reg_param	-	-	0.001	-	-	-	-	-
hidden_layer_sizes	-	-	-	500	-	-	-	-
solver	-	-	-	lbfgs	-	-	-	-
learning_rate	-	-	-	constant	-	-	-	-
kernel	-	-	-	-	rbf	-	-	-
C	-	-	-	-	100	-	-	-
gamma	-	-	-	-	1	-	-	-
n_estimators	-	-	-	-	-	90	80	70
max_features	-	-	-	-	-	-	-	0.4

Table 7: Best parameter values for each classifier for the ADFA-LD

	DT	k-NN	QDA	MLP	SVM	DTBoost	DTBagg	RForest
selected times	1 / 10	10 / 10	7 / 10	1 / 10	8 / 10	3 / 10	2 / 10	1 / 10
mean performance	0.9650	0.9880	0.9730	0.9780	0.9810	0.9700	0.9820	0.9850
standard deviation	0.0310	0.0160	0.0240	0.0210	0.0190	0.0280	0.0220	0.0200
criterion	entropy	-	-	-	-	entropy	entropy	entropy
min_sample_split	2	-	-	-	-	2	2	-
max_depth	None	-	-	-	-	None	None	-
min_samples_leaf	1	-	-	-	-	1	1	-
max_leaf_nodes	None	-	-	-	-	None	None	-
n_neighbors	-	2	-	-	-	-	-	-
weights	-	uniform	-	-	-	-	-	-
reg_param	-	-	10^(-7)	-	-	-	-	-
hidden_layer_sizes	-	-	-	300	-	-	-	-
solver	-	-	-	lbfgs	-	-	-	-
learning_rate	-	-	-	constant	-	-	-	-
kernel	-	-	-	-	rbf	-	-	-
C	-	-	-	-	1000	-	-	-
gamma	-	-	-	-	0.1	-	-	-
n_estimators	-	-	-	-	-	40	40	40
max_features	-	-	-	-	-	-	-	sqrt

Table 8: Best parameter values for each classifier for the ICS-PSD-NvNA

	DT	k-NN	QDA	MLP	SVM	DTBoost	DTBagg	RForest
selected times	5 / 10	10 / 10	10 / 10	1 / 10	10 / 10	X / 10	X / 10	2 / 10
mean performance	0.9280	0.9360	0.7170	0.9280	0.9260	0.9280	0.9530	0.9540
standard deviation	0.0220	0.0120	0.0290	0.0140	0.0200	0.0120	0.0120	0.0140
criterion	entropy	-	-	-	-	entropy	entropy	entropy
min_sample_split	2	-	-	-	-	2	2	-
max_depth	None	-	-	-	-	None	None	-
min_samples_leaf	1	-	-	-	-	1	1	-
max_leaf_nodes	None	-	-	-	-	None	None	-
n_neighbors	-	2	-	-	-	-	-	-
weights	-	distance	-	-	-	-	-	-
reg_param	-	-	0.1	-	-	-	-	-
hidden_layer_sizes	-	-	-	400	-	-	-	-
solver	-	-	-	lbfgs	-	-	-	-
learning_rate	-	-	-	constant	-	-	-	-
kernel	-	-	-	-	rbf	-	-	-
C	-	-	-	-	1000	-	-	-
gamma	-	-	-	-	1	-	-	-
n_estimators	-	-	-	-	-	30	50	40
max_features	-	-	-	-	-	-	-	sqrt

Table 9: Best parameter values for each classifier for the ICS-PSD-NNvA

Since this performance is based on the average score of the F-Measure and G-Mean metrics, we cannot compare this performances with the basic evaluation performances from the last chapter. However, the found parameters lay the foundation for all future tests and evaluations in the next subchapter, in which all performances will be evaluated by classifiers which are configured with these parameters. First, the three datasets will be evaluated with k-fold cross validation by the optimized classifiers and then the performance will be compared to the basic evaluation to demonstrate the performance improvement through the grid search. Then, all imbalanced data method tests will be executed and the achieved performances will be compared among each other to choose the best method for the smart grid hierarchy test implementation.

6.3 Imbalanced methods evaluation

The final set of classifiers for all tests with original data and imbalanced data methods are

- k-nearest neighbor (k-NN)
- Quadratic Discriminant Analysis (QDA)
- Multilayer Perceptrons (MLP)
- Support vector machine (SVM)
- AdaBoost with decision trees as base learners (DTBoost)
- Bagging with decision trees as base learners (DTBagg)
- Random forest (RForest)
- Plurality voting (PlurVt)
- Weighted voting (WeighVt)

Since both DTBoost and DTBagg are using decision trees as their base learners, the single decision tree classifier will not be used any longer. But therefore, the two voting classifiers, plurality and weighted voting, will be added to the set of classifiers.

As illustrated in Figure 38, the tests can be distinguished by the use of original data and sampled data. If the original data is used, a normal test or one of the three different used cost-sensitive learning tests (weighting, thresholding and cost classifiers) can be executed. If the data distribution is changed, various data sampling tests can be executed. The sampling methods used in this research project can be divided into 11 under-sampling methods, 3 over-sampling methods and 2 hybrid-sampling methods, all introduced previously.

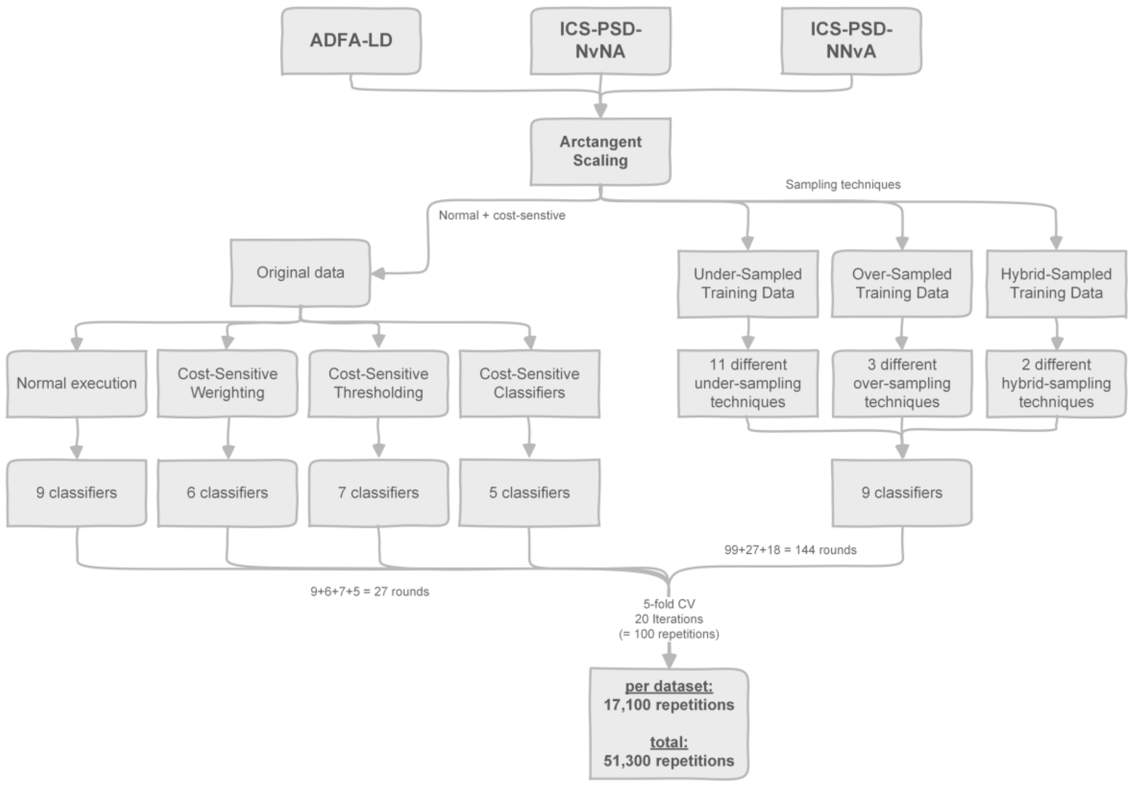


Figure 38: Test-setup for normal, cost-sensitive and sampling tests

However, each method uses several classifiers to evaluate the performance. While the normal execution and all sampling methods can use all 9 previously stated classifiers, the cost-sensitive learning methods have restricted possibilities. The cost-sensitive weighting method is only executable for selected classifiers such as decision trees and SVMs. Therefore, the weighted classifier set consists of the classifiers SVM, DTBoost, DTBagg, RForest and the two voting classifiers PlurVt and WeighVt. Since thresholding is only possible for classifiers which can produce probability outputs, the classifier set is restricted to the classifiers k-NN, MLP, QDA, SVM, DTBoost, DTBagg and RForest (voting classifiers cannot produce probability outputs). The cost-sensitive classifier set, with directly built-in cost-sensitivity, consists of a decision tree, bagging, pasting, random forests and random patches. This setup leads for each dataset to 20 different scenarios (1 normal execution scenario, 1 cost-sensitive weighting scenario, 1 cost-sensitive thresholding scenario, 1 cost-sensitive classifier set, 11 under-sampling methods, 3 over-sampling methods and 2 hybrid-sampling methods) and to 171 different rounds through the amount of used classifiers (9 classifiers for normal execution, 6 classifiers for cost-sensitive weightings, 7 classifiers for cost-sensitive thresholding, 5 cost-sensitive classifiers, 9 classifiers for 11 under-sampling methods, 3 over-sampling methods and 2 hybrid-sampling methods).

Each round is executed with a 5-fold cross validation and 20 repetitions to create robust classifiers. This means that each round needs 100 iterations to accomplish the evaluation and to obtain the performance. Consequently, each dataset needs 17,100 iterations to perform all different test setups. This leads to 51,300 total iterations.

To evaluate all classifiers and scenarios, two wrapper functions were built. The first wrapper function executes a full scenario and the second wrapper executes all scenarios one by one. So, the complete test setup including all evaluations for un-sampled data and original classifiers, cost-sensitive learning methods and sampling methods is illustrated as process flow in Figure 39.

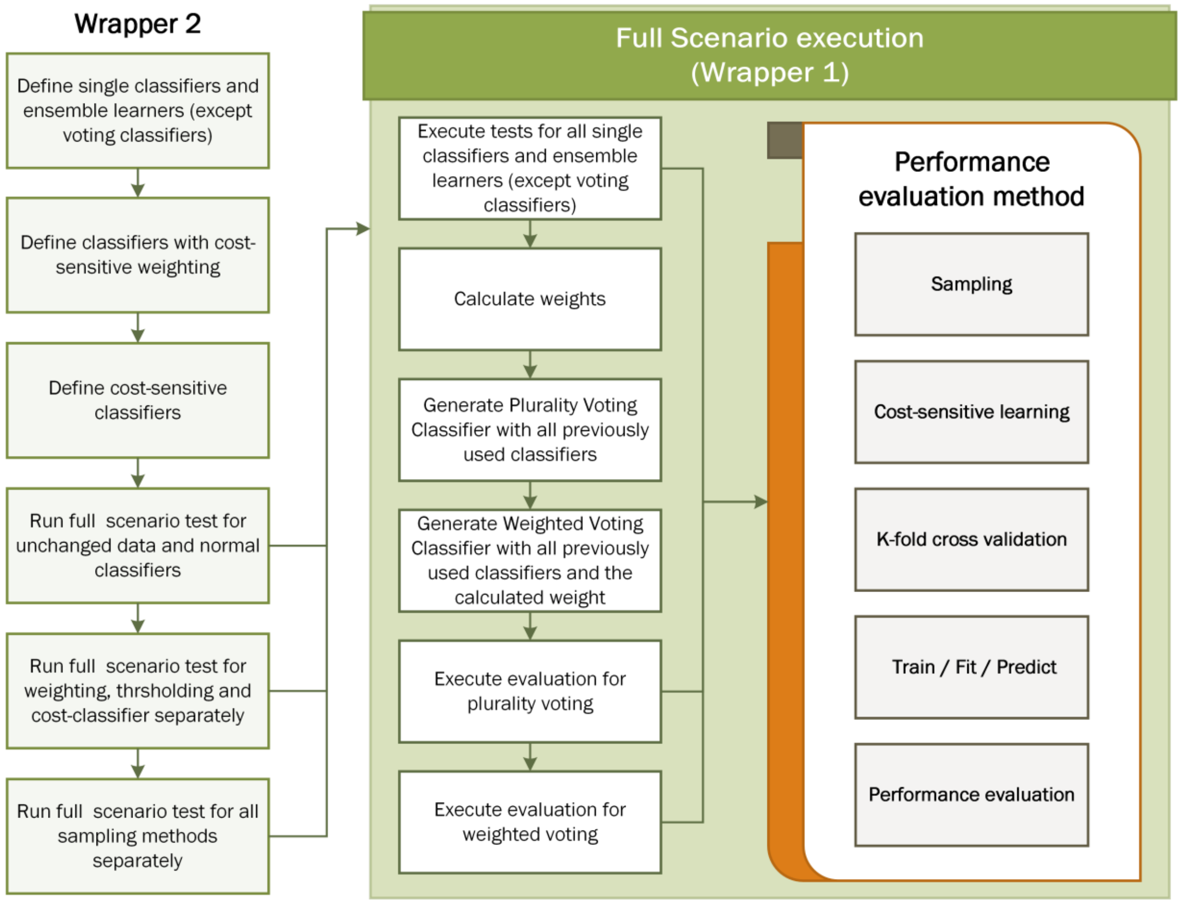

Wrapper 2
Define single classifiers and ensemble learners (except voting classifiers)
Define classifiers with cost-sensitive weighting
Define cost-sensitive classifiers
Run full scenario test for unchanged data and normal classifiers
Run full scenario test for weighting, thrsholding and cost-classifier separately
Run full scenario test for all sampling methods separately
Full Scenario execution (Wrapper 1)
Execute tests for all single classifiers and ensemble learners (except voting classifiers)
Calculate weights
Generate Plurality Voting Classifier with all previously used classifiers
Generate Weighted Voting Classifier with all previously used classifiers and the calculated weight
Execute evaluation for plurality voting
Execute evaluation for weighted voting
Performance evaluation method
Sampling
Cost-sensitive learning
K-fold cross validation
Train / Fit / Predict
Performance evaluation

Figure 39: Process of complete test setup for all evaluations

To execute all scenarios (normal execution, cost-sensitive learning methods and sampling techniques), wrapper 2 defines all single classifiers and ensemble learners, the single classifiers and ensemble learners with integrated costs and all cost-sensitive classifiers. Then, each scenario is executed one by one. First, the normal scenario without any changes at the data distribution or any cost-sensitive integrations is simulated.

Then, the three cost-sensitive learning scenarios (weighting, thresholding, cost-sensitive classifiers) are performed. Next, all different sampling technique scenarios including 11 under-sampling methods, 3 over-sampling methods and 2 hybrid-sampling methods are executed.

To execute a single scenario, wrapper 1 starts with executing the performance evaluation method (stated in Listing 8 in Appendix A) for all single classifiers and ensemble learners except the voting classifiers. This is necessary, since the weighted voting classifier needs the performance weights from the other classifiers to create the classifier. So, the weights based on the performances from all single classifiers and non-voting ensemble learners are calculated. Then, the plurality voting classifier with all used classifiers and the weighted voting classifier with the same classifiers and the calculated weights are created. Finally, the evaluation process (Listing 8 in Appendix A) is executed for the plurality and the weighted voting classifier. First in the evaluation process – similar to the basic evaluation – the function creates a dictionary with rates for all classifiers and also different dictionaries for the confusion matrix and the TPR and FPR performances for the ROC plot. Then, for n iterations the evaluation process is executed. The passed data and targets are split into k stratified and random sampled folds with the built-in function “StratifiedKFold” from Scikit-Learn in the module “model_selection”. Then, for each fold, the performance for the given scenario is executed. If it is a sampling scenario, the training data and training targets will be sampled with the passed method. Afterwards, for each classifier, it is determined if either a cost-sensitive classifier is used or not. If a cost-sensitive classifier is used, the related cost-matrix is generated and the cost-sensitive classifier is trained by the original training data. If no cost-sensitive classifier is used, the classifier is trained with the either sampled or original training data by the “fit” method. Since the cost-sensitive thresholding method changes the threshold once a classifier is trained, the next decision is to make at this point. So, if thresholding should be used, the next step instead of proceeding to the prediction is to substitute the training and test data by their predicted probabilities. Therefore, the built-in function “predict_proba” of the classifier is used (only available if the classifier can predict probabilities). Then, the related cost-matrix is generated and the thresholding class is trained by the predicted probabilities for the training data, the cost-matrix and the training targets. Finally, either the test data or the predicted probabilities are used for the prediction.

Next, the performances are evaluated and added to the rates dictionary. As soon as all folds and iterations for all classifiers are executed, the performances are divided by the iterations and finally returned. Then, to calculate various metrics for the gained rates, the function stated in Listing 9 in Appendix A is used. This function simply extracts the TP, FP, TN and FN to calculate all used performance metrics. Then the ACC, FPR, FNR, F1-Measure and G-Mean metrics are returned.

To generate a ROC plot with the collected TPR, FPR and AUC values, the function stated in Listing 10 in Appendix A is used. Therefore, the Matplotlib module "pyplot", denoted as "plt", is used. A diagonal line is added to represent a guessing classifier. Additional to the True Positive Rate on the y-axis and the False Positive Rate on the x-axis, the AUC performance metric is added in the lower right corner. Finally, the ROC curve is either showed or saved.

Normal scenario without imbalanced data methods

First of all, let us have a look at the performances achieved for the normal scenario, which was executed based on the process illustrated in Figure 34 without any adaptions. Thus, we can firstly compare the achieved performance to the basic evaluation and secondly compare the achieved performance of the ensemble learners to the single classifiers. As reminder and for an easier compare, all performances with arctangent scaling from the basic evaluation for all datasets are combined and stated in Table 10.

	ADFA-LD			*ICS-PSD-NvNA*			*ICS-PSD-NNvA*		
	TPR	**TNR**	**ACC**	**TPR**	**TNR**	**ACC**	**TPR**	**TNR**	**ACC**
k-NN	0.8491	0.9778	0.9616	0.9967	0.8481	0.9915	0.9423	0.7276	0.8948
MLP	0.8536	0.9832	0.9670	0.9967	0.8481	0.9915	0.9363	0.5358	0.8476
QDA	0.9741	0.4652	0.5291	0.9967	0.8135	0.9903	0.3950	0.9442	0.5166
linSVM	0.8183	0.9720	0.9527	0.9944	0.5423	0.9786	0.9604	0.1564	0.7823
rbfSVM	0.0009	0.9999	0.8746	1.0000	0.0000	0.9651	1.0000	0.0000	0.7785

Table 10: Basic evaluation results – all datasets with arctangent scaling

In general, all upcoming presented performances will consist of the metrics FPR, FNR, the ROC AUC (denoted as AUC), the ACC, the F1-Measure (denoted as F1) and the G-Mean (denoted as G). To compare these metrics with the metrics TPR, TNR and ACC from the basic evaluation, the FNR is simply to consider as the complementary metric to TPR and the FPR is simply to consider the complementary metric to TNR.

Now, the achieved performance for the ADFA-LD is stated in Table 11. As we can see, the overall performance improvement through the parameter grid search is quite good. For example, the k-NN classifier achieved a 2.50% higher TPR (1-FNR) and a 0.50% higher TNR (1-FPR). The MLP classifier has an 1.30% higher TPR and about the same TNR. Even if the ACC improvements for k-NN and MLP classifiers are not high, the performance for the minority class was improved remarkable. The QDA classifier performs still not well.

ADFA-LD		FPR	FNR	AUC	ACC	F1	G
Single classifiers	k-NN	0.0186	0.1251	0.9281	0.9680	0.8728	0.9266
	MLP	0.0165	0.1334	0.9250	0.9689	0.8746	0.9232
	QDA	0.1661	0.0319	0.9010	0.8507	0.6191	0.8985
	SVM	0.0133	0.1443	0.9212	0.9703	0.8783	0.9189
Ensemble classifiers	DTBoost	0.0135	0.1487	0.9189	0.9695	0.8751	0.9164
	DTBagg	0.0111	0.1444	0.9223	0.9722	0.8852	0.9199
	RForest	0.0097	0.1436	0.9233	0.9735	0.8901	0.9209
	PlurVt	0.0107	0.1176	0.9358	0.9759	0.9017	0.9343
	WeighVt	0.0099	0.1190	0.9355	0.9764	0.9034	0.9339

Table 11: K-fold cross validation results - ADFA-LD

But we could actually achieve the expected performance for the SVM classifier. Now the SVM is an equally good classifier and achieves noteworthy results. However, this is the first time we see the performances of the ensemble learners in comparison with the single classifiers. The DTBagg and RForest classifiers were able to improve the detection rate for the majority class but at the expense of a reduced detection rate for the minority class. The ACC metric for the DTBagg and RForest ensemble learners are higher than each single classifier but on the other hand each AUC metric of these ensemble learners is lower than the AUC metric of the k-NN and MLP classifiers. The SVM and DTBoost classifiers have about the same performance. However, the voting classifiers were able to achieve the best performances by far. They could achieve approximately a 3 to 8% lower FPR and a 1 to 3% lower FNR in comparison to the k-NN, MLP and SVM single classifiers. But they even outperform the other ensemble learners. In terms of the AUC metric, the best classifier was the plurality voting classifier with 93.58%.

The achieved performance for the ICS-PSD-NvNA is stated in Table 12. Even if the performance for this classifier was already very good, through the parameter grid search the performance was even increased.

ICS-PSD-NvNA		FPR	FNR	AUC	ACC	F1	G
Single classifiers	k-NN	0.0428	0.0030	0.9771	0.9956	0.9977	0.9769
	MLP	0.0844	0.0038	0.9559	0.9934	0.9966	0.9551
	QDA	0.0870	0.0120	0.9505	0.9854	0.9924	0.9498
	SVM	0.0512	0.0073	0.9707	0.9912	0.9954	0.9705
Ensemble classifiers	DTBoost	0.0451	0.0006	0.9771	0.9979	0.9989	0.9769
	DTBagg	0.0694	0.0010	0.9648	0.9967	0.9983	0.9642
	RForest	0.0656	0.0004	0.9669	0.9973	0.9986	0.9664
	PlurVt	0.0387	0.0006	0.9803	0.9980	0.9990	0.9801
	WeighVt	0.0425	0.0004	0.9785	0.9982	0.9991	0.9783

Table 12: K-fold cross validation results - ICS-PSD-NvNA

While the detection rate for the majority class is still close to 100%, the detection rate for the minority class was improved by 11% for the k-NN classifier and by nearly 7% for the MLP classifier. The QDA classifier detected 1% less majority instances but therefor was able to detect 10% more minority instances. Again, the SVM classifier improved the performance drastically. The ensemble learners raised the already very high detection rate for the majority class to over 99.90%. While the DTBoost and both voting classifiers were able to improve the detection rate for the minority class to over 95% (similar to k-NN), DTBagg and RForest detected just about 93% of the minority instances. This is better than the MLP and QDA classifier but worse than the others. Again, the plurality voting classifier achieved with 98.03% the best performance in terms of the AUC metric.

The achieved performance for the ICS-PSD-NNvA is stated in Table 13. The behavior is similar to the other two datasets. Through grid search a quite good performance improvement was achieved.

ICS-PSD-NNvA		FPR	FNR	AUC	ACC	F1	G
Single classifiers	k-NN	0.1370	0.0393	0.9119	0.9391	0.9609	0.9106
	MLP	0.1585	0.0429	0.8993	0.9315	0.9560	0.8974
	QDA	0.1779	0.3946	0.7138	0.6534	0.7312	0.7055
	SVM	0.1935	0.0195	0.8935	0.9419	0.9634	0.8893
Ensemble classifiers	DTBoost	0.1915	0.0411	0.8837	0.9256	0.9525	0.8805
	DTBagg	0.1188	0.0152	0.9330	0.9618	0.9757	0.9315
	RForest	0.1173	0.0150	0.9338	0.9623	0.9760	0.9324
	PlurVt	0.1205	0.0186	0.9304	0.9588	0.9737	0.9290
	WeighVt	0.1297	0.0171	0.9266	0.9580	0.9733	0.9249

Table 13: K-fold cross validation results - ICS-PSD-NNvA

The k-NN classifier improved its accuracy by over 4% by detecting 1.80% more majority instances and 13.50% more minority instances. The MLP classifier performed even better and achieved an 8.40% higher accuracy by detecting 2% more majority instances and nearly 31% more minority instances. The performance for the QDA classifier is still bad. Again, the performance for the SVM classifier was improved to an equal level. The detection rate for majority instances is better but in expense of the detection rate for the minority instances. Except DTBoost, the ensemble learner performances are a lot better than the performances of the single classifiers. They improved the amount of detected majority instances by 0.50 to 2.50% and the amount of detected minority instances up to 7.60%. This time, a non-voting ensemble learner, namely the RForest classifier, achieved the best AUC with 93.38%.

All in all, it was shown that the parameter grid search was very effective to boost the performance of the classifiers. The behavior was for all datasets similar but only a different range of improvements is to consider. Beside that, the ensemble learners showed also for all datasets an even better performance than the single classifiers.

Next, all the imbalanced data methods will be executed and compared to these performances. Finally, the best method will be selected for the smart grid hierarchy setup.

Imbalanced data methods performances

To this point, all performances were achieved with the help of the NumPy [64], SciPy [65], Matplotlib [67] and especially the Scikit-Learn [66] package. To execute the cost-sensitive learning methods, the CostsensitiveClassification package [36] includes the stated cost-sensitive classifiers and was also used to perform thresholding. To add weights to the classifiers, the classifiers within the Scikit-Learn package could be used. Finally, for all sampling methods, the Imbalanced-Learn package [68] was used.

Before the performances from the imbalanced data methods will be compared to the normal scenario, let us consider that all the previous evaluations were performed by the straightforward process illustrated in Figure 34. To execute the imbalanced methods, some individual changes in this process were necessary. Therefore, each individual changed process will be illustrated separately. The changes will be outlined with red color.

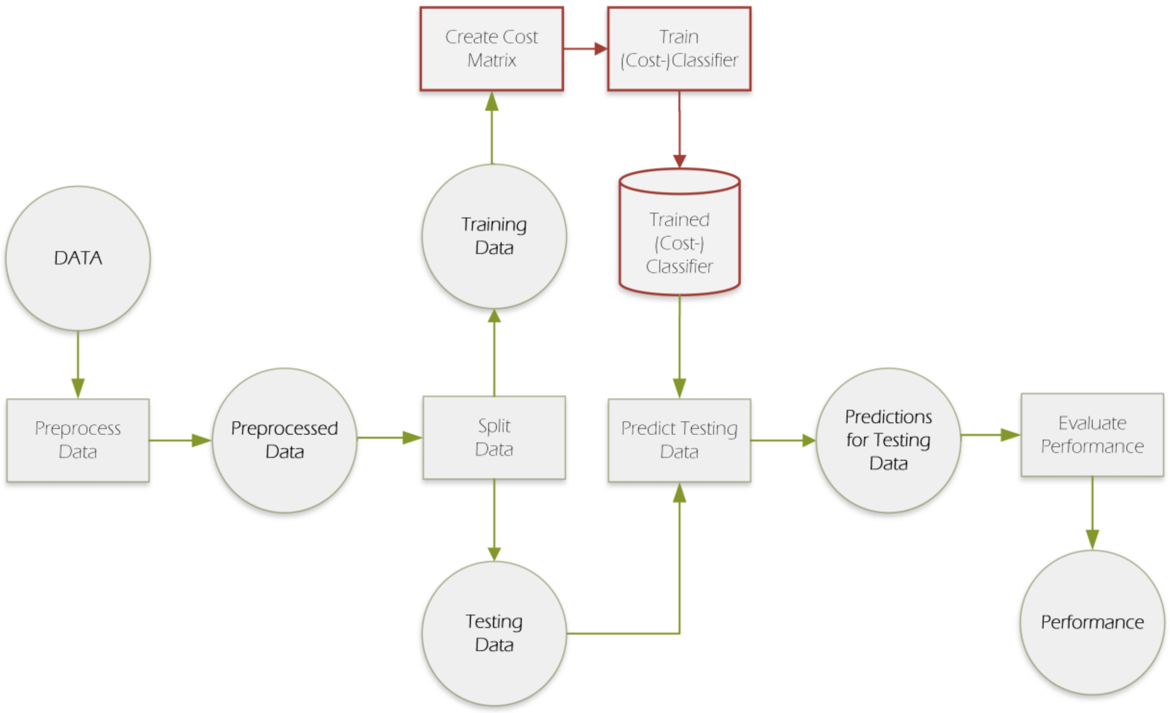


Figure 40: Cost-sensitive weighting and classifiers evaluation process

Let us start with the changes for cost-sensitive weighting and cost-sensitive classifiers. The changes for both methods are illustrated in Figure 40, since the changes are similar and only different classifiers are used. So, the only relevant change is that a cost-matrix is created which is either directly integrated into the classifier (weighting) or passed to train the cost-sensitive classifier. The rest of the process remains the same.

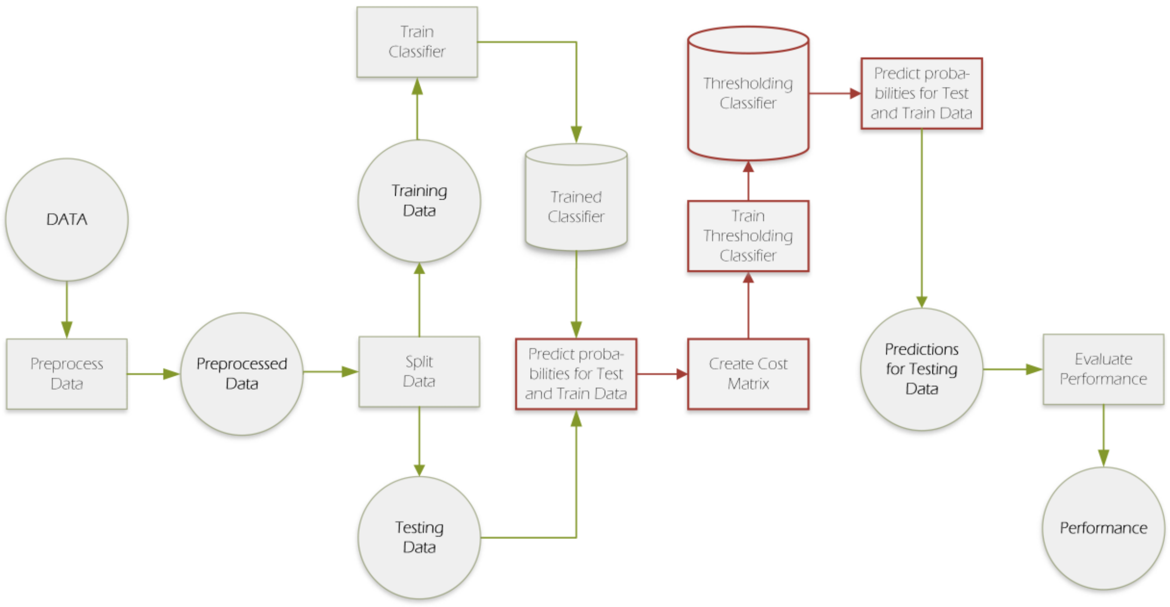


Figure 41: Cost-sensitive thresholding evaluation process

More changes are necessary for cost-sensitive thresholding (see Figure 41). Once the original classifier is trained, the classifier is used to predict probabilities for test and training data. Then, a cost-matrix is created and the thresholding classifier is trained by the predicted training data probabilities and the cost-matrix. Finally, the targets are predicted with the thresholding classifier by the predicted test data probabilities.

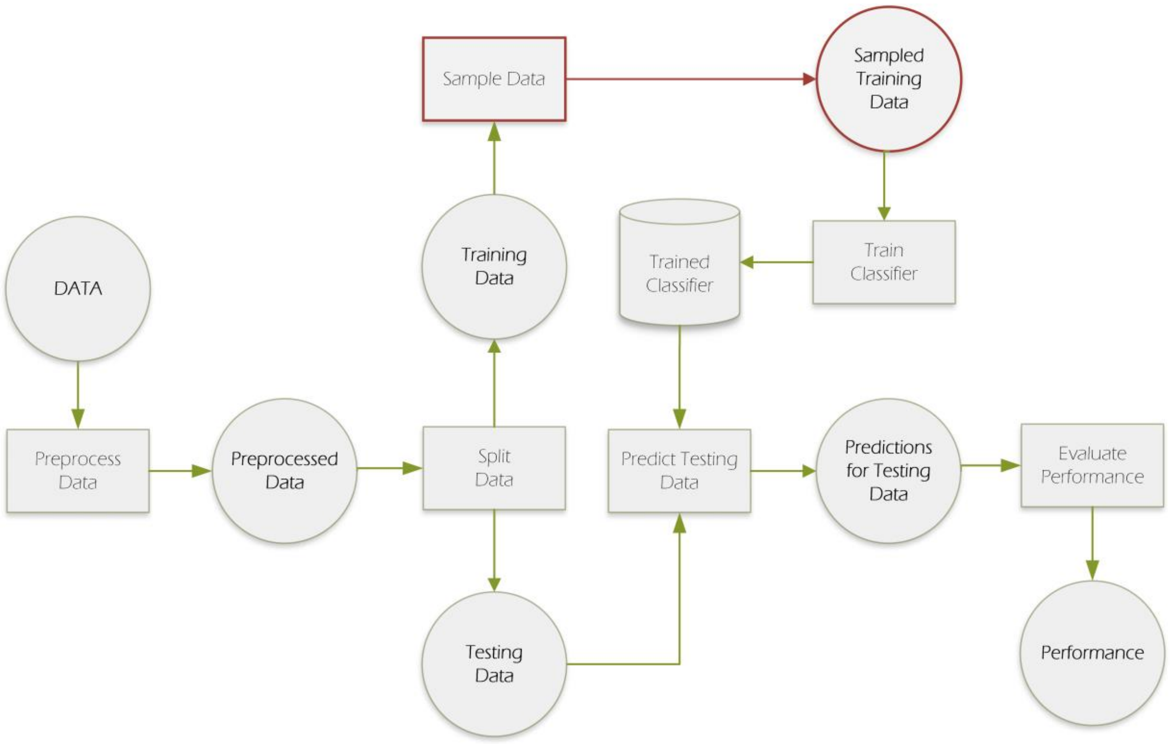


Figure 42: Sampling evaluation process

To execute the evaluation process with sampling methods, only a single change is necessary. As illustrated in Figure 42, the training data is sampled with the according sampling method and then the classifier is trained with the sampled training data.

As stated, there are 20 different scenarios for each dataset. So, not each performance result will be stated in the following pages. Instead, only the best method for each sampling type and the three cost-sensitive methods will be shown and compared to the normal scenario. Nevertheless, all performance results can be found in the Appendix. The results for the ADFA-LD are stated in Appendix B, the results for the ICS-PSD-NvNA can be found in Appendix C and the results for the ICS-PSD-NNvA are listed in Appendix D.

Since the comparison of all performance metrics would be beyond the scope, the AUC, FPR and FNR metrics were chosen for the comparison. The AUC metric was chosen because the ACC and also the F1-Measure are too biased related to a single class. Since the G-mean evaluates similar metrics (square of TPR multiplied by TNR) than the AUC (curve area for TPR and FPR with different thresholds) and the results for both metrics are quite similar, the AUC was chosen because the performance can be illustrated additionally with the ROC plot. The FPR and the FNR metrics were chosen as addition to the AUC metric for a better comparison of the detection rate for both the minority and majority class. Based on this metrics, the best under-, over- and hybrid-sampling methods were chosen.

After the presentation of the results, each dataset will be individually compared by the different imbalanced data methods. Additionally, the results from the imbalanced data methods from each dataset will be compared to the normal scenario. Then, the different imbalanced data methods will be compared over all datasets. Finally, the best method will be chosen. This method will then be used for the final test, the smart grid hierarchy model.

Now, let us finally come to the results for the imbalanced data methods. The results for all test scenarios for the ADFA-LD are stated in Table 14, the results for the ICS-PSD-NvNA are stated in Table 15 and the results for the ICS-PSD-NNvA are stated in Table 16.

If we compare the imbalanced data method performances for the ADFA-LD (Table 14) to the results from the normal scenario (Table 11), then all sampling methods were able to improve the AUC score. On the other hand, nearly all cost-sensitive learning methods decreased the performance. Let us start with the RENN under-sampling method, which was chosen as the best under-sampling method. This method was able to improve the AUC performance of each single classifier and ensemble learner. An improvement of the AUC between 0.30% and 1.50% was achieved by a more balanced detection rate. For the normal scenario, the common (without QDA) false detection rate for the majority class (FPR) is between 1 and 2% and for the minority class (FNR) between 11 and 15%. With RENN under-sampling, the common FPR is now between 4 and 7% and the FNR remains only between 2 and 10%. This means, that there are less detected majority instances but therefore a lot more detected minority instances. To highlight this, the average AUC was raised from 92.35% (FPR 2.99%, FNR 12.31%) to 93.30% (FPR 6.47%, FNR 6.94%). This behavior is exactly as expected and improves the overall detection rate (based on the AUC metric) while a much better detection rate for the minority class was achieved.

ADFA-LD		FPR	FNR	AUC	ACC	F1	G
Under-sampling: **RepeatedEdited-NearestNeighbour (RENN)**	k-NN	0.0663	0.0603	0.9367	0.9345	0.7824	0.9367
	MLP	0.0583	0.0660	0.9379	0.9407	0.7980	0.9379
	QDA	0.1633	0.0288	0.9040	0.8536	0.6245	0.9015
	SVM	0.0542	0.0777	0.9341	0.9428	0.8018	0.9340
	DTBoost	0.0546	0.1018	0.9218	0.9394	0.7881	0.9215
	DTBagg	0.0435	0.0883	0.9341	0.9509	0.8230	0.9338
	RForest	0.0386	0.0839	0.9387	0.9557	0.8384	0.9385
	PlurVt	0.0514	0.0613	0.9437	0.9474	0.8173	0.9437
	WeighVt	0.0523	0.0564	0.9456	0.9472	0.8174	0.9456
Over-sampling: **ADASYN**	k-NN	0.0407	0.0735	0.9429	0.9552	0.8383	0.9428
	MLP	0.0347	0.0849	0.9402	0.9590	0.8484	0.9399
	QDA	0.2209	0.0392	0.8699	0.8019	0.5487	0.8652
	SVM	0.0339	0.0929	0.9366	0.9587	0.8462	0.9361
	DTBoost	0.0229	0.1176	0.9297	0.9652	0.8641	0.9285
	DTBagg	0.0274	0.0887	0.9420	0.9649	0.8670	0.9415
	RForest	0.0237	0.0928	0.9418	0.9677	0.8755	0.9411
	PlurVt	0.0283	0.0697	0.9510	0.9665	0.8746	0.9508
	WeighVt	0.0254	0.0745	0.9500	0.9684	0.8802	0.9497
Hybrid-sampling: **SMOTETomek**	k-NN	0.0385	0.0721	0.9447	0.9573	0.8448	0.9446
	MLP	0.0263	0.1092	0.9322	0.9633	0.8588	0.9313
	QDA	0.1458	0.0464	0.9039	0.8667	0.6420	0.9025
	SVM	0.0257	0.1200	0.9272	0.9625	0.8547	0.9260
	DTBoost	0.0196	0.1284	0.9260	0.9668	0.8680	0.9244
	DTBagg	0.0212	0.1098	0.9345	0.9677	0.8734	0.9334
	RForest	0.0175	0.1056	0.9385	0.9715	0.8872	0.9375
	PlurVt	0.0212	0.0867	0.9460	0.9706	0.8862	0.9455
	WeighVt	0.0199	0.0874	0.9464	0.9717	0.8898	0.9458
Cost-sensitive weighting	SVM	0.0219	0.2355	0.8713	0.9514	0.7976	0.8647
	DTBoost	0.0150	0.1543	0.9154	0.9676	0.8674	0.9127
	DTBagg	0.0118	0.1451	0.9215	0.9715	0.8826	0.9191
	RForest	0.0101	0.1442	0.9228	0.9731	0.8885	0.9204
	PlurVt	0.0083	0.1638	0.9140	0.9722	0.8830	0.9106
	WeighVt	0.0093	0.1451	0.9228	0.9737	0.8905	0.9203
Cost-sensitive thresholding	k-NN	0.0415	0.0791	0.9397	0.9538	0.8332	0.9395
	MLP	0.0183	0.1830	0.8993	0.9610	0.8402	0.8955
	QDA	0.0195	0.1218	0.9293	0.9677	0.8719	0.9279
	SVM	0.0183	0.1831	0.8993	0.9610	0.8402	0.8955
	DTBoost	0.0183	0.1831	0.8993	0.9610	0.8402	0.8955
	DTBagg	0.0195	0.1221	0.9292	0.9677	0.8719	0.9278
	RForest	0.0195	0.1221	0.9292	0.9677	0.8719	0.9278
Cost-sensitive classifiers	DT	0.0256	0.4955	0.7395	0.9155	0.5996	0.7011
	Bagging	0.0864	0.1014	0.9061	0.9117	0.7185	0.9061
	Pasting	0.0965	0.0879	0.9078	0.9046	0.7056	0.9078
	RForest	0.0879	0.0839	0.9141	0.9126	0.7244	0.9141
	RPatches	0.1215	0.0753	0.9016	0.8843	0.6671	0.9013

Table 14: Results for imbalanced data methods – ADFA-LD

Although the RENN under-sampling for the ADFA-LD achieved the best detection rates for the minority class, with ADASYN over-sampling the AUC score could be raised again. So, an average AUC of 93.38% was achieved with an average FPR of 5.09% and an average FNR of 8.15%. Comparing the best single classifiers, the normal scenario achieved an AUC of 93.58% with the plurality voting classifier, the RENN under-sampling achieved an AUC of 94.56% with the weighted voting classifier and through ADASYN over-sampling the plurality voting classifier achieved an AUC score of 95.10%. However, SMOTETomek hybrid-sampling achieved also for each classifier a better AUC score than the normal scenario. But, the detection rate for the minority class is worse than the detection rate with under- or over-sampling. Anyway, a slightly better detection rate for the majority class was achieved. Nevertheless, the best classifier from SMOTETomek hybrid-sampling achieved just a 0.10% better performance than the best classifier from RENN under-sampling and a 0.50% worse performance than ADASYN over-sampling.

The cost-sensitive weighting had only insignificant impact on the performance. While individual classifiers are slightly better, other classifiers are slightly worse and the voting classifiers achieve also a worse performance since there are less single classifiers to consider for the voting. So, the overall performance of cost-sensitive weighting is worse than the normal scenario. Also, with cost-sensitive thresholding, the performance of each classifier but k-NN and QDA was decreased. The k-NN classifier with thresholding achieved an over 1% higher AUC by detecting 2.30% less majority instances but nearly 5% more minority instances. The curious behavior of the QDA classifier continues, since this classifier improved the AUC score by nearly 3% with an over 14% higher detection rate for the majority class and a 9% lower detection rate for the minority class. Anyway, the overall lower performance through thresholding seems very inconsistent. But the decreasing performance is unfortunately continued with the cost-sensitive classifiers. While the decision tree with built-in cost-sensitiveness performs at the very worst, the other cost-sensitive classifiers can achieve at least an AUC score barely over 90% with at least more balanced detection rates. All in all, the cost-sensitive classifiers are very disappointing, since cost-sensitive weighting achieved only a similar performance to the normal scenario and thresholding and the cost-sensitive classifiers even decreased the performance. However, the sampling methods compensate this bad performance. Regarding the best method among them, the decision is not easy to make.

Compared only based on the AUC metric, ADASYN over-sampling has definitely the best performance even when RENN under-sampling achieved just a 0.50% lower AUC score while having generally a higher detection rate for the minority class. Nevertheless, ADASYN over-sampling achieved with the plurality voting classifier the best AUC performance for the ADFA-LD. A comparison between the normal scenario and ADASYN over-sampling is illustrated through ROC plots in Figure 43. A close inspection reveals that the higher TPR emerges as a direct consequence of the lower FPR.

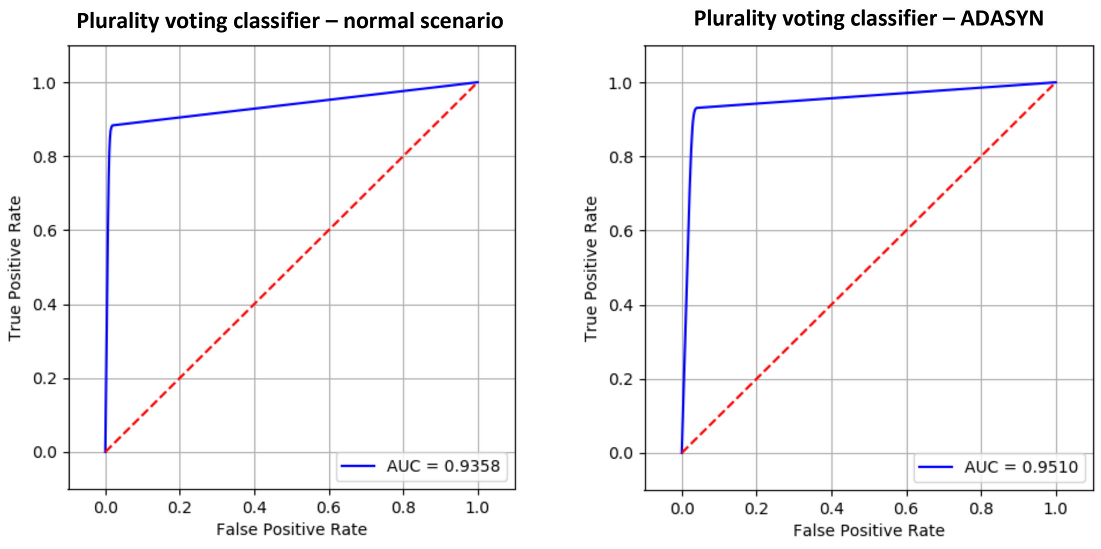


Figure 43: ROC plots –Best normal vs. imbalanced method (ADFA-LD)

The performance of the normal scenario for the ICS-PSD-NvNA is stated in Table 12. Compared to the results from the imbalanced data methods from Table 15, the behavior is similar to the ADFA-LD. But the first difference is that not the RENN method performed best among the under-sampling methods. Instead, the CNN under-sampling method achieved the best results. Anyway, the behavior is similar. The detection rate for the minority class was improved by up to 5% which leads for example for the k-NN classifier to a detection rate of over 99% while the detection rate for the majority class was decreased by up to 14% in an exceptional case and by up to 3.50% for the rest of the classifiers. This leads in general to a better AUC score for nearly all classifiers.

Again, ADASYN was chosen as the best over-sampling method. In average, the detection rate of the majority class was just slightly decreased while the detection rate for the minority class was improved remarkable.

ICS-PSD-NvNA		FPR	FNR	AUC	ACC	F1	G
Under-sampling: **Condensed-NearestNeighbour (CNN)**	k-NN	0.0069	0.1433	0.9247	0.8614	0.9227	0.9223
	MLP	0.0520	0.0358	0.9560	0.9636	0.9808	0.9560
	QDA	0.0633	0.0121	0.9623	0.9861	0.9928	0.9620
	SVM	0.0234	0.0414	0.9676	0.9592	0.9784	0.9676
	DTBoost	0.0142	0.0256	0.9800	0.9748	0.9868	0.9801
	DTBagg	0.0176	0.0222	0.9800	0.9780	0.9885	0.9801
	RForest	0.0188	0.0207	0.9801	0.9793	0.9892	0.9802
	PlurVt	0.0150	0.0152	0.9847	0.9848	0.9920	0.9849
	WeighVt	0.0162	0.0120	0.9857	0.9878	0.9936	0.9859
Over-sampling: **ADASYN**	k-NN	0.0355	0.0044	0.9799	0.9945	0.9971	0.9799
	MLP	0.0723	0.0063	0.9607	0.9914	0.9956	0.9602
	QDA	0.0780	0.0871	0.9174	0.9132	0.9531	0.9174
	SVM	0.0442	0.0083	0.9737	0.9905	0.9950	0.9736
	DTBoost	0.0228	0.0019	0.9876	0.9974	0.9986	0.9876
	DTBagg	0.0246	0.0029	0.9862	0.9964	0.9981	0.9862
	RForest	0.0231	0.0018	0.9874	0.9974	0.9987	0.9875
	PlurVt	0.0205	0.0020	0.9886	0.9974	0.9986	0.9887
	WeighVt	0.0159	0.0015	0.9912	0.9980	0.9990	0.9913
Hybrid-sampling: **SMOTETomek**	k-NN	0.0321	0.0041	0.9819	0.9950	0.9974	0.9818
	MLP	0.0540	0.0063	0.9698	0.9920	0.9959	0.9695
	QDA	0.1043	0.0100	0.9428	0.9867	0.9931	0.9417
	SVM	0.0540	0.0064	0.9698	0.9920	0.9958	0.9695
	DTBoost	0.0280	0.0014	0.9852	0.9977	0.9988	0.9852
	DTBagg	0.0341	0.0035	0.9812	0.9955	0.9977	0.9811
	RForest	0.0309	0.0022	0.9834	0.9968	0.9983	0.9833
	PlurVt	0.0182	0.0020	0.9898	0.9974	0.9987	0.9899
	WeighVt	0.0231	0.0013	0.9877	0.9979	0.9989	0.9877
Cost-sensitive weighting	SVM	0.0633	0.0057	0.9655	0.9922	0.9960	0.9650
	DTBoost	0.0436	0.0008	0.9778	0.9977	0.9988	0.9775
	DTBagg	0.0789	0.0016	0.9597	0.9957	0.9978	0.9590
	RForest	0.0780	0.0007	0.9607	0.9966	0.9983	0.9599
	PlurVt	0.0286	0.0009	0.9852	0.9982	0.9991	0.9852
	WeighVt	0.0393	0.0003	0.9802	0.9983	0.9991	0.9800
Cost-sensitive thresholding	k-NN	0.0393	0.0029	0.9789	0.9958	0.9978	0.9787
	MLP	0.0208	0.4022	0.7885	0.6111	0.7480	0.7651
	QDA	0.0373	0.0728	0.9449	0.9284	0.9615	0.9448
	SVM	0.0208	0.4022	0.7885	0.6111	0.7480	0.7651
	DTBoost	0.0208	0.4022	0.7885	0.6111	0.7480	0.7651
	DTBagg	0.0393	0.0029	0.9789	0.9958	0.9978	0.9787
	RForest	0.0393	0.0029	0.9789	0.9958	0.9978	0.9787
Cost-sensitive classifiers	DT	0.5477	0.0160	0.7181	0.9655	0.9822	0.6671
	Bagging	0.0740	0.0283	0.9488	0.9701	0.9843	0.9486
	Pasting	0.0546	0.0313	0.9570	0.9679	0.9831	0.9570
	RForest	0.0572	0.0282	0.9573	0.9708	0.9847	0.9572
	RPatches	0.0497	0.0310	0.9596	0.9684	0.9834	0.9596

Table 15: Results for imbalanced data methods – ICS-PSD-NvNA

For example, the weighted voting classifier, which is the best individual classifier for the ICS-PSD-NvNA, increased the AUC score through oversampling from 97.85% (FPR 4.25%, FNR 0.04%) to 99.12% (FPR 1.59%, FNR 0.15%) with nearly the same detection rate for the majority class. This is a desirable result since the accuracy stays the same while a crucial increased detection rate for the minority class was achieved. But also, the rest of the classifiers (except QDA) improved their performances, whereby the ensemble learners were able to raise the AUC score the most (over 2%). Also, the SMOTETomek hybrid-sampling method improved the performance of nearly all classifiers and again the ensemble learners improved their already superior performance at most. The best classifier within the hybrid-sampling method is the plurality voting classifier with an AUC of 98.98%, which is just 0.14% lower than the best classifier within over-sampling.

The cost-sensitive learning methods were again not able to keep up with the sampling methods. The cost-sensitive weighting showed about the same performance than the normal scenario. Only the voting classifiers were slightly worse due to the small number of participating classifiers. While the cost-sensitive thresholding method were able to balance the detection rates for the k-NN, QDA, DTBagg and RForest classifiers a bit, the performance for the MLP, SVM and DTBoost classifiers decreased drastically. Finally, the cost-sensitive classifiers caused an overall worse AUC score but therefore a more balanced detection rate. But again, the decision tree classifier is the worst classifier.

However, the choice of the best method was not as difficult as for the ADFA-LD. The ADASYN over-sampling method achieved the best performance in all categories.

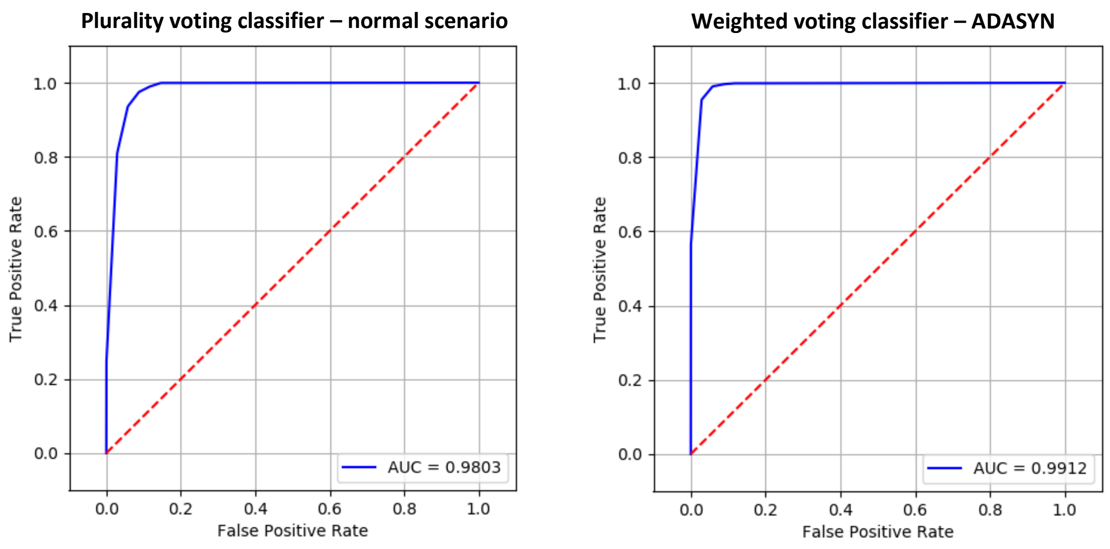


Figure 44: ROC plots – Best normal vs. imbalanced method (ICS-PSD-NvNA)

A comparison between the normal scenario and the ADASYN over-sampling is illustrated through ROC plots in Figure 44. Even if the AUC score was improved by only 1.09%, one can see how close this comes to an ideal performance.

The last dataset, the ICS-PSD-NNvA is compared between the performances of the normal scenario from Table 13 and the performances of the imbalanced data methods from Table 16. Since the behavior is very similar to the other two datasets, the performances for this dataset will be compared less detailed. This time, NCR under-sampling was the best under-sampling method and was able to improve the AUC score for nearly each classifier. Again, the under-sampling method was able to detect the most minority instances compared to over-sampling and hybrid-sampling. Anyway, ADASYN over-sampling achieved again the best performance among the over-sampling methods and gained also the overall best performance. Instead of one of the voting classifiers, the random forest classifier achieved the best performance among the different classifiers. So, the best AUC score was improved from 93.38% (normal scenario) to 95.08% (ADASYN over-sampling). Also, similar to the previous evaluations, SMOTETomek was the best hybrid-sampling method and performed in terms of AUC better than NCR under-sampling and only slightly worse than ADASYN over-sampling.

While the cost-sensitive weighting method is comparable to the normal scenario, cost-sensitive thresholding and the cost-sensitive classifiers stepped completely out of the line. Thresholding tuned the threshold for the k-NN, MLP, QDA and SVM classifiers in such a way, that all minority instances were detected but no single majority instance. This time, not only the cost-sensitive decision tree classifier performs badly but all cost-sensitive classifiers are in an agreement about a terrible performance.

However, a comparison between the normal scenario and the best method, the ADASYN over-sampling, is illustrated through ROC plots in Figure 45. Now we can see that the minority and the majority classes are interchanged since the curve behaves exactly inversely since the TPR decreases while the FPR improves. However, the AUC score was improved by 1.70%.

ICS-PSD-NNvA		**FPR**	**FNR**	**AUC**	**ACC**	**F1**	**G**
Under-sampling: **Neighbourhood-CleaningRule (NCR)**	k-NN	0.0917	0.1046	0.9018	0.8982	0.9320	0.9018
	MLP	0.1179	0.0897	0.8962	0.9040	0.9366	0.8961
	QDA	0.3039	0.3047	0.6957	0.6955	0.7805	0.6957
	SVM	0.1620	0.0566	0.8907	0.9201	0.9484	0.8891
	DTBoost	0.1521	0.0680	0.8899	0.9134	0.9437	0.8890
	DTBagg	0.0830	0.0494	0.9338	0.9432	0.9630	0.9336
	RForest	0.0785	0.0455	0.9380	0.9472	0.9657	0.9378
	PlurVt	0.0849	0.0624	0.9264	0.9327	0.9559	0.9263
	WeighVt	0.0856	0.0634	0.9255	0.9316	0.9552	0.9254
Over-sampling: **ADASYN**	k-NN	0.1260	0.0443	0.9148	0.9376	0.9598	0.9139
	MLP	0.1327	0.0574	0.9049	0.9259	0.9519	0.9042
	QDA	0.8764	0.0725	0.5255	0.7494	0.8521	0.3386
	SVM	0.1617	0.0246	0.9069	0.9451	0.9651	0.9043
	DTBoost	0.1820	0.0903	0.8639	0.8894	0.9276	0.8627
	DTBagg	0.0932	0.0574	0.9247	0.9347	0.9574	0.9245
	RForest	0.0477	0.0508	0.9508	0.9499	0.9672	0.9508
	PlurVt	0.1007	0.0318	0.9338	0.9530	0.9697	0.9331
	WeighVt	0.1022	0.0330	0.9324	0.9516	0.9689	0.9317
Hybrid-sampling: **SMOTETomek**	k-NN	0.1162	0.0511	0.9163	0.9345	0.9575	0.9158
	MLP	0.1440	0.0562	0.8999	0.9243	0.9510	0.8988
	QDA	0.0675	0.6118	0.6603	0.5088	0.5517	0.6017
	SVM	0.1824	0.0243	0.8966	0.9407	0.9624	0.8931
	DTBoost	0.1703	0.0701	0.8798	0.9077	0.9401	0.8784
	DTBagg	0.0785	0.0343	0.9436	0.9559	0.9715	0.9433
	RForest	0.0756	0.0321	0.9462	0.9583	0.9731	0.9459
	PlurVt	0.0852	0.0329	0.9409	0.9555	0.9713	0.9406
	WeighVt	0.0988	0.0286	0.9363	0.9559	0.9716	0.9357
Cost-sensitive weighting	SVM	0.1959	0.0184	0.8929	0.9423	0.9636	0.8885
	DTBoost	0.1057	0.1347	0.8798	0.8717	0.9131	0.8797
	DTBagg	0.1345	0.0135	0.9260	0.9597	0.9744	0.9241
	RForest	0.1294	0.0111	0.9297	0.9627	0.9763	0.9278
	PlurVt	0.0985	0.0197	0.9409	0.9629	0.9762	0.9401
	WeighVt	0.1425	0.0094	0.9240	0.9611	0.9754	0.9216
Cost-sensitive thresholding	k-NN	0.0960	0.0852	0.9094	0.9124	0.9421	0.9094
	MLP	0.0000	1.0000	0.5000	0.2215	nan	0.0000
	QDA	0.0000	1.0000	0.5000	0.2215	nan	0.0000
	SVM	0.0000	1.0000	0.5000	0.2215	nan	0.0000
	DTBoost	0.0000	1.0000	0.5000	0.2215	nan	0.0000
	DTBagg	0.1759	0.0244	0.8999	0.9421	0.9633	0.8967
	RForest	0.1759	0.0244	0.8999	0.9421	0.9633	0.8967
Cost-sensitive classifiers	DT	0.6166	0.0817	0.6509	0.7998	0.8772	0.5934
	Bagging	0.1453	0.2745	0.7901	0.7541	0.8212	0.7874
	Pasting	0.1453	0.2745	0.7901	0.7541	0.8212	0.7874
	RForest	0.1453	0.2745	0.7901	0.7541	0.8212	0.7874
	RPatches	0.1400	0.2490	0.8055	0.7751	0.8387	0.8036

Table 16: Results for imbalanced data methods – ICS-PSD-NNvA

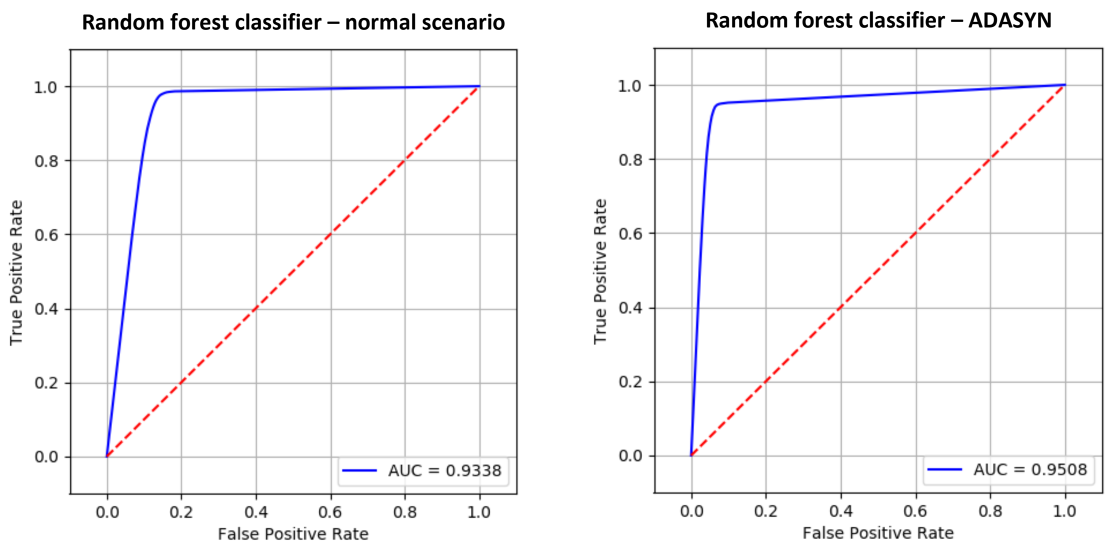


Figure 45: ROC plots – Best normal vs. imbalanced method (ICS-PSD-NNvA)

In general, the cost-sensitive classifiers are basically good for balancing the detection rates respectively improve the detection rate for the minority class but they are prone to over-balance the rates and so it is possible that the performance drops heavily. The sampling methods provide a very constant improvement in comparison to the normal scenario. The under-sampling methods achieved an overall higher AUC score and were mostly able to detect the most minority class instances among the sampling methods. On the other hand, the over-sampling method is the superior method and can achieve every time the best scores. Last but not least, the hybrid-sampling method keeps the performance somewhere between under- and over-sampling. In general, not only over-sampling is the superior method, especially ADASYN performed outstanding and achieved the best performances overall for each of the three datasets. While this is the same for SMOTETomek hybrid-sampling, the best under-sampling method was different for each dataset. However, it is suggested to use ADASYN over-sampling for the smart grid IDS implementation, which will be introduced subsequently, since it is the most promising imbalanced data method.

7 Hierarchical smart grid IDS communication system

In this chapter, a hierarchical smart grid IDS will be built to simulate a communication flow. To evaluate the performance of this communication system, the ADFA-LD will be used to provide normal and attack data. One simulation round will use the unchanged data set and another simulation round will use the previously chosen best imbalanced data method, namely ADASYN over-sampling. Then, the two performances for the smart grid IDS will be compared.

To execute a single simulation round, a process as illustrated in Figure 46 is executed. The yellow outlined tasks state optional processes, since the first round is executed without over-sampling. Anyway, in each simulation round the test data is predicted by a hierarchical smart grid IDS communication system (outlined in red).

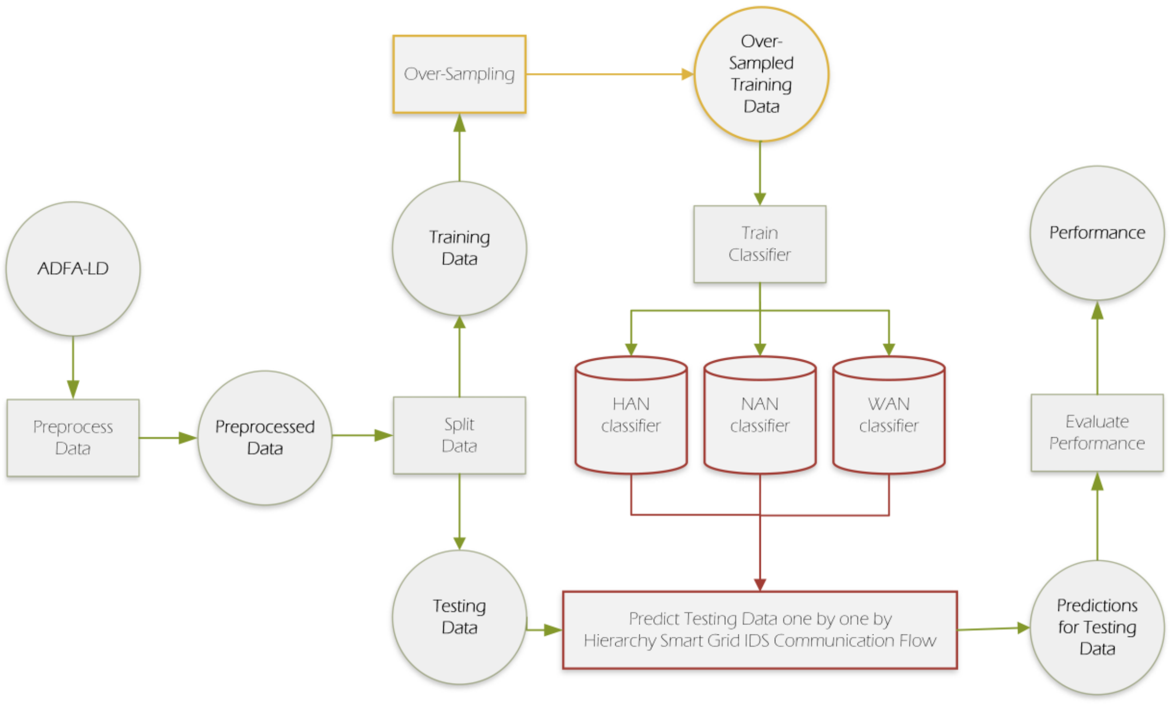


Figure 46: Hierarchical smart grid IDS simulation process

So, the red outlined part represents a three-layer hierarchy smart grid architecture which will be built similar to the hierarchical smart grid IDS system as illustrated in Figure 5 and described in [18]. Since the used architecture and communication flow for this hierarchical smart grid IDS are very complex, a prototypical implementation with a more simplified communication flow will be created. Therefore, the created prototype uses only a single IDS at each layer and is simulated only by if/else decisions.

For a better understanding of the created three-layer smart grid architecture, a single decision process for a single data instance is illustrated in Figure 47.

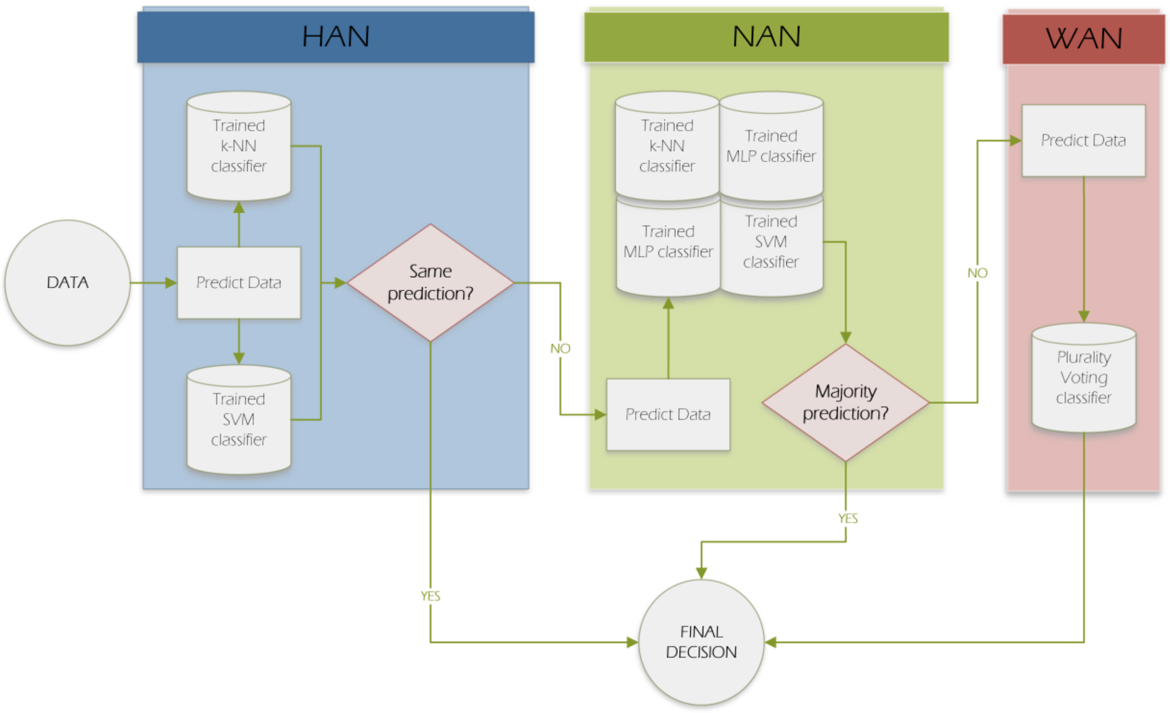


Figure 47: Hierarchical smart grid communication flow and decision process

This process illustrates the communication flow of a single data instance and at this point we assume that the classifiers are already trained. Now, a single data instance is passed at first to the HAN layer. Since the devices used in HANs (e.g., smart meters) have usually a low-performance, just the two fastest (measured during the method screening process) but still well performing classifiers were chosen for the HAN IDS. The used classifiers are the k-NN and the SVM classifier. The single data instance is then predicted with both classifiers. If both the k-NN and the SVM classifiers predict the same class, then this prediction is a final decision. If they disagree in their decision, the data instance is passed to the next layer.

Within the NAN layer, the data instance is now predicted from an IDS with four different classifiers, namely k-NN, MLP, SVM and RForest. If the majority of the classifiers decide for one class, then this class is the final decision. On the other hand, if two of the classifiers predict one class and the other two classifiers predict the other class, then the data instance is again forwarded to the next and last layer. In the WAN layer, the data instance is predicted by the plurality voting classifier. Since this is the last layer, no further decision is necessary. Consequently, the predicted class is the final decision.

However, to perform the simulation at all, an own module “sg_hierarchy_simulation” was developed. This module contains two scripts “model” and “simulation”. The “model” script contains the training and the prediction functions and the “simulation” script contains the simulation routine to execute the “model” functions for n rounds and to evaluate the performances.

The function for a complete simulation round (either with or without over-sampling) is stated in Listing 11 in Appendix A. Basically, the data is scaled and performance values are created. Then, the process of splitting data, over-sampling (if true), training and predicting is executed for n rounds to finally get an average performance. However, the training and prediction functions are the most important parts for the hierarchical smart grid IDS (red outline section in Figure 46) and will be explained subsequently.

The function to create the classifiers and to train them is stated in Listing 3.

```python
def build_and_train_classifier_2HAN_4NAN_voteWAN(data, targets):
    # load HAN classifier + training
    HAN_clf = __get_classifier(use_kNN=True, use_SVM=True)
    for key, classifier in HAN_clf.items():
        classifier.fit(data, targets)

    # load NAN classifier + training
    NAN_clf = __get_classifier(use_kNN=True, use_MLP=True, use_SVM=True,
           use_RForest=True)
    for key, classifier in NAN_clf.items():
        classifier.fit(data, targets)

    # load WAN classifier + training
    WAN_clf = __get_classifier(use_kNN=True, use_QDA=True, use_MLP=True,
           use_SVM=True, use_DTBoost=True, use_DTBagg=True,
           use_RForest=True)
    estimators = []
    for key, classifier in sorted(WAN_clf.items()):
        estimators.append((key, classifier))
    WAN_clf = VotingClassifier(estimators=estimators, voting='hard')
    WAN_clf.fit(data, targets)

    return HAN_clf, NAN_clf, WAN_clf
```

Listing 3: Create and train smart grid hierarchy classifiers

This function creates and trains the classifiers only for the introduced architecture (2 HAN classifiers, 4 WAN classifiers and a single WAN voting classifier). If a different architecture should be used (e.g., 2 HAN classifiers, 6 WAN classifiers and 2 WAN classifier), it is only necessary to create another function which creates the classifiers for the desired architecture.

To finally evaluate the full test dataset, the process of a single data decision, as previously described and illustrated in Figure 47, needs to be repeated for all test data instances. Additionally, a few evaluation variables are used to evaluate the performance of the whole training data set (e.g., for each layer, the correct predicted instances and amount of passed instances are measured). The function to execute these tasks is stated in Listing 4.

```
def model_evaluation_2HAN_4NAN_voteWAN(data, targets, HAN_clf, NAN_clf,
WAN_clf):
    predictions = []
    HAN_correct = 0
    NAN_correct = 0
    WAN_correct = 0
    from_han_to_nan = 0
    from_nan_to_wan = 0

    pos = 0
    for single_instance in data:
        # HAN prediction
        HAN_predicts = []
        for key, classifier in HAN_clf.items():
            HAN_predicts.append(classifier.predict(single_data))

        if HAN_predicts[0] == HAN_predicts[1]:
            prediction = HAN_predicts[0]
            predictions.append(prediction)
            if prediction == targets[pos]:
                HAN_correct += 1
        else:
            # HAN classifier in disagreement
            from_han_to_nan += 1

            NAN_predicts = []
            for key, classifier in NAN_clf.items():
                NAN_predicts.append(classifier.predict(single_data))

            negatives = 0
            positives = 0
            for NAN_predict in NAN_predicts:
                if NAN_predict == 0:
                    negatives += 1
                else:
                    positives += 1

            if positives > negatives:
                prediction = 1
                if prediction == targets[pos]:
                    NAN_correct += 1
            elif negatives > positives:
                prediction = 0
                if prediction == targets[pos]:
                    NAN_correct += 1
            else:
                # equal votes for NAN classifier
                from_nan_to_wan += 1
                prediction = WAN_clf.predict(single_data)
                if prediction == targets[pos]:
                    WAN_correct += 1

            predictions.append(prediction)

        pos += 1
```

```
# calculate accuracies
HAN_total = len(targets) - from_han_to_nan
NAN_total = from_han_to_nan - from_nan_to_wan
WAN_total = from_nan_to_wan

HAN_accuracy = HAN_correct / HAN_total
NAN_accuracy = NAN_correct / NAN_total
if WAN_total > 0:
    WAN_accuracy = WAN_correct / WAN_total
else:
    WAN_accuracy = 1

return np.asarray(predictions, dtype=int), HAN_total, HAN_accuracy,
 NAN_total, NAN_accuracy, WAN_total, WAN_accuracy
```

Listing 4: Hierarchical smart grid model evaluation process

To execute this function successfully, the following parameters are necessary:

- **data [numpy matrix]:** Numpy dataset matrix (columns=features)
- **targets [numpy matrix]:** Numpy target matrix (each row = one target)
- **HAN_clf [dictionary]:** All trained classifiers used for the HAN layer. The name of the classifiers as key for each entry and the classifier itself as value.
- **NAN_clf [dictionary]:** All trained classifiers used for the NAN layer. The name of the classifiers as key for each entry and the classifier itself as value.
- **WAN_clf [dictionary]:** All trained classifiers used for the WAN layer. The name of the classifiers as key for each entry and the classifier itself as value.

At the end, this function returns the amount of classified data instances at each layer, the achieved accuracy at each layer and the list with the final predictions. These values are used to evaluate the performance of the hierarchy itself and to compare the performances with the simple train and fit tests from the previous chapter and between the normal simulation and the simulation with over-sampled training data. Again, to use a different architecture, it is only necessary to create another function with the desired if/else architecture.

However, the processed data instances and the accuracy for these instances are stated in Table 17 for each hierarchy layer.

	HAN predictions	**HAN accuracy**	**NAN predictions**	**NAN accuracy**	**WAN predictions**	**WAN accuracy**
ADFA-LD original	1161.54	97.92%	19.14	83.32%	10.32	74.22%
ADDA-LD sampled	1143.56	97.03%	32.04	82.92%	15.40	69.57%

Table 17: Data instances and accuracy for the hierarchy layers in the smart grid IDS

So, in total 1,191 test data instances were processed. Since the smart grid IDS detection process was repeated 100 times, the amount of processed data for each hierarchy layer has decimals. However, the number of processed instances within the HAN layer is very high. In the simulation process with un-sampled training data, on average 1,161.54 data instances were processed in the HAN. This means, that less than 30 instances were passed to next layer (or less than 2.50%). For the simulation round with over-sampled training data the number of processed instances at the HAN layer is 1,143.56 averaged. Less than 48 data instances or less than 4% were passed to the next layer. But this fact is only remarkable since the total accuracy at this layer is very high. The simulation with the original training data achieved an accuracy of 97.92% within the HAN layer and the simulation with the over-sampled training data achieved an accuracy of 97.03%. The first comparison of this accuracy already implies, that the behavior for the over-sampled data might be similar as described in the previous chapter. However, for the scenario with the original data, the NAN layer processed over 19 data instances from the approximately remaining 30 (approximately 65% of the remaining instances). This was accomplished with a total accuracy of 83.32%. On the other hand, the scenario with the over-sampled training data processed over 32 instances from the averaged 47.44 remaining instances (approximately 67.50% of the remaining instances). The total accuracy at the NAN layer for these instances is 82.92%. The WAN layer processed only 10.32 data instances for the round with original data with an achieved accuracy of 74.22% and only 15.40 data instances for the round with over-sampled data with an achieved accuracy of 69.57%. So, even the supposed difficult to predict instances which were passed to the next layers were predicted with an acceptable accuracy.

Finally, the performance metrics for the complete smart grid communication model are stated in Table 18. As assumed, the behavior of the smart grid communication system is similar to a common prediction process from the previous chapter.

	FPR	**FNR**	**AUC**	**ACC**	**F1**	**G**
ADFA-LD original	1.18%	11.96%	93.46%	97.48%	89.68%	93.27%
ADDA-LD sampled	3.23%	6.97%	94.90%	96.30%	86.34%	94.88%

Table 18: Performance results for the smart grid IDS

For the simulation with unchanged methods, the detection rate for the minority class is at 88.04% and the detection rate for the majority class is at 98.82%. This leads to an AUC score of 93.46%. Through over-sampling the training data for the simulation process the AUC score was raised to 94.90%. This score was achieved with a detection rate of 93.03% for the minority class and a detection rate of 96.77% for the majority class. So, the 1.44% higher AUC score with 5% more detected minority instances is achieved at the expense of a 1.18% lower accuracy score.

The ROC plots for both scenarios are illustrated Figure 48. These plots show visually the change of the detection rates for minority and majority class which led to a higher AUC score.

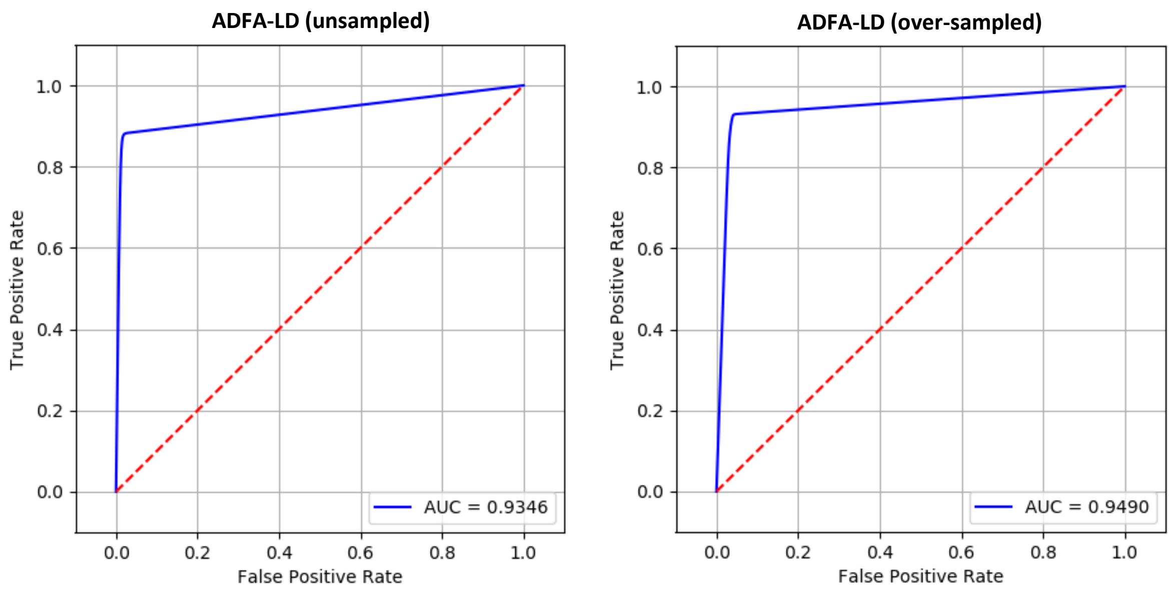


Figure 48: ROC plots for the best performers in the smart grid IDS

Finally, if we compare this performance to the results from the previous chapter, we can see that the performance of the smart grid IDS communication system achieves the same results as the respective best classifier. This means, the smart grid IDS communication system achieves for both scenarios a similar performance as their respective voting classifiers from the previous chapter.

8 Conclusion

The goal of this research project was to investigate imbalanced data methods and to use these methods for anomaly detection in smart grids. For this purpose, two different datasets were chosen, whereby one dataset consists of normal network data and cyber-attacks and the other dataset consists of normal, natural and attack events. Both datasets were evaluated with a selection of classifiers. These classifiers were tested with their default parameters and then a grid search was executed to improve their performance. In general, it is a promising idea to execute a grid search within the hyper-parameter space since the performance were improved for all datasets and all classifiers. Then, the performance for all datasets were evaluated with various imbalanced data methods. Therefore, sampling methods such as under-sampling, over-sampling and hybrid-sampling and also cost-sensitive learning methods such as weighting, thresholding and cost-sensitive classifiers were tested. While the performance for the cost-sensitive learning methods were disappointing, the sampling methods fulfilled their expectations. Especially through over-sampling they were able to improve the detection rate of the minority class while the detection rate for majority class nearly remained. Overall, this behavior led to an improved AUC score.

After the exploration of all methods, the best method for the ADFA-LD was chosen to build a smart grid IDS. To build the smart grid IDS, a hierarchical three-layer communication system were constructed with if/else conditions. Then, both the best common method and the best chosen imbalanced data method were evaluated for the built smart grid IDS. The expectation was, that the imbalanced data method outperforms existing approaches. If we consider the AUC score, this goal was definitely reached. The behavior for the three-layer smart grid IDS with ADASYN over-sampling was similar to a common evaluation and so the detection rate for the minority class was improved while the performance for the majority class was just slightly worse. This led to an overall better AUC score of 1.50%.

But the hierarchical smart grid IDS itself was also able to improve the overall performance. So, the performance for both methods match their respective best performing classifier, namely the plurality voting classifier. Consequently, a higher performance was achieved only through the use of the hierarchical three-layer smart grid IDS. This performance was again improved through ADASYN over-sampling which led finally to the overall best performance.

To extend this work, one might experiment with various combinations of classifiers and structures for the hierarchical smart grid IDS to improve the performance. Another possibility would be to add some ensemble solution classifiers to the classifier set or to add some classifiers from the algorithm-level solutions (e.g., kernel-based learning framework, one-class learning approach or active learning approach). Finally, one could change the prototypical implementation with if/else conditions to a real smart grid communication system as created in [18].

9 Bibliography

[1] Y. Mo, T. H.-J. Kim, K. Brancik, D. Dickinson, H. Lee, A. Perrig und B. Sinopoli, „Cyber–Physical Security of a Smart Grid Infrastructure,“ *Proceedings of the IEEE,* Bd. 100, Nr. 1, pp. 195 - 209, January 2012.

[2] R. Berthier, W. H. Sanders und H. Khurana, „Intrusion Detection for Advanced Metering Infrastructures: Requirements and Architectural Directions,“ in *IEEE International Conference on Smart Grid Communications,* Gaithersburg, MD, USA, 2010.

[3] 50Hertz Transmission GmbH, „50Hertz Stromkreuzungen,“ [Online]. Available: http://www.50hertz.com/Portals/3/Galerien/Broschueren/50Hertz_BR_Stromkreuzungen-DE-Web.pdf. [Accessed 16 May 2017].

[4] Deutsches Bundesministerium für Wirtschaft und Energie, „Dossier Netze und Netzausbau,“ [Online]. Available: https://www.bmwi.de/Redaktion/DE/Dossier/netze-und-netzausbau.html. [Accessed 16 May 2017].

[5] M. Orcutt, „How a Smarter Grid Can Prevent Blackouts - and Cut Your Energy Bills,“ August 2010. [Online]. Available: http://www.popularmechanics.com/science/energy/a6013/how-a-smarter-grid-can-prevent-blackouts/. [Accessed 16 May 2017].

[6] V. C. Gungor, D. Sahin, T. Kocak, S. Ergut, C. Buccella, C. Cecati und G. P. Hancke, „Smart Grid Technologies: Communication Technologies and Standards,“ *IEEE Transactions on Industrial Informatics,* Bd. 7, Nr. 4, pp. 529-539, November 2011.

[7] H. Farhangi, „The Path of the Smart Grid,“ *IEEE power & energy magazine,* pp. 18-28, 2010.

[8] National Energy Technology Laboratory und Office of Electricity Delivery & Energy Reliabily, „Advanced Metering Infrastructure,“ U.S. Department of Energy, 2008.

[9] R. Ma, H.-H. Chen und Y.-R. Huang, „Smart Grid Communication: Its Challenges and Opportunities,“ *IEEE Transactions on Smart Grid,* Bd. 4, Nr. 1, pp. 36-46, March 2013.

[10] Y. Yan, H. Sharif und D. Tipper, „Survey on Smart Grid Communication Infrastructures: Motivations, Requirements and Challenges,“ *IEEE COMMUNICATIONS SURVEYS & TUTORIALS,* Bd. 1, pp. 5-20, 2013.

[11] H. F. Tipton, Official (ISC)2 Guide to the CISSP CBK, Bd. 2, Boca Raton, FL: Tayler & Francis Group, 2010.

[12] SANS Institute, „Intrusion Detection Systems: Definition, Need and Challenges,“ SANS Institute, Bethesda, 2001.

[13] SANS Institute, „Understanding Intrusion Detection Systems,“ SANS Institute, Bethesda, 2001.

[14] T. Boyles, CCNA Security Study Guide, Indianapolis, IN: Wiley, 2010.

[15] F. M. Tabrizi und K. Pattabiraman, „A Model-Based Intrusion Detection System for Smart Meters,“ in *International Symposium on High-Assurance Systems Engineering,* Miami Beach, FL, USA, 2014.

[16] R. Mitchell und I.-R. Chen, „Behavior-Rule Based Intrusion Detection Systems for Safety Critical Smart Grid Applications,“ *IEEE Transactions on Smart Grid,* pp. 1254-1263, 29 April 2013.

[17] O. Linda, M. Manic und T. Vollmer, „Improving cyber-security of smart grid systems via anomaly detection and linguistic domain knowledge,“ in *International Symposium on Resilient Control Systems*, Salt Lake City, UT, USA, 2012.

[18] Y. Zhang, L. Wang, W. Sun, R. C. Green II und M. Alam, „Distributed Intrusion Detection System in a Multi-Layer Network Architecture of Smart Grids,“ *IEEE Transactions on Smart Grid,* pp. 796-808, 29 July 2011.

[19] R. O. Duda, P. E. Hart und D. G. Stork, Pattern Classification (2nd Edition), Wiley-Interscience, 2000.

[20] A. K. Jain, R. P. Duin und J. Mao, „Statistical Pattern Recognition: A Review,“ *IEEE Transactions on Pattern Analysis and Machine Intelligence,* Bd. 22, Nr. 1, pp. 4-37, January 2000.

[21] A. R. Webb, Statistical Pattern Recognition, Second Edition, John Wiley & Sons, Ltd., 2002.

[22] C. M. Bishop, Pattern Recognition and Machine Learning, Springer, 2006.

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher und P, „Tuning the hyper-parameters of an estimator,“ [Online]. Available: http://scikit-learn.org/stable/modules/grid_search.html. [Accessed 31 May 2017].

[24] T. Whitney, „Classification,“ [Online]. Available: http://trevorwhitney.com/data_mining/classification. [Accessed 2 June 2017].

[25] OpenCV dev team, „Introduction to Support Vector Machines,“ [Online]. Available: http://docs.opencv.org/2.4/doc/tutorials/ml/introduction_to_svm/introduction_to_svm.html. [Accessed 2 June 2017].

[26] Z.-H. Zhou, Ensemble Methods: Foundations and Algorithms, Boca Raton, FL: Taylor & Francis Group, LLC, 2012.

[27] H. He und E. A. Garcia, „Learning from Imbalanced Data,“ *IEEE Transactions on Knowledge and Data Engineering,* Bd. 21, Nr. 9, pp. 1263-1284, September 2009.

[28] B. Zhu, B. Baesens und S. K. vanden Broucke, „An empirical comparison of techniques for the class imbalance problem in churn prediction,“ *Information Sciences,* Bd. 408, pp. 84-99, October 2017.

[29] S. Shekarforoush, R. Green und R. Dyer, „Classifying Commit Messages: A Case Study in Resampling Techniques,“ in *International Joint Conference on Neural Networks*, Anchorage, Alaska, 2017.

[30] G. Lemaitre , F. Nogueira und C. K. Aridas, „Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning,“ *Journal of Machine Learning Research,* Bd. 18, Nr. 17, pp. 1-5, 2017.

[31] I. Tomek, „An Experiment with the Edited Nearest-Neighbor Rule,“ *IEEE Transactions on Systems, Man, and Cybernetics,* Bde. %1 von %2SMC-6, Nr. 6, pp. 448-452, June 1976.

[32] B. Zadrozny, J. Langford und N. Abe, „Cost-sensitive learning by cost-proportionate example weighting,“ in *Third IEEE International Conference on Data Mining*, Melbourne, FL, USA, 2003.

[33] P. Branco, L. Torgo und R. P. Ribeiro, „A Survey of Predictive Modeling on Imbalanced Domains,“ *ACM Computing Surveys*, Bd. 49, Nr. 2, p. Article No. 31, November 2016.

[34] P. Domingos, „MetaCost: a general method for making classifiers cost-sensitive,“ *KDD '99 Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 155-164, August 1999.

[35] V. S. Sheng und C. X. Ling, „Thresholding for Making Classifiers Cost-sensitive,“ *Proceedings of the 21st national conference on Artificial intelligence*, Bd. 1, pp. 476-481, July 2006.

[36] A. Correa Bahnsen, „Example-Dependent Cost-Sensitive Classification with Applications in Financial Risk Modeling and Marketing Analytics,“ University of Luxembourg, 2015.

[37] G. Creech und J. Hu, „Generation of a new IDS test dataset: Time to retire the KDD collection,“ in *Wireless Communications and Networking Conference (WCNC)*, Shangai, China, 2013.

[38] Canonical Ltd. Ubuntu, „Ubuntu,“ [Online]. Available: https://www.ubuntu.com/. [Accessed 10 May 2017].

[39] R. C. Borges, J. M. Beaver, M. A. Buckner, T. Morris, U. Adhikari und S. Pan, „Machine Learning for Power System Disturbance and Cyber-attack Discrimination,“ in *International Symposium on Resilient Control Systems*, Denver, Colorado, 2014.

[40] S. Pan, T. Morris und U. Adhikari, „Developing a Hybrid Intrusion Detection System Using Data Mining for Power Systems,“ *IEEE Transactions on Smart Grid*, Bd. 6, Nr. 6, pp. 3104-3113, 18 March 2015.

[41] S. Pan, T. Morris und U. Adhikari, „Classification of Disturbances and Cyber-Attacks in Power Systems Using Heterogeneous Time-Synchronized Data,“ *IEEE Transactions on Industrial Informatics*, Bd. 11, Nr. 3, pp. 650-662, 8 April 2015.

[42] S. Pan, T. Morris und U. Adhikari, „A Specification-based Intrusion Detection Framework for Cyber-physical Environment in Electric Power System,“ *International Journal of Network Security*, Bd. 17, Nr. 2, pp. 174-188, March 2015.

[43] I. University of California, „The UCI KDD Archive - KDD Cup 1998 Data,“ , 16 February 1999. [Online]. Available: https://kdd.ics.uci.edu/databases/kddcup98/kddcup98.html. [Accessed 10 May 2017].

[44] University of New Mexico - Computer Science Department, „Computer Immune Systems - Data Sets and Software,“ [Online]. Available: https://www.cs.unm.edu/~immsec/data-sets.htm. [Accessed 10 May 2017].

[45] „The Apache Software Foundation,“ [Online]. Available: https://www.apache.org/. [Accessed 10 May 2017].

[46] The PHP Group, „PHP,“ [Online]. Available: https://secure.php.net/. [Accessed 10 May 2017].

[47] Oracle Corporation, „MySQL,“ [Online]. Available: https://www.mysql.com/. [Accessed 10 May 2017].

[48] J. Posel und J. Reynolds, „File Transfer Protocol (FTP) - RFC 959,“ October 1985. [Online]. Available: https://www.ietf.org/rfc/rfc959.txt. [Accessed 10 May 2017].

[49] T. Ylonen, SSH Communications Security Corp, C. E. Lonvick und Cisco Systems Inc., „The Secure Shell (SSH) Transport Layer Protocol - RFC 4253,“ January 2006. [Online]. Available: https://tools.ietf.org/rfc/rfc4253.txt. [Accessed 10 May 2017].

[50] Tiki Software Community Association, „tiki wiki cms groupware,“ [Online]. Available: https://tiki.org/tiki-index.php. [Accessed 10 May 2017].

[51] M. Xie und J. Hu, „Evaluating Host-Based Anomaly Detection Systems: A Preliminary Analysis of ADFA-LD,“ in *International Congress on Image and Signal Processing (CISP),* Hangzhou, China, 2013.

[52] Australian Defence Force Academy, „The ADFA Intrusion Detection Datasets,“ 31 October 2013. [Online]. Available: https://www.unsw.adfa.edu.au/australian-centre-for-cyber-security/cybersecurity/ADFA-IDS-Datasets/. [Accessed 11 May 2017].

[53] S. Forrest, S. A. Hofmeyr, A. Somayaji und T. A. Longstaff, „A sense of self for Unix processes,“ in *IEEE Symposium on Security and Privacy,* 1996.

[54] G. Creech, „Developing a high-accuracy cross platform Host-Based Intrusion Detection System capable of reliably detecting zero-day attacks,“ The University of New South Wales, 2013.

[55] G.-B. Huang, Q.-Y. Zhu und C.-K. Siew, „Extreme learning machine: a new learning scheme of feedforward neural networks,“ in *IEEE International Joint Conference on Neural Networks,* Budapest, Hungary, 2004.

[56] L. Rabiner und B. Juang, „An introduction to hidden Markov models,“ *IEEE ASSP Magazine,* Bd. 3, Nr. 1, pp. 4-16, 1 January 1986.

[57] G. Creech und J. Hu, „A Semantic Approach to Host-based Intrusion Detection Systems Using Contiguous and Discontiguous System Call Patterns,“ *IEEE Transactions on Computers,* Bd. 63, Nr. 4, pp. 807-819, 24 January 2013.

[58] Cisco and/or its affiliates, „SNORT,“ [Online]. Available: https://www.snort.org/. [Accessed 11 May 2017].

[59] R. Gerhards, „The Syslog Protocol - RFC 5424,“ March 2009. [Online]. Available: https://tools.ietf.org/rfc/rfc5424.txt. [Accessed 11 May 2017].

[60] T. Morris, „Industrial Control System (ICS) Cyber Attack Datasets,“ [Online]. Available: https://sites.google.com/a/uah.edu/tommy-morris-uah/ics-data-sets. [Accessed 11 May 2017].

[61] Python Software Foundation, „Python,“ [Online]. Available: https://www.python.org/. [Accessed 12 May 2017].

[62] Continuum Analytics, Inc., „Anaconda,“ [Online]. Available: https://www.continuum.io/. [Accessed 12 May 2017].

[63] JetBrains s.r.o., „PyCharm Python IDE for Professional Developers,“ [Online]. Available: https://www.jetbrains.com/pycharm/. [Accessed 12 May 2017].

[64] S. van der Walt, S. C. Colbert und G. Varoquaux, „The NumPy Array: A Structure for Efficient Numerical Computation,“ *Computing in Science & Engineering,* pp. 22-30, 2011.

[65] E. Jones, T. Oliphant, P. Peterson und others, „SciPy: Open source scientific tools for Python," [Online]. Available: https://www.scipy.org/. [Accessed 12 May 2017].

[66] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot und E. Duchesnay, „Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research,* pp. 2825-2830, October 2011.

[67] J. D. Hunter, „Matplotlib: A 2D Graphics Environment," *Computing in Science & Engineering,* pp. 90-95, 2007.

[68] G. Lemaitre, F. Nogueira und C. K. Aridas, „Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning," *Journal of Machine Learning Research,* Bd. 18, Nr. 17, pp. 1-5, 2017.

[69] Georg Brandl and the Sphinx team, „Sphinx: Python Documentation generator," [Online]. Available: http://www.sphinx-doc.org/en/stable/authors.html. [Accessed 13 May 2017].

[70] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher und P, „sklearn.tree.DecisionTreeClassifier," [Online]. Available: http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html. [Accessed 13 June 2017].

[71] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher und P, „sklearn.neighbors.KNeighborsClassifier," [Online]. Available: http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html. [Accessed 13 June 2017].

[72] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher und P, „sklearn.discriminant_analysis.QuadraticDiscriminantAnalysis," [Online]. Available: http://scikit-learn.org/stable/modules/generated/sklearn.discriminant_analysis. QuadraticDiscriminantAnalysis.html. [Accessed 13 June 2017].

[73] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher und P, „sklearn.neural_network.MLPClassifier," [Online]. Available: http://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html. [Accessed 13 June 2017].

[74] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher und P, „sklearn.svm.SVC," [Online]. Available: http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html. [Accessed 13 June 2017].

[75] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher und P, „sklearn.ensemble.AdaBoostClassifier," [Online]. Available: http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html. [Accessed 13 June 2017].

[76] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher und P, „sklearn.ensemble.BaggingClassifier," [Online]. Available: http://scikit-

learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html. [Accessed 13 June 2017].

[77] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher und P, „sklearn.ensemble.RandomForestClassifier,“ [Online]. Available: http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html. [Accessed 13 June 2017].

Appendix A: Source code

Following, some used functions from the created Python program will be stated. The functions are referenced and explained within the research project.

Basic evaluation function (scaling and classifier tests)

```
def basic_evaluation_binary(data, targets, classifiers,
  training_share=0.5, iterations=10, use_unscaled_data,
  use_interval_scaled_data, use_zscore_scaled_data,
  use_atan_scaled_data):

    # create empty dictionary for rates
    rates = {}
    for key in classifiers.keys():
        rates[key] = {}
        if use_unscaled_data:
            rates[key]['unscaled'] = np.zeros((2, 2))
        if use_interval_scaled_data:
            rates[key]['interval'] = np.zeros((2, 2))
        if use_zscore_scaled_data:
            rates[key]['zscore'] = np.zeros((2, 2))
        if use_atan_scaled_data:
            rates[key]['atan'] = np.zeros((2, 2))

    for x in range(0, iterations):
        # random sampling
        train_unscaled, test_unscaled, train_targets, test_targets =
           train_test_split(data, targets, test_size=1-training_share,
           stratify=targets)

        # scaling
        if use_interval_scaled_data:
            scaler = preprocessing.MinMaxScaler().fit(train_unscaled)
            train_interval = scaler.transform(train_unscaled)
            test_interval = scaler.transform(test_unscaled)
        if use_zscore_scaled_data:
            scaler = preprocessing.StandardScaler().fit(train_unscaled)
            train_zscore = scaler.transform(train_unscaled)
            test_zscore = scaler.transform(test_unscaled)
        if use_atan_scaled_data:
            if not use_zscore_scaled_data:
                scaler =
                   preprocessing.StandardScaler().fit(train_unscaled)
                train_zscore = scaler.transform(train_unscaled)
                test_zscore = scaler.transform(test_unscaled)
            train_atan = (2 * np.arctan(train_zscore)) / np.pi
            test_atan = (2 * np.arctan(test_zscore)) / np.pi

        # prediction and metric calculation
        for key, classifier in classifiers.items():
            if use_unscaled_data:
                rates[key]['unscaled'] +=
                   metrics.confusion_matrix(test_targets,
                   classifier.fit(train_unscaled,
                   train_targets).predict(test_unscaled))
            if use_interval_scaled_data:
                rates[key]['interval'] +=
                   metrics.confusion_matrix(test_targets,
                   classifier.fit(train_interval,
                    train_targets).predict(test_interval))
            if use_zscore_scaled_data:
                rates[key]['zscore'] +=
                    metrics.confusion_matrix(test_targets,
                    classifier.fit(train_zscore,
                    train_targets).predict(test_zscore))
            if use_atan_scaled_data:
                rates[key]['atan'] +=
                    metrics.confusion_matrix(test_targets,
                    classifier.fit(train_atan,
                    train_targets).predict(test_atan))


    # divide rates by iterations to get the mean
    for key in classifiers.keys():
        if use_unscaled_data:
            rates[key]['unscaled'] /= iterations
        if use_interval_scaled_data:
            rates[key]['interval'] /= iterations
        if use_zscore_scaled_data:
            rates[key]['zscore'] /= iterations
        if use_atan_scaled_data:
            rates[key]['atan'] /= iterations

    return rates
```

Listing 5: Basic evaluation function (scaling and classifier tests)

To execute this function successfully, the following parameters are necessary:

- **data [numpy matrix]:** Numpy dataset matrix (columns=features)
- **targets [numpy matrix]:** Numpy target matrix (each row = one target)
- **classifiers [dictionary]:** The name of the classifiers as key for each entry and the classifier itself as value; the evaluation is executed for each classifier
- **training_share [double value > 0 *and* < 1]:** The percentage of data which should be used to train the classifier
- **iterations [int value ≥ 1]:** The number of repetitions for a single evaluation process
- **use_unscaled_data [boolean]:** True if unscaled data should be used for the evaluation
- **use_interval_scaled_data [boolean]:** True if the data should be fitted to the interval [0,1] and used for the evaluation
- **use_zscore_scaled_data [boolean]:** True if the data should be z-score scaled (standardized) and used for the evaluation
- **use_atan_scaled_data [boolean]:** True if the data should be arctangent scaled and used for the evaluation

Custom scorer function for the GridSearchCV

```
def __custom_scorer_mixed_f1_gmean(targets, predictions):
    rates = metrics.confusion_matrix(targets, predictions)
    TN = rates[0][0]; FP = rates[0][1];
    FN = rates[1][0]; TP = rates[1][1];

    # calculate F1 performance metrics
    if (TP + FN == 0) or (TP + FP == 0):
        F1 = 0
    else:
        Recall = TP / (TP + FN)
        Precision = TP / (TP + FP)

        if Recall == 0 or Precision == 0:
            F1 = 0
        else:
            F1 = (2 * Precision * Recall) / (Precision + Recall)

    # calculate Gmean performance metrics
    if (TP + FN == 0) or (TN + FP == 0):
        Gmean = 0
    else:
        Recall = TP / (TP + FN)
        Specifity = TN / (TN + FP)
        Gmean = (Recall * Specifity) ** (1 / 2)

    return np.mean([F1, Gmean])
```

Listing 6: Custom scorer function for the GridSearchCV

Generation of grid search classifiers and parameters

```
def __generate_grid_search_classifiers(use_DT, use_kNN, use_QDA, use_MLP,
                                       use_SVM, use_DTBoost, use_DTBagg,
                                       use_RForest):
    grid_search_classifiers = {}

    if use_DT:
        grid_search_classifiers['DT'] = {}
        grid_search_classifiers['DT']['classifier'] =
            DecisionTreeClassifier()
        grid_search_classifiers['DT']['parameter'] = \
            {
                "criterion": ["gini", "entropy"],
                "min_samples_split": [2, 5, 10],
                "max_depth": [None, 5, 10, 20],
                "min_samples_leaf": [1, 2, 5],
                "max_leaf_nodes": [None, 20, 50]
            }
    if use_kNN:
        grid_search_classifiers['k-NN'] = {}
        grid_search_classifiers['k-NN']['classifier'] =
            KNeighborsClassifier()
        grid_search_classifiers['k-NN']['parameter'] = \
            {
                "n_neighbors": np.arange(1, 9),
                "weights": ['uniform', 'distance'],
            }
    if use_QDA:
        grid_search_classifiers['QDA'] = {}
        grid_search_classifiers['QDA']['classifier'] =
            QuadraticDiscriminantAnalysis()
        grid_search_classifiers['QDA']['parameter'] =
            {
                        "reg_param": np.concatenate([np.zeros(1), np.logspace(-
                                10, 0, 11)])
            }
    if use_MLP:
        grid_search_classifiers['MLP'] = {}
        grid_search_classifiers['MLP']['classifier'] = MLPClassifier()
        grid_search_classifiers['MLP']['parameter'] = \
            {
                "hidden_layer_sizes": np.arange(100, 1001, 100),
                "solver": ['lbfgs', 'sgd', 'adam'],
                "learning_rate": ['constant', 'invscaling', 'adaptive']
            }
    if use_SVM:
        grid_search_classifiers['SVM'] = {}
        grid_search_classifiers['SVM']['classifier'] = SVC()
        grid_search_classifiers['SVM']['parameter'] = \
            [
                {
                    'kernel': ['rbf'],
                    'C': [1, 10, 100, 1000],
                    'gamma': np.logspace(-9, 3, 13)
                },
                {
                    'kernel': ['linear'],
                    'C': [1, 10, 100, 1000]
                }
            ]
    if use_DTBoost:
        grid_search_classifiers['DTBoost'] = {}
        grid_search_classifiers['DTBoost']['classifier'] =
            AdaBoostClassifier(base_estimator= \
                        DecisionTreeClassifier(criterion="entropy"))
        grid_search_classifiers['DTBoost']['parameter'] =
            {
                        "n_estimators": np.arange(10, 101, 10)
            }
    if use_DTBagg:
        grid_search_classifiers['DTBagg'] = {}
        grid_search_classifiers['DTBagg']['classifier'] = BaggingClassifier(
            base_estimator=DecisionTreeClassifier(criterion="entropy"))
        grid_search_classifiers['DTBagg']['parameter'] =
            {
                        "n_estimators": np.arange(10, 101, 10)
            }
    if use_RForest:
        grid_search_classifiers['RandomForest'] = {}
        grid_search_classifiers['RandomForest']['classifier'] =
            RandomForestClassifier()
        grid_search_classifiers['RandomForest']['parameter'] = \
            {
                "n_estimators": np.arange(10, 101, 10),
                "max_features": ['sqrt', None, 0.01, 0.1, 0.2, 0.3, 0.4,
                    0.5, 0.6, 0.7, 0.8, 0.9],
                "criterion": ['gini', 'entropy'],
            }

    return grid_search_classifiers
```

Listing 7: Generation of grid search classifiers and parameters

K-fold cross validation for normal, cost-sensitive and sampling tests

```
def k_fold_cv_binary(data, targets, clfs, folds, iterations,
     use_thresholding=False, use_cost_models=False, sampling=None):

    rates = {}
    for key in clfs.keys():
        rates[key] = {}
        rates[key]['cm'] = np.zeros((2, 2))
        rates[key]['roc'] = {}
        rates[key]['roc']['FPR'] = np.linspace(0, 1, 2501)
        rates[key]['roc']['TPR'] = []

    for x in range(0, iterations):
         # k-fold sampling // iterate over each fold
        skf = StratifiedKFold(n_splits=folds, shuffle=True)
        for train_index, test_index in skf.split(data, targets):

            # set train and test data for this fold
            train_data, test_data = data[train_index], data[test_index]
            train_targets, test_targets = targets[train_index],
                 targets[test_index]

            if sampling != None:
                train_data, train_targets = sampling.fit_sample(train_data,
                 train_targets)

            # prediction and metric calculation
            for key, classifier in clfs.items():
                if not use_cost_models:
                    pred_clf = classifier.fit(train_data, train_targets)
                    if use_thresholding and not hasattr(pred_clf, 'voting'):
                        train_data = pred_clf.predict_proba(train_data)
                        test_data = pred_clf.predict_proba(test_data)
                        cost_mat = __generate_imb_cost_mat(train_targets)
                        pred_clf = ThresholdingOptimization().fit(train_data,
                         cost_mat, train_targets)
                else:
                    costs = __generate_imb_cost_mat(train_targets)
                    pred_clf = classifier.fit(train_data, train_targets, costs)

                # predict the test data
                prediction_targets = pred_clf.predict(test_data)

                # calculate metrics and store them in dict
                rates[key]['cm'] += metrics.confusion_matrix(test_targets,
                 prediction_targets)
                fpr, tpr, th_roc = metrics.roc_curve(test_targets,
                 prediction_targets)
                rates[key]['roc']['TPR'].append(
                 np.interp(rates[key]['roc']['FPR'], fpr, tpr))
                rates[key]['roc']['TPR'][-1][0] = 0.0

    # divide rates by iterations to get the average of the whole evaluation
    for key in clfs.keys():
        rates[key]['cm'] /= iterations
        rates[key]['roc']['TPR'] = np.mean(rates[key]['roc']['TPR'], axis=0)
        rates[key]['roc']['AUC'] = metrics.auc(rates[key]['roc']['FPR'],
         rates[key]['roc']['TPR'])

    return rates
```

Listing 8: K-fold cross validation for normal, cost-sensitive and sampling tests

To execute this function successfully, the following parameters are necessary:

- **data [numpy matrix]:** Numpy dataset matrix (columns=features)
- **targets [numpy matrix]:** Numpy target matrix (each row = one target)
- **clfs [dictionary]:** The name of the classifiers as key for each entry and the classifier itself as value; the evaluation is executed for each classifier
- **folds [int value $>$ 1]:** The number of fold for a single grid search
- **iterations [int value $\geq$ 1]:** The number of repetitions for the CV grid search
- **use_thresholding [boolean]:** True if thresholding should be performed to adapt the fit and predict behavior. Default: False
- **use_cost_models [boolean]:** True if cost classifiers are passed to adapt the fit and predict behavior. Default: False
- **sampling [Sampling Class]:** If a sampling method should be used, pass the sampling class to this parameter. Default: None

Confusion matrix performance evaluation function

```
def cm_performance_evaluation(rates):
    # calculate performance metrics
    TN = rates[0][0];
    FP = rates[0][1];
    FN = rates[1][0];
    TP = rates[1][1];

    Precision = TP / (TP + FP)
    Recall = TP / (TP + FN)
    Specifity = TN / (TN + FP)

    ACC = (TP + TN) / (TP + TN + FP + FN)
    FPR = 1 - Specifity
    FNR = 1 - Recall

    F1_Score = (2 * Precision * Recall) / (Precision + Recall)
    G_mean = (Recall * Specifity) ** (1 / 2)

    return ACC, FPR, FNR, F1_Score, G_mean
```

Listing 9: Confusion matrix performance evaluation function

ROC curve generation function

```
def roc_plot(dataset_name, classifier_name, rates_FPR, rates_TPR,
rates_AUC, save_plot=False, base_path="", scaling_name=""):
    plt.figure(figsize=(5, 5))
    plt.title('ROC: ' + dataset_name + ' - ' + classifier_name)
    plt.plot(rates_FPR, rates_TPR, 'b', label='AUC = %0.4f' % rates_AUC)
    plt.legend(loc='lower right')
    plt.plot([0, 1], [0, 1], 'r--')
    plt.xlim([-0.1, 1.1])
    plt.ylim([-0.1, 1.1])
    plt.ylabel('True Positive Rate')
    plt.xlabel('False Positive Rate')
    plt.grid(True)

    if save_plot:
        …
    else:
        plt.show()

    plt.close()
```

Listing 10: ROC curve generation function

Hierarchical smart grid IDS simulation process

```
def simulate(data, targets, training_size, iterations, sampling):

    # preprocess data (atan scaling)
    data = preprocessing.StandardScaler().fit_transform(data)
    data = (2 * np.arctan(data)) / np.pi

    # evaluation values
    rates = {}
    rates['cm'] = np.zeros((2, 2))
    rates['roc'] = {}
    rates['roc']['FPR'] = np.linspace(0, 1, 2501)
    rates['roc']['TPR'] = []
    HAN_total = 0
    HAN_accuracy = 0
    NAN_total = 0
    NAN_accuracy = 0
    WAN_total = 0
    WAN_accuracy = 0


    # execute simulation for x rounds
    for x in range(iterations):

        # split data
        data_train, data_test, targets_train, targets_test =
         train_test_split(data, targets, train_size=training_size)

        # oversampling for training data
        if sampling:
            data_train, targets_train = ADASYN().fit_sample(data_train,
                  targets_train)
        # build and train classifier for sg architecture
        HAN_clf, NAN_clf, WAN_clf =
         build_and_train_classifier_2HAN_4NAN_voteWAN(data_train, targets_train)

        # run model simulation for sg architecture
        prediction, HAN_total_round, HAN_accuracy_round, NAN_total_round,
         NAN_accuracy_round, WAN_total_round, WAN_accuracy_round = \
            model_evaluation_2HAN_4NAN_voteWAN(data_test, targets_test, HAN_clf,
                 NAN_clf, WAN_clf)

        # add performance values
        …

    # divide performance values by iterations to get the average performance
    …

    # calculate performance measures
    ACC, FPR, FNR, F1_Score, G_mean = cm_performance_evaluation(rates['cm'])
```

Listing 11: Hierarchical smart grid IDS simulation process

To execute this function successfully, the following parameters are necessary:

- **data [numpy matrix]:** Numpy dataset matrix (columns=features)
- **targets [numpy matrix]:** Numpy target matrix (each row = one target)
- **training_size [double value > 0 *and* < 1]:** The percentage of data which should be used to train the classifiers
- **iterations [int value ≥ 1]:** The number of repetitions for a single evaluation process
- **sampling [boolean]:** True, if ADASYN over-sampling should be used

Appendix B: ADFA-LD results

ADFA-LD results

ADFA-LD		FPR	FNR	AUC	ACC	F1	G
Single classifiers	k-NN	0.0186	0.1251	0.9281	0.9680	0.8728	0.9266
	MLP	0.0165	0.1334	0.9250	0.9689	0.8746	0.9232
	QDA	0.1661	0.0319	0.9010	0.8507	0.6191	0.8985
	SVM	0.0133	0.1443	0.9212	0.9703	0.8783	0.9189
Ensemble classifiers	DTBoost	0.0135	0.1487	0.9189	0.9695	0.8751	0.9164
	DTBagg	0.0111	0.1444	0.9223	0.9722	0.8852	0.9199
	RForest	0.0097	0.1436	0.9233	0.9735	0.8901	0.9209
	PlurVt	0.0107	0.1176	0.9358	0.9759	0.9017	0.9343
	WeighVt	0.0099	0.1190	0.9355	0.9764	0.9034	0.9339

Table 19: ADFA-LD results

ADFA-LD results (Under-sampling)

ADFA-LD		FPR	FNR	AUC	ACC	F1	G
Random-UnderSampler (RUS)	k-NN	0.1091	0.0411	0.9249	0.8994	0.7050	0.9243
	MLP	0.0802	0.0615	0.9291	0.9222	0.7514	0.9291
	QDA	0.1548	0.0284	0.9084	0.8611	0.6368	0.9062
	SVM	0.0676	0.0607	0.9358	0.9333	0.7792	0.9358
	DTBoost	0.0961	0.0702	0.9168	0.9071	0.7150	0.9167
	DTBagg	0.0896	0.0486	0.9309	0.9155	0.7385	0.9307
	RForest	0.0830	0.0456	0.9357	0.9217	0.7535	0.9355
	PlurVt	0.0821	0.0407	0.9386	0.9231	0.7577	0.9384
	WeighVt	0.0795	0.0405	0.9400	0.9254	0.7633	0.9398
Condensed-NearestNeighbour (CNN)	k-NN	0.0746	0.0922	0.9166	0.9232	0.7477	0.9165
	MLP	0.0680	0.1122	0.9099	0.9265	0.7517	0.9096
	QDA	0.1651	0.0413	0.8968	0.8504	0.6164	0.8946
	SVM	0.0314	0.1068	0.9309	0.9591	0.8456	0.9301
	DTBoost	0.0957	0.1148	0.8947	0.9019	0.6934	0.8947
	DTBagg	0.0621	0.1116	0.9131	0.9317	0.7653	0.9128
	RForest	0.0531	0.1063	0.9203	0.9402	0.7894	0.9199
	PlurVt	0.0332	0.0886	0.9391	0.9599	0.8506	0.9387
	WeighVt	0.0345	0.0812	0.9422	0.9596	0.8509	0.9419
Edited-NearestNeighbour (ENN)	k-NN	0.0476	0.0796	0.9364	0.9484	0.8172	0.9363
	MLP	0.0414	0.0831	0.9378	0.9534	0.8315	0.9375
	QDA	0.1635	0.0294	0.9036	0.8533	0.6239	0.9011
	SVM	0.0373	0.0947	0.9340	0.9555	0.8361	0.9336
	DTBoost	0.0425	0.1192	0.9192	0.9479	0.8091	0.9184
	DTBagg	0.0291	0.1027	0.9341	0.9617	0.8544	0.9334
	RForest	0.0265	0.1011	0.9362	0.9642	0.8628	0.9355
	PlurVt	0.0349	0.0757	0.9447	0.9600	0.8528	0.9445
	WeighVt	0.0354	0.0745	0.9451	0.9597	0.8520	0.9449

ADFA-LD		FPR	FNR	AUC	ACC	F1	G
RepeatedEdited-NearestNeighbour (RENN)	k-NN	0.0663	0.0603	0.9367	0.9345	0.7824	0.9367
	MLP	0.0583	0.0660	0.9379	0.9407	0.7980	0.9379
	QDA	0.1633	0.0288	0.9040	0.8536	0.6245	0.9015
	SVM	0.0542	0.0777	0.9341	0.9428	0.8018	0.9340
	DTBoost	0.0546	0.1018	0.9218	0.9394	0.7881	0.9215
	DTBagg	0.0435	0.0883	0.9341	0.9509	0.8230	0.9338
	RForest	0.0386	0.0839	0.9387	0.9557	0.8384	0.9385
	PlurVt	0.0514	0.0613	0.9437	0.9474	0.8173	0.9437
	WeighVt	0.0523	0.0564	0.9456	0.9472	0.8174	0.9456
All-KNN	k-NN	0.0585	0.0695	0.9360	0.9401	0.7957	0.9360
	MLP	0.0514	0.0738	0.9374	0.9458	0.8108	0.9374
	QDA	0.1633	0.0294	0.9036	0.8535	0.6241	0.9011
	SVM	0.0464	0.0854	0.9341	0.9487	0.8172	0.9339
	DTBoost	0.0483	0.1079	0.9219	0.9442	0.8004	0.9214
	DTBagg	0.0369	0.0932	0.9349	0.9560	0.8378	0.9345
	RForest	0.0330	0.0914	0.9378	0.9596	0.8495	0.9373
	PlurVt	0.0433	0.0698	0.9434	0.9534	0.8335	0.9434
	WeighVt	0.0436	0.0655	0.9455	0.9537	0.8349	0.9454
InstanceHardness-Threshold (IHT)	k-NN	0.0910	0.0534	0.9278	0.9138	0.7335	0.9277
	MLP	0.0772	0.0607	0.9311	0.9249	0.7581	0.9310
	QDA	0.1658	0.0255	0.9043	0.8518	0.6224	0.9016
	SVM	0.0667	0.0716	0.9308	0.9326	0.7756	0.9308
	DTBoost	0.0886	0.0816	0.9149	0.9123	0.7241	0.9149
	DTBagg	0.0765	0.0639	0.9298	0.9251	0.7580	0.9298
	RForest	0.0701	0.0573	0.9363	0.9315	0.7753	0.9363
	PlurVt	0.0762	0.0483	0.9378	0.9273	0.7665	0.9377
	WeighVt	0.0750	0.0475	0.9387	0.9284	0.7694	0.9386
NearMiss (NM) version 1	k-NN	0.5422	0.1140	0.6719	0.5115	0.3126	0.6369
	MLP	0.5061	0.1134	0.6903	0.5431	0.3273	0.6617
	QDA	0.7159	0.4902	0.3969	0.3124	0.1567	0.3806
	SVM	0.7257	0.1040	0.5852	0.3523	0.2575	0.4958
	DTBoost	0.4547	0.1211	0.7121	0.5871	0.3480	0.6923
	DTBagg	0.2859	0.1231	0.7955	0.7345	0.4530	0.7914
	RForest	0.3051	0.1215	0.7867	0.7179	0.4384	0.7813
	PlurVt	0.5604	0.1182	0.6607	0.4950	0.3045	0.6226
	WeighVt	0.5508	0.1194	0.6649	0.5033	0.3077	0.6289
NearMiss (NM) version 2	k-NN	0.5197	0.0843	0.6980	0.5349	0.3305	0.6632
	MLP	0.4217	0.0682	0.7551	0.6226	0.3824	0.7341
	QDA	0.5732	0.1741	0.6263	0.4768	0.2836	0.5937
	SVM	0.7413	0.0645	0.5971	0.3436	0.2632	0.4920
	DTBoost	0.3886	0.0853	0.7631	0.6494	0.3955	0.7479
	DTBagg	0.2898	0.0871	0.8115	0.7356	0.4640	0.8052
	RForest	0.2995	0.0827	0.8089	0.7277	0.4579	0.8016
	PlurVt	0.4938	0.0779	0.7141	0.5583	0.3436	0.6832
	WeighVt	0.4962	0.0769	0.7135	0.5564	0.3428	0.6820

ADFA-LD		FPR	FNR	AUC	ACC	F1	G
TomekLinks	k-NN	0.0229	0.1117	0.9327	0.9660	0.8675	0.9316
	MLP	0.0209	0.1188	0.9301	0.9668	0.8695	0.9288
	QDA	0.1661	0.0322	0.9009	0.8507	0.6191	0.8984
	SVM	0.0169	0.1349	0.9241	0.9683	0.8725	0.9222
	DTBoost	0.0163	0.1359	0.9239	0.9687	0.8739	0.9220
	DTBagg	0.0134	0.1331	0.9267	0.9716	0.8844	0.9248
	RForest	0.0114	0.1311	0.9287	0.9736	0.8919	0.9268
	PlurVt	0.0141	0.1057	0.9401	0.9744	0.8975	0.9390
	WeighVt	0.0137	0.1074	0.9394	0.9746	0.8979	0.9383
OneSidedSelections (OSS)	k-NN	0.0233	0.1149	0.9309	0.9652	0.8644	0.9297
	MLP	0.0209	0.1194	0.9299	0.9668	0.8692	0.9286
	QDA	0.1670	0.0320	0.9005	0.8499	0.6179	0.8980
	SVM	0.0166	0.1304	0.9265	0.9691	0.8760	0.9247
	DTBoost	0.0162	0.1357	0.9240	0.9688	0.8741	0.9221
	DTBagg	0.0129	0.1345	0.9263	0.9719	0.8852	0.9243
	RForest	0.0112	0.1340	0.9274	0.9734	0.8909	0.9253
	PlurVt	0.0144	0.1067	0.9395	0.9741	0.8962	0.9383
	WeighVt	0.0138	0.1091	0.9385	0.9743	0.8967	0.9373
Neighbourhood-CleaningRule (NCR)	k-NN	0.0376	0.0899	0.9362	0.9558	0.8378	0.9359
	MLP	0.0339	0.0893	0.9384	0.9591	0.8482	0.9380
	QDA	0.1641	0.0298	0.9030	0.8527	0.6229	0.9005
	SVM	0.0289	0.1075	0.9318	0.9613	0.8525	0.9310
	DTBoost	0.0384	0.1233	0.9191	0.9509	0.8174	0.9181
	DTBagg	0.0232	0.1099	0.9335	0.9660	0.8677	0.9325
	RForest	0.0206	0.1078	0.9358	0.9685	0.8766	0.9348
	PlurVt	0.0273	0.0866	0.9431	0.9653	0.8683	0.9426
	WeighVt	0.0271	0.0850	0.9440	0.9656	0.8697	0.9435

Table 20: ADFA-LD results (Under-sampling)

ADFA-LD results (Over-sampling)

ADFA-LD		FPR	FNR	AUC	ACC	F1	G
Random-OverSampler (ROS)	k-NN	0.0358	0.0802	0.9420	0.9587	0.8480	0.9418
	MLP	0.0289	0.0973	0.9369	0.9625	0.8580	0.9363
	QDA	0.1685	0.0319	0.8998	0.8486	0.6158	0.8972
	SVM	0.0285	0.1130	0.9292	0.9609	0.8504	0.9283
	DTBoost	0.0147	0.1475	0.9189	0.9687	0.8721	0.9165
	DTBagg	0.0191	0.1197	0.9306	0.9683	0.8744	0.9292
	RForest	0.0146	0.1129	0.9362	0.9730	0.8919	0.9350
	PlurVt	0.0194	0.0861	0.9472	0.9722	0.8920	0.9467
	WeighVt	0.0175	0.0867	0.9479	0.9738	0.8975	0.9473

ADFA-LD		FPR	FNR	AUC	ACC	F1	G
SMOTE	k-NN	0.0394	0.0732	0.9437	0.9564	0.8420	0.9436
	MLP	0.0268	0.1077	0.9327	0.9630	0.8582	0.9319
	QDA	0.1465	0.0469	0.9033	0.8660	0.6406	0.9019
	SVM	0.0267	0.1188	0.9273	0.9617	0.8524	0.9261
	DTBoost	0.0204	0.1286	0.9255	0.9661	0.8656	0.9240
	DTBagg	0.0225	0.1109	0.9333	0.9664	0.8689	0.9322
	RForest	0.0184	0.1048	0.9384	0.9708	0.8849	0.9374
	PlurVt	0.0220	0.0838	0.9471	0.9702	0.8852	0.9466
	WeighVt	0.0204	0.0875	0.9460	0.9712	0.8882	0.9454
ADASYN	k-NN	0.0407	0.0735	0.9429	0.9552	0.8383	0.9428
	MLP	0.0347	0.0849	0.9402	0.9590	0.8484	0.9399
	QDA	0.2209	0.0392	0.8699	0.8019	0.5487	0.8652
	SVM	0.0339	0.0929	0.9366	0.9587	0.8462	0.9361
	DTBoost	0.0229	0.1176	0.9297	0.9652	0.8641	0.9285
	DTBagg	0.0274	0.0887	0.9420	0.9649	0.8670	0.9415
	RForest	0.0237	0.0928	0.9418	0.9677	0.8755	0.9411
	PlurVt	0.0283	0.0697	0.9510	0.9665	0.8746	0.9508
	WeighVt	0.0254	0.0745	0.9500	0.9684	0.8802	0.9497

Table 21: ADFA-LD results (Over-sampling)

ADFA-LD results (Hybrid-sampling)

ADFA-LD		FPR	FNR	AUC	ACC	F1	G
SMOTETomek	k-NN	0.0385	0.0721	0.9447	0.9573	0.8448	0.9446
	MLP	0.0263	0.1092	0.9322	0.9633	0.8588	0.9313
	QDA	0.1458	0.0464	0.9039	0.8667	0.6420	0.9025
	SVM	0.0257	0.1200	0.9272	0.9625	0.8547	0.9260
	DTBoost	0.0196	0.1284	0.9260	0.9668	0.8680	0.9244
	DTBagg	0.0212	0.1098	0.9345	0.9677	0.8734	0.9334
	RForest	0.0175	0.1056	0.9385	0.9715	0.8872	0.9375
	PlurVt	0.0212	0.0867	0.9460	0.9706	0.8862	0.9455
	WeighVt	0.0199	0.0874	0.9464	0.9717	0.8898	0.9458
SMOTEENN	k-NN	0.0340	0.0824	0.9418	0.9599	0.8515	0.9414
	MLP	0.0225	0.1194	0.9290	0.9653	0.8642	0.9278
	QDA	0.1398	0.0481	0.9061	0.8717	0.6504	0.9049
	SVM	0.0223	0.1301	0.9238	0.9642	0.8591	0.9222
	DTBoost	0.0215	0.1466	0.9160	0.9628	0.8519	0.9138
	DTBagg	0.0183	0.1210	0.9303	0.9688	0.8761	0.9289
	RForest	0.0146	0.1152	0.9351	0.9728	0.8907	0.9337
	PlurVt	0.0177	0.0970	0.9427	0.9724	0.8912	0.9418
	WeighVt	0.0167	0.0976	0.9428	0.9731	0.8939	0.9420

Table 22: ADFA-LD results (Hybrid-sampling)

ADFA-LD results (Cost-sensitive learning)

ADFA-LD		FPR	FNR	AUC	ACC	F1	G
Weighted	SVM	0.0219	0.2355	0.8713	0.9514	0.7976	0.8647
	DTBoost	0.0150	0.1543	0.9154	0.9676	0.8674	0.9127
	DTBagg	0.0118	0.1451	0.9215	0.9715	0.8826	0.9191
	RForest	0.0101	0.1442	0.9228	0.9731	0.8885	0.9204
	PlurVt	0.0083	0.1638	0.9140	0.9722	0.8830	0.9106
	WeighVt	0.0093	0.1451	0.9228	0.9737	0.8905	0.9203
Thresholding	k-NN	0.0415	0.0791	0.9397	0.9538	0.8332	0.9395
	MLP	0.0183	0.1830	0.8993	0.9610	0.8402	0.8955
	QDA	0.0195	0.1218	0.9293	0.9677	0.8719	0.9279
	SVM	0.0183	0.1831	0.8993	0.9610	0.8402	0.8955
	DTBoost	0.0183	0.1831	0.8993	0.9610	0.8402	0.8955
	DTBagg	0.0195	0.1221	0.9292	0.9677	0.8719	0.9278
	RForest	0.0195	0.1221	0.9292	0.9677	0.8719	0.9278
Cost-sensitive classifier	DT	0.0256	0.4955	0.7395	0.9155	0.5996	0.7011
	Bagging	0.0864	0.1014	0.9061	0.9117	0.7185	0.9061
	Pasting	0.0965	0.0879	0.9078	0.9046	0.7056	0.9078
	RForest	0.0879	0.0839	0.9141	0.9126	0.7244	0.9141
	RPatches	0.1215	0.0753	0.9016	0.8843	0.6671	0.9013

Table 23: ADFA-LD results (Cost-sensitive learning)

Appendix C: ICS-PSD-NvNA results

ICS-PSD-NvNA results

ICS-PSD-NvNA		FPR	FNR	AUC	ACC	F1	G
Single classifiers	k-NN	0.0428	0.0030	0.9771	0.9956	0.9977	0.9769
	MLP	0.0844	0.0038	0.9559	0.9934	0.9966	0.9551
	QDA	0.0870	0.0120	0.9505	0.9854	0.9924	0.9498
	SVM	0.0512	0.0073	0.9707	0.9912	0.9954	0.9705
Ensemble classifiers	DTBoost	0.0451	0.0006	0.9771	0.9979	0.9989	0.9769
	DTBagg	0.0694	0.0010	0.9648	0.9967	0.9983	0.9642
	RForest	0.0656	0.0004	0.9669	0.9973	0.9986	0.9664
	PlurVt	0.0387	0.0006	0.9803	0.9980	0.9990	0.9801
	WeighVt	0.0425	0.0004	0.9785	0.9982	0.9991	0.9783

Table 24: ICS-PSD-NvNA results

ICS-PSD-NvNA results (Under-sampling)

ICS-PSD-NvNA		FPR	FNR	AUC	ACC	F1	G
Random-UnderSampler (RUS)	k-NN	0.0032	0.1511	0.9227	0.8541	0.9182	0.9199
	MLP	0.0266	0.0584	0.9574	0.9427	0.9694	0.9573
	QDA	0.0691	0.0102	0.9603	0.9877	0.9936	0.9599
	SVM	0.0110	0.0604	0.9642	0.9413	0.9687	0.9640
	DTBoost	0.0069	0.0479	0.9725	0.9536	0.9754	0.9724
	DTBagg	0.0078	0.0587	0.9666	0.9430	0.9696	0.9664
	RForest	0.0040	0.0586	0.9685	0.9433	0.9697	0.9683
	PlurVt	0.0052	0.0479	0.9733	0.9536	0.9754	0.9732
	WeighVt	0.0095	0.0370	0.9766	0.9640	0.9810	0.9766
Condensed-NearestNeighbour (CNN)	k-NN	0.0069	0.1433	0.9247	0.8614	0.9227	0.9223
	MLP	0.0520	0.0358	0.9560	0.9636	0.9808	0.9560
	QDA	0.0633	0.0121	0.9623	0.9861	0.9928	0.9620
	SVM	0.0234	0.0414	0.9676	0.9592	0.9784	0.9676
	DTBoost	0.0142	0.0256	0.9800	0.9748	0.9868	0.9801
	DTBagg	0.0176	0.0222	0.9800	0.9780	0.9885	0.9801
	RForest	0.0188	0.0207	0.9801	0.9793	0.9892	0.9802
	PlurVt	0.0150	0.0152	0.9847	0.9848	0.9920	0.9849
	WeighVt	0.0162	0.0120	0.9857	0.9878	0.9936	0.9859
Edited-NearestNeighbour (ENN)	k-NN	0.0431	0.0083	0.9743	0.9905	0.9950	0.9742
	MLP	0.0780	0.0053	0.9583	0.9921	0.9959	0.9576
	QDA	0.0824	0.0119	0.9529	0.9857	0.9925	0.9522
	SVM	0.0474	0.0088	0.9719	0.9898	0.9947	0.9717
	DTBoost	0.0355	0.0024	0.9810	0.9964	0.9982	0.9809
	DTBagg	0.0676	0.0038	0.9643	0.9940	0.9969	0.9638
	RForest	0.0584	0.0027	0.9694	0.9954	0.9976	0.9691
	PlurVt	0.0324	0.0030	0.9823	0.9960	0.9979	0.9822
	WeighVt	0.0353	0.0024	0.9812	0.9964	0.9982	0.9810

ICS-PSD-NvNA		FPR	FNR	AUC	ACC	F1	G
RepeatedEdited-NearestNeighbour (RENN)	k-NN	0.0399	0.0096	0.9752	0.9893	0.9944	0.9751
	MLP	0.0725	0.0064	0.9605	0.9913	0.9955	0.9600
	QDA	0.0772	0.0119	0.9554	0.9858	0.9926	0.9549
	SVM	0.0384	0.0094	0.9760	0.9896	0.9946	0.9760
	DTBoost	0.0347	0.0033	0.9809	0.9956	0.9977	0.9809
	DTBagg	0.0746	0.0044	0.9604	0.9931	0.9964	0.9599
	RForest	0.0584	0.0037	0.9688	0.9944	0.9971	0.9686
	PlurVt	0.0335	0.0033	0.9815	0.9957	0.9977	0.9815
	WeighVt	0.0350	0.0030	0.9810	0.9959	0.9979	0.9809
All-KNN	k-NN	0.0451	0.0080	0.9734	0.9907	0.9952	0.9733
	MLP	0.0717	0.0055	0.9613	0.9921	0.9959	0.9608
	QDA	0.0844	0.0119	0.9518	0.9856	0.9925	0.9512
	SVM	0.0431	0.0087	0.9741	0.9901	0.9949	0.9740
	DTBoost	0.0341	0.0022	0.9818	0.9967	0.9983	0.9817
	DTBagg	0.0688	0.0039	0.9636	0.9938	0.9968	0.9631
	RForest	0.0552	0.0031	0.9708	0.9951	0.9974	0.9705
	PlurVt	0.0373	0.0027	0.9800	0.9961	0.9980	0.9799
	WeighVt	0.0376	0.0025	0.9799	0.9963	0.9981	0.9798
InstanceHardness-Threshold (IHT)	k-NN	0.0263	0.0276	0.9730	0.9725	0.9855	0.9731
	MLP	0.0240	0.0370	0.9694	0.9634	0.9807	0.9695
	QDA	0.0777	0.0115	0.9554	0.9861	0.9928	0.9548
	SVM	0.0211	0.0241	0.9773	0.9760	0.9874	0.9774
	DTBoost	0.0136	0.0388	0.9737	0.9621	0.9800	0.9737
	DTBagg	0.0367	0.0446	0.9592	0.9556	0.9765	0.9593
	RForest	0.0249	0.0437	0.9656	0.9569	0.9772	0.9657
	PlurVt	0.0150	0.0329	0.9759	0.9677	0.9830	0.9760
	WeighVt	0.0223	0.0301	0.9737	0.9702	0.9843	0.9738
NearMiss (NM) version 1	k-NN	0.0168	0.5218	0.7306	0.4958	0.6467	0.6857
	MLP	0.0590	0.5040	0.7185	0.5115	0.6621	0.6832
	QDA	0.1757	0.6284	0.5979	0.3874	0.5393	0.5534
	SVM	0.0197	0.7634	0.6084	0.2625	0.3825	0.4816
	DTBoost	0.0127	0.3859	0.8006	0.6271	0.7607	0.7786
	DTBagg	0.0231	0.4258	0.7755	0.5883	0.7292	0.7490
	RForest	0.0182	0.3893	0.7961	0.6236	0.7580	0.7743
	PlurVt	0.0084	0.5085	0.7415	0.5089	0.6589	0.6981
	WeighVt	0.0104	0.5024	0.7435	0.5147	0.6644	0.7017
NearMiss (NM) version 2	k-NN	0.0000	0.8966	0.5517	0.1346	0.1874	0.3216
	MLP	0.0066	0.8808	0.5563	0.1497	0.2130	0.3441
	QDA	0.0095	0.7930	0.5987	0.2343	0.3429	0.4528
	SVM	0.0020	0.8948	0.5516	0.1363	0.1904	0.3240
	DTBoost	0.0084	0.6915	0.6500	0.3323	0.4714	0.5531
	DTBagg	0.0064	0.7002	0.6467	0.3240	0.4612	0.5458
	RForest	0.0023	0.8762	0.5607	0.1542	0.2203	0.3514
	PlurVt	0.0055	0.8846	0.5549	0.1460	0.2069	0.3388
	WeighVt	0.0058	0.8853	0.5544	0.1453	0.2058	0.3377

ICS-PSD-NvNA		FPR	FNR	AUC	ACC	F1	G
TomekLinks	k-NN	0.0405	0.0029	0.9783	0.9958	0.9978	0.9781
	MLP	0.0824	0.0037	0.9569	0.9936	0.9967	0.9562
	QDA	0.0792	0.0117	0.9545	0.9859	0.9927	0.9540
	SVM	0.0526	0.0076	0.9699	0.9909	0.9953	0.9697
	DTBoost	0.0431	0.0007	0.9780	0.9978	0.9989	0.9779
	DTBagg	0.0723	0.0009	0.9634	0.9966	0.9983	0.9628
	RForest	0.0682	0.0004	0.9656	0.9972	0.9986	0.9651
	PlurVt	0.0379	0.0004	0.9808	0.9983	0.9991	0.9807
	WeighVt	0.0402	0.0005	0.9796	0.9981	0.9990	0.9795
OneSidedSelections (OSS)	k-NN	0.0390	0.0044	0.9782	0.9944	0.9971	0.9781
	MLP	0.0867	0.0042	0.9546	0.9930	0.9964	0.9537
	QDA	0.0815	0.0115	0.9535	0.9861	0.9928	0.9529
	SVM	0.0517	0.0083	0.9699	0.9902	0.9949	0.9697
	DTBoost	0.0436	0.0010	0.9777	0.9975	0.9987	0.9774
	DTBagg	0.0789	0.0011	0.9600	0.9962	0.9980	0.9592
	RForest	0.0679	0.0005	0.9658	0.9971	0.9985	0.9652
	PlurVt	0.0364	0.0006	0.9815	0.9982	0.9991	0.9813
	WeighVt	0.0396	0.0006	0.9798	0.9980	0.9990	0.9797
Neighbourhood-CleaningRule (NCR)	k-NN	0.0410	0.0049	0.9770	0.9939	0.9968	0.9769
	MLP	0.0815	0.0044	0.9570	0.9929	0.9963	0.9563
	QDA	0.0812	0.0117	0.9536	0.9859	0.9926	0.9529
	SVM	0.0486	0.0077	0.9718	0.9908	0.9952	0.9716
	DTBoost	0.0355	0.0017	0.9813	0.9972	0.9985	0.9812
	DTBagg	0.0702	0.0020	0.9638	0.9956	0.9977	0.9633
	RForest	0.0595	0.0014	0.9694	0.9965	0.9982	0.9691
	PlurVt	0.0347	0.0016	0.9818	0.9973	0.9986	0.9817
	WeighVt	0.0425	0.0015	0.9780	0.9971	0.9985	0.9778

Table 25: ICS-PSD-NvNA results (Under-sampling)

ICS-PSD-NvNA results (Over-sampling)

ICS-PSD-NvNA		FPR	FNR	AUC	ACC	F1	G
Random-OverSampler (ROS)	k-NN	0.0428	0.0032	0.9770	0.9954	0.9976	0.9768
	MLP	0.0425	0.0063	0.9756	0.9924	0.9961	0.9754
	QDA	0.0818	0.0116	0.9534	0.9860	0.9927	0.9527
	SVM	0.0480	0.0072	0.9724	0.9914	0.9955	0.9722
	DTBoost	0.0494	0.0008	0.9749	0.9975	0.9987	0.9746
	DTBagg	0.0517	0.0023	0.9730	0.9960	0.9979	0.9727
	RForest	0.0503	0.0012	0.9743	0.9971	0.9985	0.9740
	PlurVt	0.0303	0.0012	0.9842	0.9977	0.9988	0.9841
	WeighVt	0.0324	0.0010	0.9833	0.9979	0.9989	0.9832

ICS-PSD-NvNA		FPR	FNR	AUC	ACC	F1	G
SMOTE	k-NN	0.0306	0.0045	0.9824	0.9946	0.9972	0.9824
	MLP	0.0566	0.0060	0.9686	0.9922	0.9959	0.9683
	QDA	0.1000	0.0102	0.9449	0.9867	0.9931	0.9438
	SVM	0.0526	0.0064	0.9704	0.9920	0.9958	0.9702
	DTBoost	0.0260	0.0015	0.9862	0.9977	0.9988	0.9862
	DTBagg	0.0367	0.0033	0.9800	0.9955	0.9977	0.9799
	RForest	0.0301	0.0021	0.9839	0.9970	0.9984	0.9838
	PlurVt	0.0202	0.0020	0.9888	0.9974	0.9986	0.9889
	WeighVt	0.0220	0.0012	0.9883	0.9981	0.9990	0.9884
ADASYN	k-NN	0.0355	0.0044	0.9799	0.9945	0.9971	0.9799
	MLP	0.0723	0.0063	0.9607	0.9914	0.9956	0.9602
	QDA	0.0780	0.0871	0.9174	0.9132	0.9531	0.9174
	SVM	0.0442	0.0083	0.9737	0.9905	0.9950	0.9736
	DTBoost	0.0228	0.0019	0.9876	0.9974	0.9986	0.9876
	DTBagg	0.0246	0.0029	0.9862	0.9964	0.9981	0.9862
	RForest	0.0231	0.0018	0.9874	0.9974	0.9987	0.9875
	PlurVt	0.0205	0.0020	0.9886	0.9974	0.9986	0.9887
	WeighVt	0.0159	0.0015	0.9912	0.9980	0.9990	0.9913

Table 26: ICS-PSD-NvNA results (Over-sampling)

ICS-PSD-NvNA results (Hybrid-sampling)

ICS-PSD-NvNA		FPR	FNR	AUC	ACC	F1	G
SMOTETomek	k-NN	0.0321	0.0041	0.9819	0.9950	0.9974	0.9818
	MLP	0.0540	0.0063	0.9698	0.9920	0.9959	0.9695
	QDA	0.1043	0.0100	0.9428	0.9867	0.9931	0.9417
	SVM	0.0540	0.0064	0.9698	0.9920	0.9958	0.9695
	DTBoost	0.0280	0.0014	0.9852	0.9977	0.9988	0.9852
	DTBagg	0.0341	0.0035	0.9812	0.9955	0.9977	0.9811
	RForest	0.0309	0.0022	0.9834	0.9968	0.9983	0.9833
	PlurVt	0.0182	0.0020	0.9898	0.9974	0.9987	0.9899
	WeighVt	0.0231	0.0013	0.9877	0.9979	0.9989	0.9877
SMOTEENN	k-NN	0.0280	0.0095	0.9812	0.9899	0.9947	0.9812
	MLP	0.0483	0.0078	0.9719	0.9908	0.9952	0.9717
	QDA	0.0991	0.0101	0.9454	0.9868	0.9931	0.9443
	SVM	0.0468	0.0078	0.9726	0.9908	0.9952	0.9725
	DTBoost	0.0251	0.0040	0.9854	0.9953	0.9976	0.9854
	DTBagg	0.0329	0.0063	0.9803	0.9927	0.9962	0.9803
	RForest	0.0266	0.0049	0.9842	0.9944	0.9971	0.9842
	PlurVt	0.0231	0.0044	0.9861	0.9949	0.9974	0.9862
	WeighVt	0.0214	0.0041	0.9872	0.9953	0.9976	0.9872

Table 27: ICS-PSD-NvNA results (Hybrid-sampling)

<u>ICS-PSD-NvNA results (Cost-sensitive learning)</u>

ICS-PSD-NvNA		FPR	FNR	AUC	ACC	F1	G
Weighted	SVM	0.0633	0.0057	0.9655	0.9922	0.9960	0.9650
	DTBoost	0.0436	0.0008	0.9778	0.9977	0.9988	0.9775
	DTBagg	0.0789	0.0016	0.9597	0.9957	0.9978	0.9590
	RForest	0.0780	0.0007	0.9607	0.9966	0.9983	0.9599
	PlurVt	0.0286	0.0009	0.9852	0.9982	0.9991	0.9852
	WeighVt	0.0393	0.0003	0.9802	0.9983	0.9991	0.9800
Thresholding	k-NN	0.0393	0.0029	0.9789	0.9958	0.9978	0.9787
	MLP	0.0208	0.4022	0.7885	0.6111	0.7480	0.7651
	QDA	0.0373	0.0728	0.9449	0.9284	0.9615	0.9448
	SVM	0.0208	0.4022	0.7885	0.6111	0.7480	0.7651
	DTBoost	0.0208	0.4022	0.7885	0.6111	0.7480	0.7651
	DTBagg	0.0393	0.0029	0.9789	0.9958	0.9978	0.9787
	RForest	0.0393	0.0029	0.9789	0.9958	0.9978	0.9787
Cost-sensitive classifier	DT	0.5477	0.0160	0.7181	0.9655	0.9822	0.6671
	Bagging	0.0740	0.0283	0.9488	0.9701	0.9843	0.9486
	Pasting	0.0546	0.0313	0.9570	0.9679	0.9831	0.9570
	RForest	0.0572	0.0282	0.9573	0.9708	0.9847	0.9572
	RPatches	0.0497	0.0310	0.9596	0.9684	0.9834	0.9596

Table 28: ICS-PSD-NvNA results (Cost-sensitive learning)

Appendix D: ICS-PSD-NNvA results

ICS-PSD-NNvA results

ICS-PSD-NNvA		FPR	FNR	AUC	ACC	F1	G
Single classifiers	k-NN	0.1370	0.0393	0.9119	0.9391	0.9609	0.9106
	MLP	0.1585	0.0429	0.8993	0.9315	0.9560	0.8974
	QDA	0.1779	0.3946	0.7138	0.6534	0.7312	0.7055
	SVM	0.1935	0.0195	0.8935	0.9419	0.9634	0.8893
Ensemble classifiers	DTBoost	0.1915	0.0411	0.8837	0.9256	0.9525	0.8805
	DTBagg	0.1188	0.0152	0.9330	0.9618	0.9757	0.9315
	RForest	0.1173	0.0150	0.9338	0.9623	0.9760	0.9324
	PlurVt	0.1205	0.0186	0.9304	0.9588	0.9737	0.9290
	WeighVt	0.1297	0.0171	0.9266	0.9580	0.9733	0.9249

Table 29: ICS-PSD-NNvA results

ICS-PSD-NNvA results (Under-sampling)

ICS-PSD-NNvA		FPR	FNR	AUC	ACC	F1	G
Random-UnderSampler (RUS)	k-NN	0.0757	0.1528	0.8858	0.8643	0.9067	0.8849
	MLP	0.0981	0.1516	0.8751	0.8602	0.9043	0.8747
	QDA	0.0817	0.5472	0.6856	0.5559	0.6135	0.6448
	SVM	0.1224	0.1121	0.8827	0.8856	0.9236	0.8827
	DTBoost	0.1005	0.1518	0.8739	0.8596	0.9039	0.8735
	DTBagg	0.0485	0.1281	0.9117	0.8895	0.9248	0.9108
	RForest	0.0455	0.1134	0.9206	0.9016	0.9335	0.9199
	PlurVt	0.0527	0.1274	0.9099	0.8891	0.9245	0.9092
	WeighVt	0.0537	0.1216	0.9124	0.8934	0.9277	0.9117
Condensed-NearestNeighbour (CNN)	k-NN	0.1180	0.0554	0.9133	0.9307	0.9550	0.9128
	MLP	0.1429	0.1404	0.8583	0.8590	0.9047	0.8583
	QDA	0.0660	0.8600	0.5370	0.3159	0.2417	0.3617
	SVM	0.1758	0.0904	0.8669	0.8907	0.9284	0.8659
	DTBoost	0.1266	0.2162	0.8286	0.8036	0.8614	0.8274
	DTBagg	0.0959	0.1823	0.8609	0.8369	0.8864	0.8599
	RForest	0.0901	0.1527	0.8786	0.8611	0.9048	0.8780
	PlurVt	0.0895	0.1143	0.8981	0.8912	0.9269	0.8980
	WeighVt	0.0969	0.0853	0.9089	0.9121	0.9419	0.9089
Edited-NearestNeighbour (ENN)	k-NN	0.0793	0.1330	0.8939	0.8789	0.9177	0.8935
	MLP	0.1010	0.1125	0.8933	0.8900	0.9263	0.8932
	QDA	0.3795	0.2759	0.6723	0.7011	0.7905	0.6703
	SVM	0.1449	0.0784	0.8884	0.9069	0.9391	0.8877
	DTBoost	0.1280	0.0827	0.8946	0.9072	0.9390	0.8943
	DTBagg	0.0725	0.0720	0.9278	0.9279	0.9525	0.9278
	RForest	0.0662	0.0666	0.9336	0.9335	0.9562	0.9336
	PlurVt	0.0757	0.0851	0.9196	0.9169	0.9449	0.9196
	WeighVt	0.0701	0.0884	0.9207	0.9156	0.9439	0.9207

ICS-PSD-NNvA		FPR	FNR	AUC	ACC	F1	G
RepeatedEdited-NearestNeighbour (RENN)	k-NN	0.0603	0.1717	0.8840	0.8530	0.8977	0.8823
	MLP	0.0802	0.1522	0.8838	0.8637	0.9064	0.8830
	QDA	0.6244	0.1803	0.5976	0.7213	0.8208	0.5549
	SVM	0.1241	0.1020	0.8869	0.8931	0.9290	0.8869
	DTBoost	0.1087	0.1078	0.8918	0.8920	0.9279	0.8918
	DTBagg	0.0586	0.1128	0.9143	0.8992	0.9320	0.9139
	RForest	0.0506	0.1030	0.9232	0.9086	0.9386	0.9228
	PlurVt	0.0571	0.1194	0.9117	0.8944	0.9285	0.9112
	WeighVt	0.0497	0.1326	0.9089	0.8858	0.9220	0.9079
All-KNN	k-NN	0.0687	0.1496	0.8909	0.8683	0.9095	0.8899
	MLP	0.0920	0.1287	0.8896	0.8794	0.9184	0.8894
	QDA	0.5108	0.2282	0.6305	0.7092	0.8052	0.6145
	SVM	0.1362	0.0882	0.8878	0.9011	0.9349	0.8875
	DTBoost	0.1161	0.0936	0.8951	0.9014	0.9347	0.8951
	DTBagg	0.0634	0.0900	0.9233	0.9159	0.9439	0.9232
	RForest	0.0575	0.0830	0.9297	0.9227	0.9486	0.9297
	PlurVt	0.0683	0.0989	0.9164	0.9079	0.9384	0.9163
	WeighVt	0.0613	0.1076	0.9155	0.9026	0.9345	0.9152
InstanceHardness-Threshold (IHT)	k-NN	0.0312	0.4067	0.7810	0.6765	0.7406	0.7581
	MLP	0.0410	0.3792	0.7899	0.6957	0.7606	0.7716
	QDA	0.3808	0.3243	0.6475	0.6632	0.7575	0.6469
	SVM	0.0169	0.4558	0.7637	0.6414	0.7027	0.7315
	DTBoost	0.0342	0.3997	0.7831	0.6813	0.7457	0.7614
	DTBagg	0.0245	0.3935	0.7910	0.6883	0.7518	0.7692
	RForest	0.0175	0.4129	0.7848	0.6747	0.7375	0.7595
	PlurVt	0.0159	0.4066	0.7887	0.6799	0.7427	0.7641
	WeighVt	0.0174	0.4050	0.7888	0.6809	0.7438	0.7646
NearMiss (NM) version 1	k-NN	0.0300	0.4295	0.7702	0.6590	0.7226	0.7439
	MLP	0.0583	0.4966	0.7226	0.6005	0.6624	0.6885
	QDA	0.7850	0.1867	0.5142	0.6808	0.7987	0.4182
	SVM	0.0240	0.5012	0.7374	0.6045	0.6626	0.6977
	DTBoost	0.0419	0.4481	0.7550	0.6419	0.7059	0.7272
	DTBagg	0.0097	0.4081	0.7911	0.6802	0.7424	0.7656
	RForest	0.0066	0.4235	0.7849	0.6688	0.7305	0.7567
	PlurVt	0.0113	0.4131	0.7878	0.6759	0.7382	0.7618
	WeighVt	0.0125	0.4307	0.7784	0.6620	0.7239	0.7498
NearMiss (NM) version 2	k-NN	0.0324	0.6480	0.6598	0.4883	0.5171	0.5836
	MLP	0.0357	0.6542	0.6551	0.4828	0.5100	0.5775
	QDA	0.4715	0.4754	0.5265	0.5254	0.6325	0.5265
	SVM	0.0211	0.6708	0.6541	0.4731	0.4931	0.5677
	DTBoost	0.0277	0.6464	0.6629	0.4906	0.5194	0.5863
	DTBagg	0.0171	0.6513	0.6658	0.4892	0.5153	0.5855
	RForest	0.0103	0.6592	0.6653	0.4845	0.5073	0.5808
	PlurVt	0.0174	0.6597	0.6615	0.4826	0.5059	0.5783
	WeighVt	0.0190	0.6554	0.6628	0.4856	0.5105	0.5814

ICS-PSD-NNvA		FPR	FNR	AUC	ACC	F1	G
TomekLinks	k-NN	0.1280	0.0504	0.9108	0.9324	0.9563	0.9100
	MLP	0.1496	0.0493	0.9005	0.9285	0.9539	0.8991
	QDA	0.1996	0.3785	0.7110	0.6612	0.7407	0.7053
	SVM	0.1895	0.0238	0.8934	0.9395	0.9617	0.8895
	DTBoost	0.1866	0.0463	0.8836	0.9226	0.9505	0.8808
	DTBagg	0.1110	0.0203	0.9344	0.9596	0.9742	0.9333
	RForest	0.1098	0.0190	0.9356	0.9609	0.9750	0.9345
	PlurVt	0.1086	0.0256	0.9329	0.9560	0.9718	0.9320
	WeighVt	0.1172	0.0242	0.9293	0.9552	0.9713	0.9281
OneSidedSelections (OSS)	k-NN	0.1310	0.0514	0.9088	0.9309	0.9553	0.9079
	MLP	0.1499	0.0503	0.8999	0.9276	0.9533	0.8985
	QDA	0.1974	0.3807	0.7110	0.6599	0.7393	0.7050
	SVM	0.1906	0.0247	0.8923	0.9385	0.9611	0.8884
	DTBoost	0.1877	0.0471	0.8826	0.9217	0.9499	0.8798
	DTBagg	0.1115	0.0205	0.9340	0.9593	0.9740	0.9329
	RForest	0.1082	0.0193	0.9362	0.9610	0.9751	0.9352
	PlurVt	0.1100	0.0256	0.9322	0.9557	0.9716	0.9312
	WeighVt	0.1185	0.0237	0.9289	0.9553	0.9714	0.9277
Neighbourhood-CleaningRule (NCR)	k-NN	0.0917	0.1046	0.9018	0.8982	0.9320	0.9018
	MLP	0.1179	0.0897	0.8962	0.9040	0.9366	0.8961
	QDA	0.3039	0.3047	0.6957	0.6955	0.7805	0.6957
	SVM	0.1620	0.0566	0.8907	0.9201	0.9484	0.8891
	DTBoost	0.1521	0.0680	0.8899	0.9134	0.9437	0.8890
	DTBagg	0.0830	0.0494	0.9338	0.9432	0.9630	0.9336
	RForest	0.0785	0.0455	0.9380	0.9472	0.9657	0.9378
	PlurVt	0.0849	0.0624	0.9264	0.9327	0.9559	0.9263
	WeighVt	0.0856	0.0634	0.9255	0.9316	0.9552	0.9254

Table 30: ICS-PSD-NNvA results (Under-sampling)

ICS-PSD-NNvA results (Over-sampling)

ICS-PSD-NNvA		FPR	FNR	AUC	ACC	F1	G
Random-OverSampler (ROS)	k-NN	0.1395	0.0390	0.9107	0.9387	0.9607	0.9093
	MLP	0.1416	0.0536	0.9024	0.9269	0.9527	0.9013
	QDA	0.0735	0.5628	0.6819	0.5456	0.5997	0.6364
	SVM	0.1953	0.0184	0.8931	0.9424	0.9637	0.8888
	DTBoost	0.1524	0.0671	0.8902	0.9140	0.9441	0.8892
	DTBagg	0.0932	0.0237	0.9416	0.9609	0.9749	0.9409
	RForest	0.0882	0.0208	0.9455	0.9643	0.9771	0.9449
	PlurVt	0.0957	0.0239	0.9402	0.9602	0.9745	0.9395
	WeighVt	0.1089	0.0204	0.9354	0.9600	0.9744	0.9343

ICS-PSD-NNvA		FPR	FNR	AUC	ACC	F1	G
SMOTE	k-NN	0.1175	0.0478	0.9174	0.9368	0.9591	0.9167
	MLP	0.1481	0.0550	0.8984	0.9244	0.9511	0.8972
	QDA	0.0683	0.6117	0.6600	0.5087	0.5517	0.6015
	SVM	0.1849	0.0229	0.8961	0.9413	0.9628	0.8925
	DTBoost	0.1652	0.0703	0.8822	0.9087	0.9406	0.8810
	DTBagg	0.0816	0.0318	0.9433	0.9571	0.9724	0.9429
	RForest	0.0746	0.0309	0.9473	0.9594	0.9738	0.9470
	PlurVt	0.0865	0.0310	0.9413	0.9567	0.9721	0.9409
	WeighVt	0.0993	0.0266	0.9370	0.9573	0.9726	0.9363
ADASYN	k-NN	0.1260	0.0443	0.9148	0.9376	0.9598	0.9139
	MLP	0.1327	0.0574	0.9049	0.9259	0.9519	0.9042
	QDA	0.8764	0.0725	0.5255	0.7494	0.8521	0.3386
	SVM	0.1617	0.0246	0.9069	0.9451	0.9651	0.9043
	DTBoost	0.1820	0.0903	0.8639	0.8894	0.9276	0.8627
	DTBagg	0.0932	0.0574	0.9247	0.9347	0.9574	0.9245
	RForest	0.0477	0.0508	0.9508	0.9499	0.9672	0.9508
	PlurVt	0.1007	0.0318	0.9338	0.9530	0.9697	0.9331
	WeighVt	0.1022	0.0330	0.9324	0.9516	0.9689	0.9317

Table 31: ICS-PSD-NNvA results (Over-sampling)

ICS-PSD-NNvA results (Hybrid-sampling)

ICS-PSD-NNvA		FPR	FNR	AUC	ACC	F1	G
SMOTETomek	k-NN	0.1162	0.0511	0.9163	0.9345	0.9575	0.9158
	MLP	0.1440	0.0562	0.8999	0.9243	0.9510	0.8988
	QDA	0.0675	0.6118	0.6603	0.5088	0.5517	0.6017
	SVM	0.1824	0.0243	0.8966	0.9407	0.9624	0.8931
	DTBoost	0.1703	0.0701	0.8798	0.9077	0.9401	0.8784
	DTBagg	0.0785	0.0343	0.9436	0.9559	0.9715	0.9433
	RForest	0.0756	0.0321	0.9462	0.9583	0.9731	0.9459
	PlurVt	0.0852	0.0329	0.9409	0.9555	0.9713	0.9406
	WeighVt	0.0988	0.0286	0.9363	0.9559	0.9716	0.9357
SMOTEENN	k-NN	0.0645	0.1508	0.8924	0.8684	0.9095	0.8913
	MLP	0.0892	0.1292	0.8908	0.8797	0.9185	0.8906
	QDA	0.0672	0.5598	0.6865	0.5493	0.6033	0.6408
	SVM	0.1296	0.0892	0.8906	0.9019	0.9353	0.8904
	DTBoost	0.1046	0.1132	0.8911	0.8887	0.9254	0.8911
	DTBagg	0.0462	0.1105	0.9216	0.9037	0.9350	0.9211
	RForest	0.0380	0.1027	0.9296	0.9116	0.9405	0.9291
	PlurVt	0.0485	0.1208	0.9153	0.8952	0.9289	0.9146
	WeighVt	0.0525	0.1180	0.9148	0.8965	0.9299	0.9142

Table 32: ICS-PSD-NNvA results (Hybrid-sampling)

ICS-PSD-NNvA results (Cost-sensitive learning)

ICS-PSD-NNvA		FPR	FNR	AUC	ACC	F1	G
Weighted	SVM	0.1959	0.0184	0.8929	0.9423	0.9636	0.8885
	DTBoost	0.1057	0.1347	0.8798	0.8717	0.9131	0.8797
	DTBagg	0.1345	0.0135	0.9260	0.9597	0.9744	0.9241
	RForest	0.1294	0.0111	0.9297	0.9627	0.9763	0.9278
	PlurVt	0.0985	0.0197	0.9409	0.9629	0.9762	0.9401
	WeighVt	0.1425	0.0094	0.9240	0.9611	0.9754	0.9216
Thresholding	k-NN	0.0960	0.0852	0.9094	0.9124	0.9421	0.9094
	MLP	0.0000	1.0000	0.5000	0.2215	nan	0.0000
	QDA	0.0000	1.0000	0.5000	0.2215	nan	0.0000
	SVM	0.0000	1.0000	0.5000	0.2215	nan	0.0000
	DTBoost	0.0000	1.0000	0.5000	0.2215	nan	0.0000
	DTBagg	0.1759	0.0244	0.8999	0.9421	0.9633	0.8967
	RForest	0.1759	0.0244	0.8999	0.9421	0.9633	0.8967
Cost-sensitive classifier	DT	0.6166	0.0817	0.6509	0.7998	0.8772	0.5934
	Bagging	0.1453	0.2745	0.7901	0.7541	0.8212	0.7874
	Pasting	0.1453	0.2745	0.7901	0.7541	0.8212	0.7874
	RForest	0.1453	0.2745	0.7901	0.7541	0.8212	0.7874
	RPatches	0.1400	0.2490	0.8055	0.7751	0.8387	0.8036

Table 33: ICS-PSD-NNvA results (Cost-sensitive learning)