CSABA BRUNNER

INSTITUTE OF INFORMATION TECHNOLOGY

SUPERVISOR: ANDREA KŐ PH.D.
SZABINA FODOR PH.D.

CORVINUS UNIVERSITY OF BUDAPEST

DOCTORAL SCHOOL OF BUSINESS INFORMATICS

INTRUSION DETECTION BY MACHINE LEARNING

PH.D. DISSERTATION

CSABA BRUNNER

BUDAPEST

2020

Abstract

Since the early days of information technology, there have been many stakeholders who used the technological capabilities for their own benefit, be it legal operations, or illegal access to computational assets and sensitive information. Every year, businesses invest large amounts of effort into upgrading their IT infrastructure, yet, even today, they are unprepared to protect their most valuable assets: data and knowledge. This lack of protection was the main reason for the creation of this dissertation. During this study, intrusion detection, a field of information security, is evaluated through the use of several machine learning models performing signature and hybrid detection. This is a challenging field, mainly due to the high velocity and imbalanced nature of network traffic. To construct machine learning models capable of intrusion detection, the applied methodologies were the CRISP-DM process model designed to help data scientists with the planning, creation and integration of machine learning models into a business information infrastructure, and design science research interested in answering research questions with information technology artefacts. The two methodologies have a lot in common, which is further elaborated in the study. The goals of this dissertation were two-fold: first, to create an intrusion detector that could provide a high level of intrusion detection performance measured using accuracy and recall and second, to identify potential techniques that can increase intrusion detection performance. Out of the designed models, a hybrid autoencoder and stacking neural network model managed to achieve detection performance comparable to the best models that appeared in the related literature, with good detections on minority classes. To achieve this result, the techniques identified were synthetic sampling, advanced hyperparameter optimization, model ensembles and autoencoder networks. In addition, the dissertation set up a soft hierarchy among the different detection techniques in terms of performance and provides a brief outlook on potential future practical applications of network intrusion detection models as well.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AE	Autoencoder network
AI	Artificial intelligence
ANN	Artificial neural network
API	Application programming interface
AUC	Area under the ROC curve
BPSO	Binary-based particle swarm optimization
CART	Classification and regression trees
CDF	Cumulative distribution function
CRISP-DM	Cross-industry standard process for data mining
CV	Cross validation
CVAE	Conditional variational autoencoder
DARPA	Defense Advanced Research Projects Agency
DBN	Deep belief networks
DBSCAN	Density-based spatial clustering of applications with noise
DDoS	Distributed denial of service
DM	Data mining
DMZ	Demilitarized zone
DNN	Deep neural network
DoS	Denial of service
DT	Decision trees
ENN	Edited nearest neighbors
FN	False negative
FP	False positive
HIDS	Host intrusion detection system
ICT	Info-communication technologies
ID3	Interactive dichotomizer 3
IDS	Intrusion detection system
KDD	Knowledge discovery in databases
KL divergence	Kullback-Leibler divergence
KNN	K-nearest neighbor
LR	Learning rate
MARS	Multiple adaptive regression splines

ML	Machine learning
MLP	Multilayer perceptron
MSE	Mean squared error
NB	Naive Bayes
NIDS	Network intrusion detection system
NLP	Natural language processing
NN	Neural network
PCA	Principal component analysis
PF	Probability function
R2L	Remote to local
RBF	Radial basis function
RELU	Rectified linear unit
RF	Random forest
RIPPER	Repeated incremental pruning to produce error reduction
RNN	Recurrent neural network
ROC	Receiver operating characteristic
SAE	Sparse autoencoder
SEMMA	Sample, explore, modify, model, assess
SGD	Stochastic gradient descent
SMOTE	Synthetic minority oversampling technique
SPSO	Standard-based particle swarm optimization
SVM	Support vector machine
TF	TensorFlow
TN	True negative
TP	True positive
TPE	Tree-structured parzen estimator
U2R	User to root
USAF	United States Air Force
VPN	Virtual private network

1. INTRODUCTION

The need to protect information systems and resources from misuse had arisen as early as 1972 and 1980, when James P. Anderson outlined that the USAF had become increasingly aware of information security related issues (Anderson (1972) and (1980)). Since then, the reported number of system intrusions grew at an alarming rate, especially from the early 2000s, which, according to reports like Beek *et al.* (2019) only increased in severity. The most common cyber attacks were the following:

- DDoS in the early 2000s (Lau *et al.* (2000), Smith (2014)), causing significant revenue loss by shutting down services,
- Botnet infections in relation to DDoS (Smith (2014)), taking computational resources from legitimate clients and using those resources for illegal conduct,
- ransomwares, specialized malwares (Beek *et al.* (2019)), encrypting information and demanding ransom for decryption,
- and more recently, deepfake attacks (Damiani (2019), Statt (2019)), where deep learning models are used to impersonate stakeholders in key positions to gain access to sensitive information or to conduct fraud.

The presence of these attacks changes among economic sectors, the most targeted being financial services, healthcare and education. Several methods exist for countering these malicious activities at different layers of an information system, a concept often referred to as defense in depth. One example is machine learning. Intrusive activities have well-defined patterns, detecting them is simple enough for a specialized system supported by the same machine learning algorithms. Furthermore, in some cases, like deepfake attacks, machine learning might be the only effective method of detection.

Despite cybercrime becoming more and more common, machine learning techniques are still not widespread and utilized enough in IT security. This is my motivation for studying network intrusion detection systems (NIDS) from a data mining perspective. My main goal was to provide a novel intrusion detection solution applying machine learning methods. To fulfill this goal, I set up, parameterized, trained and tested several intrusion detection models, implementing artificial neural network architectures. I combined two

approaches in my research: the design science research methodology and the CRISP-DM process model.

Throughout the dissertation I created four models in total for intrusion detection, going from simple classification ensembles to more complex neural network stacking models and hybrid anomaly-signature detection solutions implemented with the help of autoencoder networks. To evaluate how well each can detect intrusive behavior, I used the KDD Cup 1999 and NSL-KDD benchmark datasets for modeling, and the accuracy and recall metrics for model evaluation. I proved that machine learning is a suitable approach for detecting intrusions. Based on certain per-class and aggregate measures, at least one of the proposed models (V3) can compete and outperform works in the related literature. More details on the proposed models are available in chapter 4.2, and the comparison with the related literature in chapter 5.6.

In the following chapters of my dissertation I will introduce the concept of intrusions, intrusion detection and intrusion detection systems (IDS), the machine learning models and techniques that I used, or could have used for intrusion detection and the wider research conducted in the field in chapter 2. In chapter 3, I detail my choice of methodology based on the design science research methodology and CRISP-DM process model, followed by the design, implementation and evaluation of the machine learning model-based detectors I created in chapters 4 and 5. Chapter 6 contains the conclusions I collected with a brief outlook on practical application and further research possibilities.

2. BACKGROUND

In this chapter, I introduce the core concepts of my dissertation: intrusions, and intrusion detection systems. I follow this up describing data mining, its key characteristics and the different machine learning algorithms it uses, both supervised and unsupervised. I found this introduction important, as machine learning has gained recognition in the last decades in detecting intrusion attempts. Moreover, I discuss artificial neural networks and autoencoder networks in a separate chapter to detail how important they were to the detectors I implemented. Further chapters provide an introduction to the overall intrusion detection research, identifying the key literature within the field like McHugh (2000), Stolfo *et al.* (2000); Tavallaee *et al.* (2009), Tsai *et al.* (2009), Ippoliti (2011), Buczak and Guven (2015), Dua and Du (2016) and Molina-Coronado *et al.* (2020).

2.1. WHAT IS INTRUSION DETECTION?

According to Bhuyan, Bhattacharyya and Kalita (2014, pp. 303, 305) "*Intrusion is a deliberate and unauthorized attempt to access information, manipulate information and render a system unreliable or unusable. Intrusion itself is a set of actions aimed to compromise the security of computer and network components in terms of confidentiality, integrity and availability*". Intrusion detection is a set of actions to detect such events, to raise alerts, and to provide information to prevent them. Bruneau (2001, p. 2) described it as a collection of "*unrelenting active attempts in discovering or detecting the presence of intrusive activities.*" These attempts refer to all processes aimed at discovering unauthorized use of network or computer resources. Dua and Du (2016, p. 10) defined intrusions and intrusion detection as *"any unauthorized attempt to access, manipulate, modify, or destroy information or to use a computer system remotely to spam, hack, or modify other computers. An IDS intelligently monitors activities that occur in a computing resource, e.g., network traffic and computer usage, to analyze the events and generate reactions"*. This is a more up to date description that accounts for botnet activities and includes both network and host intrusion detection. Molina-Coronado *et al.* (2020, p. 2) defined intrusion detection systems the following way "*Intrusion Detection Systems are deployed to uncover cyberattacks that may harm information systems.*" In further chapters of this dissertation, when I talk about intrusion detection systems, I will mean a system designed to detect attempts at unauthorized access to an information

system coming from a wider external network. The key assumption for such a system to function is that intrusive behavior is discernable from normal activity.

According to Scarfone and Mell, (2007), Dua and Du, (2016) and Molina-Coronado *et al.* (2020), the following types of IDS exist:

- **Network based (NIDS)**: monitoring traffic on network devices or segments with the aim of detecting malicious traffic aimed at devices within the protected network boundaries. Network intrusion detectors are usually deployed in DMZs, as part of an intelligent firewall, VPN servers, remote access servers and wireless network access points.
- **Host based (HIDS)**: monitoring the resource consumption on a single system for suspicious activity. This host can be a critical IT infrastructure element, typically an application or database server. Together with NIDS, these are the most researched and mature fields.
- **Wireless**: monitoring wireless network traffic for possible intrusions. The characteristics of wireless communication makes it a special category of network intrusion detection.
- **Network behavior analysis**: monitoring network traffic to identify unusual flows (which could be a result of a DDoS attack).

Many techniques have been developed to create intrusion detection systems, from manual oversight in the early days, though expert and rule-based solutions to data science and machine learning. Data science plays a key role in modern intrusion detection, as it is the only technique that can handle the sheer volume of network traffic effectively. From a data scientific point of view, Scarfone and Mell (2007), Dua and Du (2016) and Molina-Coronado *et al.* (2020) distinguished the most common classes of intrusion detection:

- **Misuse / signature detection**: IDS that generates alarms when a known intrusion occurs. Known attacks can be detected reliably with low false positive rates, however new attacks cannot be detected. Misuse detectors describe known attacks as malicious patterns; therefore, they require data on the attacks first to be able to detect them.

- **Anomaly-based detection**: alarms are triggered when a traffic flow behaves in a significantly different way compared to normal traffic patterns. Subsequently, they can detect previously unknown attacks at the cost of a higher false positive rate. Ippoliti (2011) noted the key difference between anomaly and signature-based methods: anomaly detectors detect what their name suggests: anomalies in traffic and not intrusions: legitimate albeit unusual usage might raise alerts in an anomaly detection model, while a carefully constructed attack could remain undetected if it behaves like normal activity.

- **Hybrid detection:** to improve the detection performance of IDSs, some researchers proposed to combine anomaly and misuse detection into hybrid detectors. The underlying idea is to combine the benefits of the two, like the ability to detect known attacks with low false positive rates, while maintaining some ability of detecting new attacks when needed. Zhang and Zulkernine (2006), Zhang, Zulkernine and Haque (2008), Dua and Du (2016) and Molina-Coronado *et al.* (2020) identified four possible configuration for hybrid intrusion detection, also visible in Figure 1:

    - **Parallel detection**: used to correlate signature and anomaly detection results to provide a stronger detection (Figure 1.a). Network traffic is flagged as attack if either the anomaly or the signature detector identifies it as such.
    - **Signature-Anomaly sequence detection**: designed to improve detection ability on unknown attacks missed by the signature detector (Figure 1.b).
    - **Anomaly-Signature sequence detection**: designed to reduce false positive rates (Figure 1.c). The anomaly detector flags suspicious traffic, then the misuse detector confirms the flagged anomalies.
    - **Complex mixture detection**: any detection approach using anomaly and signature detectors simultaneously, that did not fit in the categories above. For example, the model demonstrated in Figure 1.d, where traffic is evaluated by an anomaly detector first. Normal traffic is further evaluated by a signature detector to identify attacks missed, while suspicious traffic is evaluated by a second anomaly detector to refine detections of the first model.

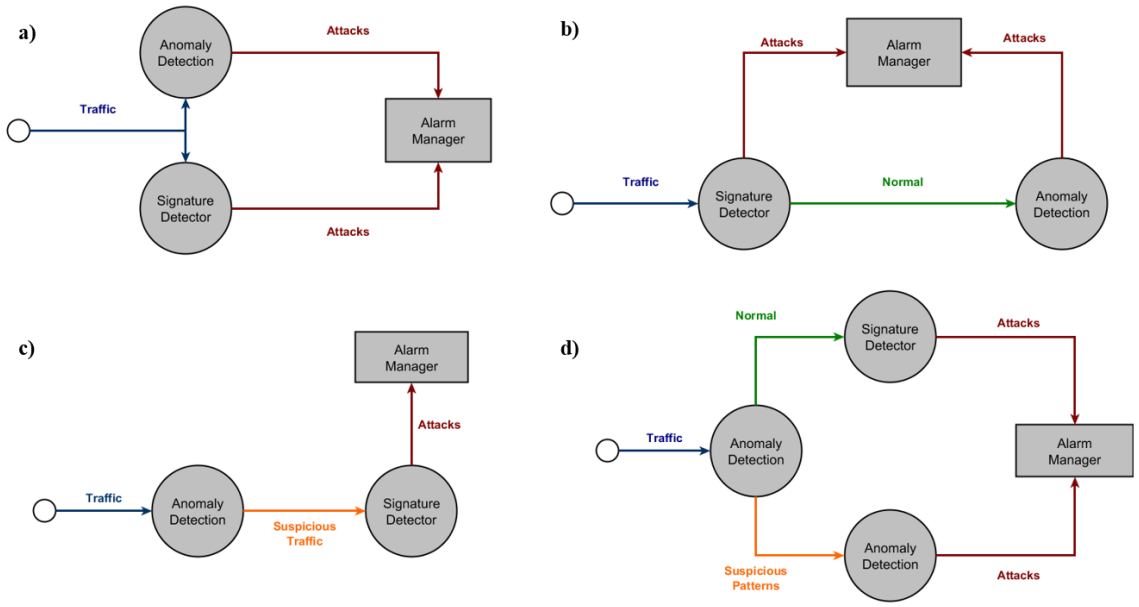

Figure 1: Types of hybrid intrusion detection. Source: Molina-Coronado *et al.*, (2020)

Even with data science and machine learning techniques, intrusion detection is a complex and challenging task for the reasons below:

- The most important challenge from a data science point of view, is the imbalanced representation of normal and intrusive activity. Normally, the volume of normal traffic outweighs that of attacks. At the sime time stakeholders are more interested in precisely detecting attacks. This implies that a learning system not only needs to be able to address an imbalance between normal and attack behavior, but it also has to be more effective at detecting attacks, even if it means a higher number of legitimate behavior gets flagged.

- A second challenge is the definition of performance. This could be the number of attacks detected, but could also mean the amount of time under which detection alerts are generated by the IDS. Both are correct depending on context. Metrics measuring detection peformance can do som from different perspectives, some less effective than others. Ultimately, the task determines the set of useful performance metrics, which is the accurate detection of attacks rather than normal traffic in case of intrusion detection.

- The amount of data available for intrusion detection is high, both in characteristics and in traffic records. The former requires a conscious effort at choosing the characteristics that best describe all or given attack patterns either through selection or information compression. The number of traffic records poses a

challenge when machine learning models are trained, but this can be mitigated by sampling the traffic. A further complication comes with the interpretation if traffic. They could be treated as individual packets, or as a communication flow. This distinction is important, as different attacks are effective at different levels.

After reviewing the approaches used for intrusion detection and the challenges it involves, I found network-based misuse / signature detection to be an interesting field to study, while also planning to include at least one hybrid intrusion detection. Therefore, three out of four of my proposed machine learning models performed signature detection only, with the fourth being a hybrid anomaly-signature detector. I used decision trees and artificial neural network (ANN) architectures set up in ensembles as machine learning models. Further algorithms, particularly used for sampling, were k-nearest neighbors (KNN) and support vector machines (SVM). I describe them in detail in the next chapter.

2.2. DATA MINING

Data mining has several definitions, Fayyad, Piatetsky-Shapiro and Smyth (1996) defined it as a part of a wider process called knowledge discovery in databases (KDD). KDD is determined as *"the overall process of discovering useful knowledge from data"* (Fayyad, Piatetsky-Shapiro and Smyth (1996, p. 40)) and data mining as *"a process using statistical, mathematical and artificial intelligence techniques to extract and identify useful information and subsequent knowledge from large sets of data"*. From the perspective of an IDS, the hidden knowledge is the unknown intent behind the source of the network traffic and the data is the inbound network traffic. The goal is to set apart traffic sent with malicious intent from the legitimate.

The terms data mining and machine learning, depending on interpretation, are often used as synonyms. In this dissertation I will use the following definition for machine learning: "*it is a field of study that gives computers the ability to learn without being explicitly programmed to*" (Samuel (1959), indirect quote). The definitions of KDD, data mining and machine learning make the relationship among them clearer (Figure 2). Data mining is an activity in the KDD process, producing patterns to discover interesting knowledge. Machine learning algorithms are frequently, though not exclusively, used in data mining to generate these patterns.

Figure 2: The relationships between KDD, data mining and machine learning. Based on Fayyad, Piatetsky-Shapiro and Smyth (1996)

The most common tasks and algorithms of data mining were organized by Sharda, Delen and Turban (2018). They distinguished prediction, association and segmentation tasks.

Prediction is referred to as the act of telling about the future. Prediction is further divided into classification and regression. Classification attempts to predict categorical, while regression attempts to predict numerical outcomes. This distinction is not as clear as it might sound at fist. Many algorithms, that were designed to perform one method, were extended to be applicable to the other as well. Typical example is the family of generalized regression models with linear and logistic regression performing regression and classification respectively. A counter example is the family of decision tree algorithms, initially created for classification, later extended to perform regression.

Association discovers interesting relationships between entities in large databases. For example, two products that are frequently purchased together. Two methods used for relationship detection are link and sequence analysis. Link analysis does not take the order of precedence between entities into account, while sequence analysis does.

In **segmentation** the goal is to split up and group structured data based on a similarity metric. Partitioning include clustering and outlier analysis techniques. The former creates homogenous groups where members in one group are more similar compared to members from other groups. Outlier analysis tries to find entities that are more dissimilar to others. By excluding these dissimilar entities, the effectiveness of following data mining algorithms can be improved.

A different classification of machine learning algorithms can be based on the learning process they use, according to Russel and Norwig (2010):

- **Supervised learning** the algorithm observes pairs of input-output observations and learns a function mapping from input to output. In supervised learning, input characteristics are called explanatory features and output is called target feature.

- **Unsupervised learning**, the algorithm learns patterns in the input, even though no expected output is supplied. These algorithms often perform self-organization as part of the learning process.
- **Semi-supervised learning**, the algorithm receives only a few examples with valid output, and the model must make decisions with data missing those labels.

Out of the data mining methods, classification, clustering and outlier analysis are the most common in intrusion detection. These methods can be organized into supervised and unsupervised types, the former representing techniques used for signature detection, the latter for anomaly detection. Just like with the categorization of Sharda, Delen and Turban (2018), some overlap between the categories exist, for example, SVMs, a supervised learning algorithm, can be altered for anomaly detection as a semi-supervised algorithm. Another example has been provided by Yao, Zhao and Maguire (2003), where an unsupervised association rule mining algorithm was extended with supervised learning models. The border between supervised and unsupervised learning approaches is not as clear as it might seem to be at first.

To systematically carry out data mining projects, a general process flow is required. Some of the most popular data mining process models are the cross industry standard process for data mining (CRISP-DM) designed by Chapman *et al.* (2000), the SEMMA model by Sharda, Delen and Turban (2018) and the knowledge discovery in databases (KDD) process I mentioned earlier from Fayyad, Piatetsky-Shapiro and Smyth (1996).

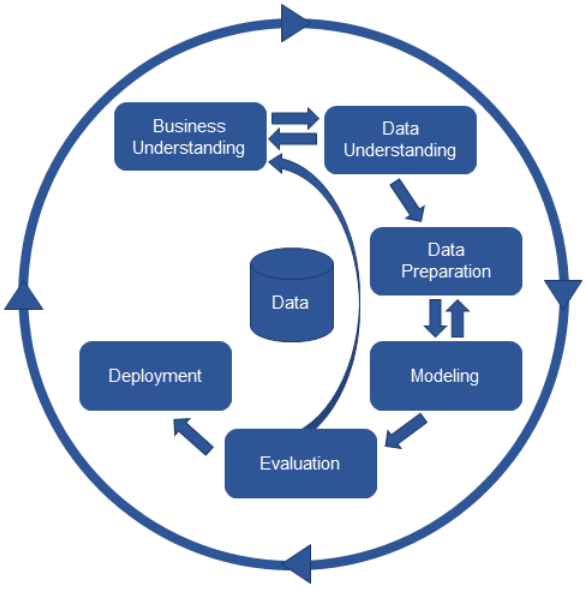

Figure 3: The CRISP-DM process model. Source: Chapman *et al.* (2000)

The CRISP-DM process shown in Figure 3 starts with a good understanding of the business and the associated need for data mining and ends with the deployment of a machine learning model that satisfies the specified business need. The process itself is iterative consisting of the following steps:

1. **Business understanding**: a key element of any data mining project is figuring out what the project is set to achieve. In this stage the tasks are to formulate business questions, and to develop a project plan with the necessary resources and budget assigned.
2. **Data understanding**: the next step in the process is to find and understand the relevant data that might come from many sources. To acquire this understanding, many simple statistical and graphical techniques are used.
3. **Data preparation**: the purpose of data preparation (or data preprocessing) is to take the data identified in the previous step and prepare it for the data mining algorithms, for example, by scaling the numerical features.
4. **Model building**: modeling techniques are selected and applied to a prepared data set to address needs and answer questions specified in the business understanding step.
5. **Testing and evaluation**: The models are assessed and evaluated for their generalization capability.
6. **Deployment**: model development and assessment are not the end of the data mining project. Knowledge acquired must be organized and presented in a way an end user can benefit from. Even with deployed models, annual re-evaluation might be necessary to maintain high performance, occasionally starting a new iteration of the CRISP-DM process.

Apart from CRISP-DM, the SEMMA methodology can be used, visible in Figure 4. It is an acronym standing for sample, explore, modify, model and assess. It starts with a representative data sample, applies exploratory statistical and visual analysis techniques, selects and transforms the most important predictive features, models them to predict outcomes and confirms the performance of a model. The main difference between CRISP-DM and SEMMA is that CRISP-DM takes a more comprehensive approach to the data mining process, including business and data understanding earlier, and model operation later in the process. SEMMA implicitly assumes that business understanding has been achieved before data mining and treats model operation as a separate process altogether.

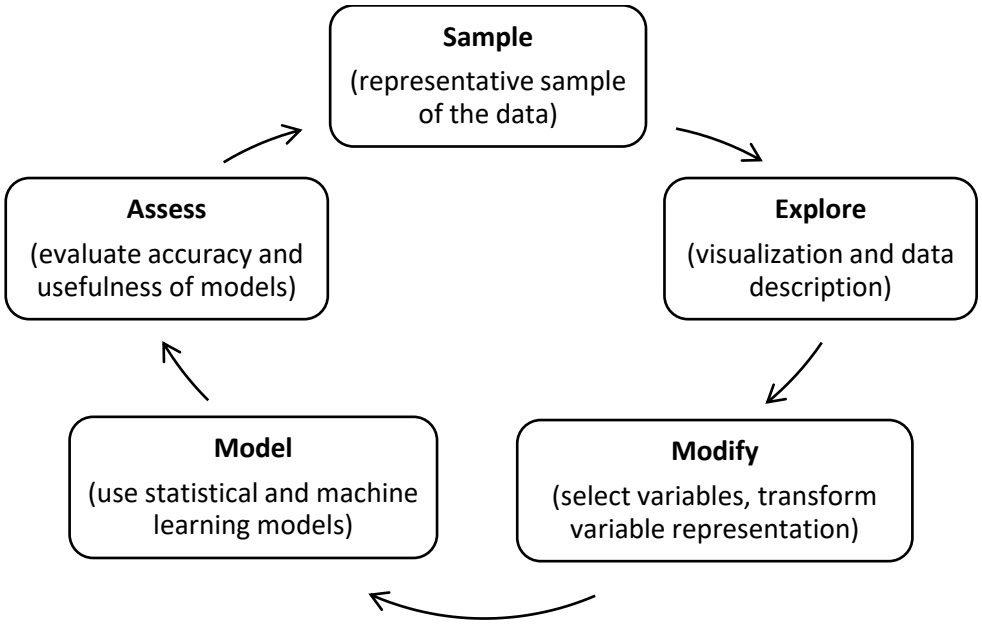

Figure 4: The SEMMA process model for data mining. Source: Sharda, Delen and Turban (2018)

The third process model is the KDD model (Figure 5). Compared to CRISP-DM, it is even more comprehensive, where data mining is only an important step, rather than the key focus. The complete list of activities of the KDD process:

1. **Data selection**: selection and query of data for analysis. Involves data integration, where data from multiple sources are joined together.
2. **Data cleaning and preprocessing**: remove noise and inconsistencies in data. Handle missing and outlier observations.
3. **Data transformation**: prepare data for analysis and data mining by performing aggregations and operations on data features.
4. **Data mining**: train models to detect hidden patterns.
5. **Interpretation and evaluation**: evaluate detected patterns to see whether they provide acceptable results and are interesting from a business perspective.

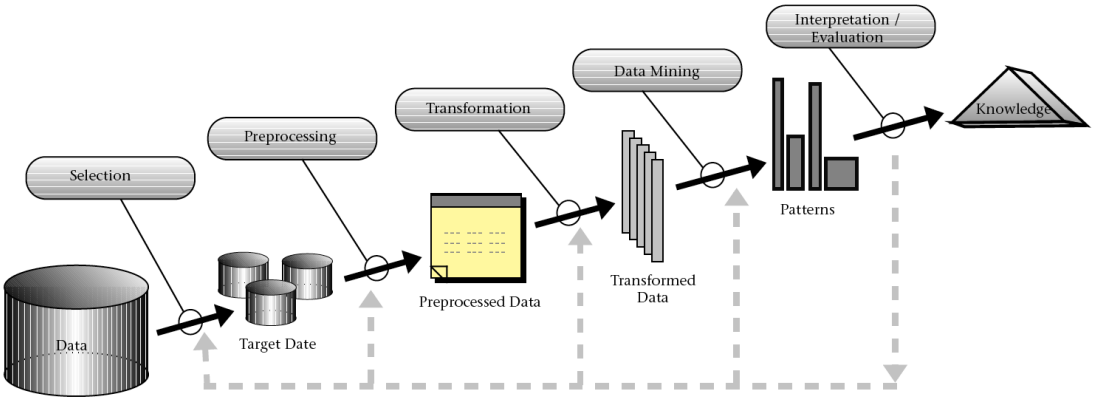

Figure 5: Knowledge discovery in databases. Source: Fayyad, Piatetsky-Shapiro and Smyth (1996)

Data mining has become a popular tool in addressing many complex business questions and opportunities. Sharda, Delen and Turban (2018) listed many economic fields where it can be useful, including customer relationship management, banking, retail and logistics, manufacturing and production, insurance, computer hardware and software, government and defense, travel industry, healthcare and medicine, entertainment industry, homeland security and law enforcement and sports. Intrusion detection can be applied in computer hardware and software, where it supports the detection of computer network security breaches, and in homeland security and law enforcement, where it plays a critical role in stopping malicious attacks on critical information infrastructures.

Thanks to data mining, organizational data, information and knowledge became the primary sources of competition on a global scale according to Nemati and Barko (2001). Organizations that successfully leverage the decision-enhancing environment realized by data mining can both obtain and maintain a lasting competitive advantage. This is the main strategic benefit of data mining.

The following chapters discuss data mining techniques sorted by type: starting with supervised and followed by unsupervised learning. Following that, I introduce the only machine learning algorithm I made an exception with, artificial neural network algorithms. I found them to be pivotal for my research, thus I dedicated an entire chapter to their introduction, focusing on fully connected feed-forward networks and autoencoder networks. In the last chapter, I will introduce techniques I used to improve model training and prediction performance. These include ensemble methods, combining results from

multiple base models, synthetic sampling methods, the metrics I used to evaluate predictions and hyperparameter optimization.

These chapters involve formula definitions when describing the different machine learning algorithms, many of which share common elements, for example explanatory and target features. I indicated the target feature for a hypothetical dataset with the letter y and the target feature values for a given observation as $y_i \in y, \forall i = 1 \dots n$. The target feature can also be described in the terms of set theory, where Y stands for the set of all possible values the target feature can take $Y_j \in Y, \forall j = 1 \dots k$. A similar notation can be created for the explanatory features as well. In this case, the complete set of explanatory variables is marked with X, with $X_s \in X, \forall s = 1 \dots m$ as the features of X. As a matrix, X can be traversed "row wise" as well, where the "rows" act as the observations of an entity or event. These are marked as $x_i \in X, \forall i = 1 \dots n$. Last, I defined the set of possible values for a given feature X_s as $x_u^{(s)} \in X_s, \forall u = 1 \dots v^s$ for $\forall s = 1 \dots m$.

Additional notation I used are the standard notation for probability ($P(\cdot)$) and conditional probability ($P(\cdot \mid \cdot)$), the notation for weight matrices (W) and the hat ($\hat{\cdot}$) notation for values estimated by the machine learning models. I will provide descriptions for every other new parameter or value that might appear in the introduced formulas in paragraphs preceding or following said introduction.

2.2.1. SUPERVISED LEARNING

A learning process is called supervised when the algorithm is provided with reference target information to compare learned patterns with. Based on the learned context, new observations can be predicted with higher probability of correctly identifying the real value than just by guessing randomly. The types of supervised learning are classification (where the reference is categorical) and regression (where the reference is numerical). Intrusion detection is concerned with predicting the class of incoming traffic; therefore, classification is a better fit. Training a classifier model can be time consuming, hence it is often performed off-line, while application is strictly on-line.

The greatest challenge with classification, especially for intrusion detection, is that the appropriate class labels must be acquired prior, which is often a tedious task itself. When determining the typical classification algorithms used for intrusion detection, I primarily

used the findings of Han, Kamber and Pei (2011), Bodon and Buza (2014) and Dua and Du (2016).

Decision Trees

Decision trees are sets of hierarchical if-then decisions generated by recursive partitioning algorithms according to a set purity measure. An example decision tree for a hypothetical credit scoring application can be seen in Figure 6. Represented in tree-like structures, an object can be classified starting from the root node and moving along the edges (~rules) towards the leaves. The final class of the object is provided by the label of the leaf node.

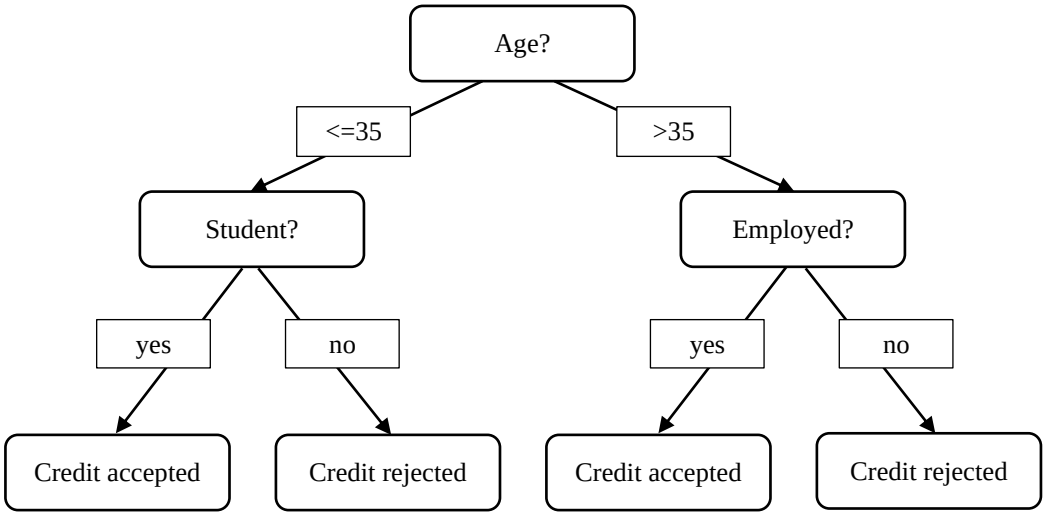

Figure 6: Decision tree. Based on: Han, Kamber and Pei (2011)

The construction process of decision trees according to Han, Kamber and Pei (2011) involve the following steps:

- Initially, the tree consists of a single root node.
- If all observations within a node belong to a single class, then the node becomes a leaf with the class value as label.
- Otherwise, an attribute is selected according to a purity measure. This purity measure is either the information gain ratio based on the Shannon-entropy, or the Gini index. This measure determines which attribute and value is selected for partitioning.
- The sample is then partitioned into subsamples.

- The above steps repeat recursively for each subsample until a stopping criterion is met:
    - If all observations on a node belong to a single class, then the associated class value will become the label of the leaf node.
    - If one feature can only be selected once and the list of available features for partitioning is empty. In this and the following cases, the label is determined by a simple majority vote.
    - The number of observations is less than a predefined threshold (prior minimum number of observations rule).
    - The number of observations in either node after a split would be smaller than a predefined threshold (posterior minimum number of observations rule)

The most common algorithms for creating decision trees are Interactive Dichotomizer 3 (ID3) by Quinlan (1986) and Classification and Regression Trees (CART) from Breiman *et al.* (1984). The main difference between the two is the measure for finding a critical attribute value for partitioning the tree. ID3-based algorithms use information gain ratio and the Shannon-entropy, CARTs prefer the Gini index. Consider target Y a probabilistic feature that can take k possible values with $P(Y_j)(j = 1, \dots, k)$ probability, then the Shannon-entropy of Y will be calculated as

$$H(Y) = H\big(P(Y_j), \dots, P(Y_k)\big) = -\sum_{j=1}^{k} P(Y_j) log_2 P(Y_j)$$

Entropy is a core concept in information theory; it refers to the uncertainty about the value of Y. If we observe probabilistic feature X_s, then the uncertainty of Y changes to

$$H(Y|X_s) = \sum_{u=1}^{v^s} P(X_s = x_u^{(s)}) H(Y|X_s = x_u^{(s)})$$

Meaning, if one observes the unique values of X_s, the uncertainty decreases by

$$I(Y, X_s) = H(Y) - H(Y|X_s)$$

This quantifies the information gained from feature X_s about Y. The entropy $H(Y|X_s)$ has a bias towards attributes with a large number of unique values according to Quinlan (1986). Information gain ratio eliminates this bias by normalizing information gain with the entropy of variable X_s:

$$gain\_ratio(X_s) = \frac{I(Y, X_s)}{H(X_s)}$$

To find the X_s feature which contributes the most to the value of Y, the information gain (or gain ratio) calculation is repeated for each $s = 1 \ldots m$, and we select X_s for which information gain (or gain ratio) is the highest.

The CART algorithm uses the Gini index instead of information gain, which is formulated as

$$Gini(Y) = 1 - \sum_{j=1}^{k} P(Y_j)^2$$

A key advantage of decision trees is the simplicity of their output for the end user. A disadvantage is their tendency to overfit: they learn specific details of the training data and generalize poorly on test data. This overfitting can be mitigated by pruning the decision trees, or in other words, replacing sub trees in a decision tree to improve predictions on the test set. The two most common methods for pruning are subtree replacement and subtree raising.

Support Vector Machines

Support vector machines (SVMs) are algorithms used for regression, classification and anomaly detection, designed by Cortes and Vapnik (1995). It constructs a m-1-dimensional separating hyperplane on m-dimensional data. A separation is considered good, when it has the highest distance (or margin) to the nearest data points, as the higher the margin, the lower the generalization error will be. A 2-dimensional example with optimal margin for SVM can be seen in Figure 7.

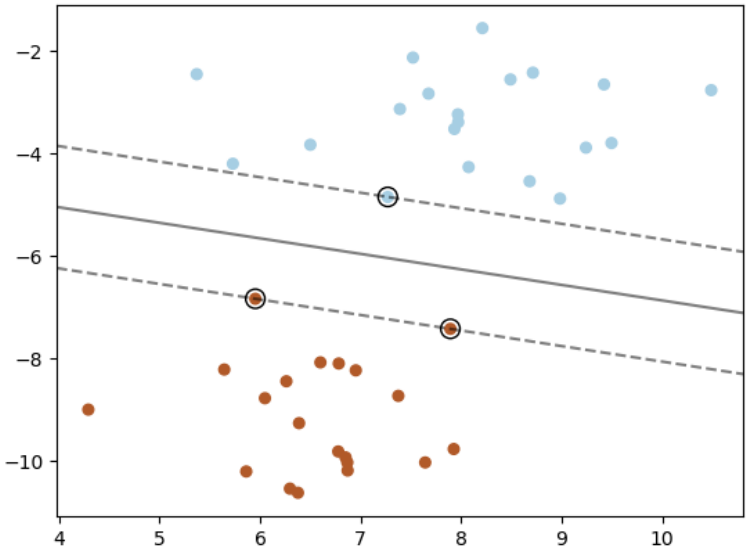

Figure 7: Optimal separating hyperplane with maximized margin created by SVM. Based on Cortes and Vapnik (1995)

Support vector classifiers take numerical observations $x_i \in X, i = 1, \ldots, n$ and a binary target vector $Y = \{1, -1\}^n$. They solve the following optimization problem:

$$\min_{W,b,\xi} \frac{\|W\|_2^2}{2} + C \sum_{i=1}^{n} \xi_i$$

$$\text{Subject to } y_i(W^T \Phi(x_i) + b) \geq 1 - \xi_i,$$

$$\xi_i \geq 0, \forall i = 1, \ldots, n$$

Its dual problem obtained from Lagrange multipliers is

$$\min_{\alpha} \frac{1}{2} \alpha^T Q \alpha - e^T \alpha$$

$$\text{Subject to } y^T \alpha = 0$$

$$0 \leq \alpha_i \leq C, \forall i = 1, \ldots, n$$

Where e is a vector of ones of length n, $C > 0$ is a tradeoff value for soft margin separation, Q is an $n \times n$ positive semidefinite matrix, for which $Q_{i_1 i_2} \equiv y_{i_1} y_{i_2} K(x_{i_1}, x_{i_2})$, where $K(x_{i_1}, x_{i_2}) = \Phi(\mathrm{x}_{i_1})^T \Phi(\mathrm{x}_{i_2})$ is the kernel function, most commonly linear, though other, more sophisticated kernel functions exist, such as gaussian, radial basis function and sigmoid. Φ stands for a function that transforms $\mathrm{x_i}$ observations into a feature space with higher dimension. This is often referred to as the kernel trick and it is used for linearly inseparable data in m dimensions. Furthermore,

ξ_i $(i = 1, \ldots, n)$ are the errors made by the SVM model on noisy data, used for soft margin classification, in other words, how much does the model permit classes on the "wrong" side of the hyperplane. With the dual solved, the decision function will be the following:

$$f(x) = sign(\sum_{i=1}^{n} y_i \alpha_i K(x_i, x) + \rho)$$

An advantage of SVMs is their simplicity: they find an optimal separating hyperplane. This hyperplane has been proven to have the highest margin, therefore SVM models tend to generalize well even when the number of explanatory features is high. They are applicable to linearly inseparable patterns in data, although then the model requires the use of the kernel trick and the C and ξ_i parameters. The only difficulty is finding the correct value for C. If too large, the model will generalize poorly, if too small, it will have a high error rate. The best strategy for finding C is to experiment, for example, with hyperparameter optimization and cross validation. A smaller issue with SVM is that it implicitly performs binary classification. This can be mitigated by using either one versus one or one vs rest classification strategies, meaning m SVM models are trained each comparing two classes, or a selected class and all the remaining classes.

K-Nearest Neighbor

The k-nearest neighbor (KNN) algorithm searches the variable space around x_i selected observation and selects the K nearest neighbors around it based on a distance metric (Han, Kamber and Pei (2011), Bodon and Buza (2014) and Dua and Du (2016)). Then, y_i will be estimated as the (weighted or non-weighted) arithmetic mean of the neighboring target values (in case of regression) or by the relative frequency of Y_j values in the neighborhood of y_i (classification). A demonstrative example of KNN with K = 1 is shown in Figure 8.

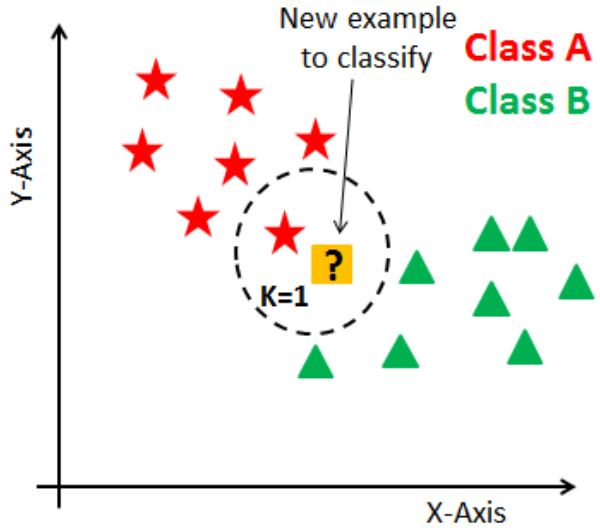

Figure 8: KNN classification with K=1. Source: Navlani (2018)

Some of the key challenges with k-nearest neighbor algorithm is finding an appropriate distance measure and a good K value for separation. The most common answer for the former is the Euclidean distance (assuming both x_{i_1} and x_{i_2} are observations with only numerical features):

$$Dist(x_{i_i}, x_{i_2}) = \left\| x_{i_1} - x_{i_2} \right\|_2$$

Finding the right value for K is more complicated: a small K might provide a good distinction between classes or a more accurate regression, but it is more sensitive to noise in the data. The best approach for finding K is trying out multiple settings and choosing the one with the best overall results.

KNN is a lazy classifier, it trains models fast. Testing, however, is slower and has a higher memory consumption, as the algorithm needs the complete training data for predictions. These characteristics make KNN less applicable on data that either has too many observations or too many features, regardless of how well KNN performs on said data.

Bayesian Networks

Bayesian networks use factored joint probability distributions in a graphical model to decide about uncertain features (Han, Kamber and Pei (2011), Bodon and Buza (2014) and Dua and Du (2016)). Bayesian networks rely on the Bayes-theorem for classification. Considering observation x_i and features $X_1 \ldots X_m$, let us mark $x_{i_1} \ldots x_{i_m}$ as the observation values for each feature. Let Y_j mark the probabilistic event that x_i belongs to class j, where $j = 1 \ldots k$. According to the Bayes rule:

$$P(Y_j|x_i) = \frac{P(x_i, Y_j)}{P(x_i)} = \frac{P(x_i|Y_j)P(Y_j)}{P(x_i)}$$

Where $P(Y_j|x_i)$ is the posterior probability of Y_j (how the likelihood of event Y_j changed knowing information about observation x_i), $P(Y_j)$ is the prior probability of Y_j (the likelihood of Y_j not knowing anything about x_i). Similarly, $P(x_i|Y_j)$ denotes the posterior probability of x_i knowing about the value of Y_j. Bayesian networks assign Y_j to x_i where $P(Y_j|x_i)$ is the highest out of $j = 1 \ldots k$ classes.

As $P(x_i)$ is constant for every class and $P(Y_j)$ is either provided already or can be estimated from sample (with relative frequencies, for example), Bayesian networks only need to maximize $P(x_i|Y_j)$ in order to maximize $P(Y_j|x_i)$. The data needed to calculate every possible $P(x_i|Y_j)$ probability is often not available in practice; therefore, some versions of Bayesian networks make assumptions about the probabilities to simplify calculations. For example, Naïve Bayes networks assume the conditional independence of $X_1 \ldots X_m$. In this case, $P(x_i|Y_j)$ can be simplified as

$$P(x_i|Y_j) = \prod_{s=1}^{m} P(x_{i_s}|Y_j)$$

For each class value. $P(x_{i_s}|Y_j)$ probabilities can be estimated from the available data. If X_s is categorical, then $P(x_{i_s}|Y_j)$ can be estimated with relative frequencies. When X_s is numerical and the distribution of $P(X_s|Y_j)$ is known, then the probability in question can be determined by estimating the parameters of the distribution with statistical methods.

The most important advantages of naïve Bayes are robustness (the models remain stable even if the conditional independence assumption is violated) and theoretical importance (the results of many neural network and curve fitting algorithm equals the maximum likelihood hypothesis provided by the naïve Bayes algorithm). The disadvantages of naïve Bayes models are their tendency to lose accuracy when their assumptions (conditional independence and the equal importance of every attribute) are violated. However, when the naïve Bayes algorithm is combined with feature selection techniques, then its classifications can rival the performance of decision trees and neural networks.

2.2.2. UNSUPERVISED LEARNING

According to Russel and Norwig (2010, p. 694) *"In unsupervised learning the agent learns patterns in the input even though no explicit feedback is supplied"*. Unsupervised learning is more useful for anomaly detection, as it provides more stable performance compared to signature detection models, are less costly to train and work well on previously unknown patterns. However, many unsupervised techniques can only handle numerical inputs, and differentiating attacks from normal activities is still a challenging task. The two most common type of unsupervised learning are clustering and outlier analysis.

Clustering algorithms partition a collection of entities into segments whose members share a similar characteristic, while members between segments are less likely to share that characteristic (Sharda, Delen and Turban (2018)). Many clustering algorithms have been invented, using different heuristics for similarity, therefore they might create different clusters even on the same data. The most common types of clustering algorithms are, according to (Bodon and Buza (2014)):

- **Partitioning** methods divide data into $C_j, j = 1 \dots k$ disjoint groups (or clusters), each containing at least one observation.
- **Hierarchical** methods construct hierarchical data structures, commonly referred to as dendrograms.
- **Density-based** methods overcome the common inability of earlier clustering algorithms to create clusters other than elliptical in shape. For a density-based cluster to be valid, at least n^* observations need to be in a predetermined radius from any other observation in the same cluster. Apart from clustering, density-based methods can be used for outlier analysis as well, making them well suited for intrusion detection.

Han, Kamber and Pei (2011) and Bodon and Buza (2014) refers to outliers as data with unusual and distinctively different characteristics from a larger set of observations. Often, outliers are either results of errors in data recording or inherent to the studied phenomena. If the latter is the case, then outliers themselves are the interesting patterns to be found. They could, for example, indicate fraudulent activities in a banking environment, or intrusive behavior in computer networks.

The task of outlier analysis is finding n^* outlier values in a dataset with n observations ($n^* \ll n$). This can be broken down to two questions: how to determine which observations are inconsistent with a large enough part of the data, and what are the effective methods for detecting them. Outliers could be defined by more than one feature, which excludes most (but not all) statistical analysis techniques used for outlier detection. A common way of creating multidimensional outlier detectors is the modification of pre-existing classification and clustering methods.

In the following subchapters, I will describe the most common algorithms used for clustering and outlier analysis with the help of Han, Kamber and Pei (2011), Bodon and Buza (2014) and Dua and Du (2016).

K-means clustering

K-means is the oldest and most common algorithm for clustering. It takes n observations and partitions them into k disjoint clusters. Observations in the same cluster are more similar to each other than to observations in other clusters. This "closeness" is captured by a distance function, most commonly the Euclidean distance, measured from the arithmetic mean of all $x_i \in C_j, \forall j = 1 \ldots k$, often represented as c_j centroid of a cluster. The goal of k-means clustering is to minimize a predetermined criteria function. The steps performed by the algorithm are shown in Figure 9.

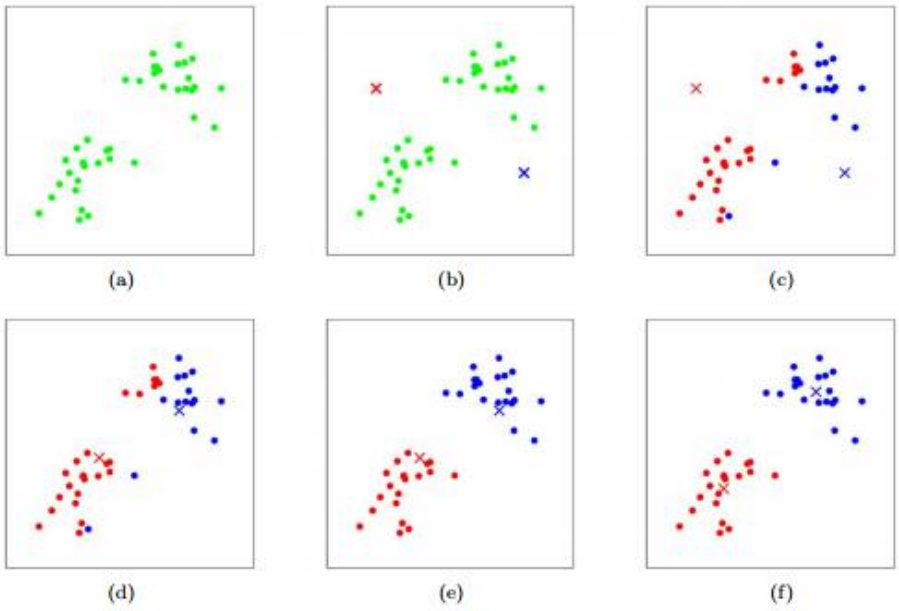

Figure 9: K-means clustering algorithm. Source: Piech (2012)

1. Start with X represented in a Euclidean space (Figure 9 (a)). Choose k points in space at random at first (Figure 9. (b)), mark them as the initial centroids ($c_j, j = 1 \dots k$).
2. Assign all $x_i \in X$ observations to the nearest c_j (Figure 9. (c)), based on a distance measure.
3. Re-calculate c_j centroids for each cluster (Figure 9. (d)).
4. Repeat steps 2 and 3 until a criteria function converges (Figure 9. (e)-(f)). This criteria function can be the squared error function:

$$SE = \sum_{i=1}^{n} \sum_{j=1}^{k} \left\| x_i^{(j)} - c_j \right\|_2^2$$

In the formula above $x_i^{(j)}$ is an observation belonging to C_j cluster with c_j centroid. This error function is the sum of distances for each observation and cluster.

The k-means algorithm works well when clusters form compact groups. It is a simple and fast algorithm that scales well with larger datasets. It is, however, not guaranteed to find global optima: it converges on a partitioning, even when a cluster setup could exist with lower squared error. Moreover, the algorithm only works with observations defined in a vector space, therefore categorical features must be excluded or encoded to numerical first.

Many variations of k-means were invented. These are different in their cluster initialization, in the distance functions from centroids or in what they treat as centroids. One of these variations is called k-medoid clustering, aiming to address two disadvantages with k-means: k-medoid results are less sensitive to outliers, and the algorithm is dependent on similarity metrics only, therefore observations are no longer required to be representable in Euclidean vector spaces. In k-medoid, a cluster centroid is not an arithmetic mean, but an actual observation ($c_j \in X \; \forall j = 1 \dots k$), called the medoid. As a result, the criteria function is altered; the squared distance is calculated from these medoids.

The k-means algorithm can be adapted for outlier analysis as well demonstrated by Dua and Du (2016). Without explicitly defining k, the clusters are also constrained by a threshold r. The difference from standard k-means algorithm comes when the distance between c_j and x_i is greater than threshold r. When that happens, a new cluster is

initialized with x_i as its initial centroid. The challenge of determining which clusters can be considered normal and which clusters as anomalous remains. The assumption is that normal data outnumbers anomalous data, therefore the clusters that contain more than a set α percentage of the training data are labelled as normal, the rest as anomaly.

DBSCAN

DBSCAN, or density-based spatial clustering of applications with noise is a clustering algorithm using two parameters (ε, a radius-like parameter and n^*, a threshold for the number of observations) for determining the density of a cluster developed. I has been developed by Ester *et al.* (1996) and it requires X explanatory features to be represented in an n-dimensional Euclidean space, just like k-means. Then, the neighborhood of x_i ($N_\varepsilon(x_i)$) is the set of observations that fall within an ε radius around x_i. Further terminology of DBSCAN is based on the following definitions:

- Observation x_i is directly density-reachable from x_j if $x_i \in N_\varepsilon(x_j)$ and $\left|N_\varepsilon(x_j)\right| \geq n^*$ (core point condition). Two core observations are density-reachable from each other, a border observation is directly density-reachable from a core observation, but a core observation is not directly density reachable from a border observation.
- Observation x_i is directly density-reachable from x_j if there exists a chain of observations $\{x_1^*, \ldots, x_n^*\}$ $x_1^* = x_i, x_n^* = x_j$ and x_{i+1}^* is directly density reachable from x_i^*.
- Observation x_i is density-connected to x_j if an observation x_ℓ exists, such that both x_i and x_j are density-reachable from x_ℓ.

Then, a cluster in DBSCAN can be defined as a set of density-connected observations. Observations, that were not assigned to any cluster will be considered as noise (or, in the case of outlier analysis, as the outliers themselves). Figure 10 demonstrates different types of observations determined by the DBSCAN algorithm.

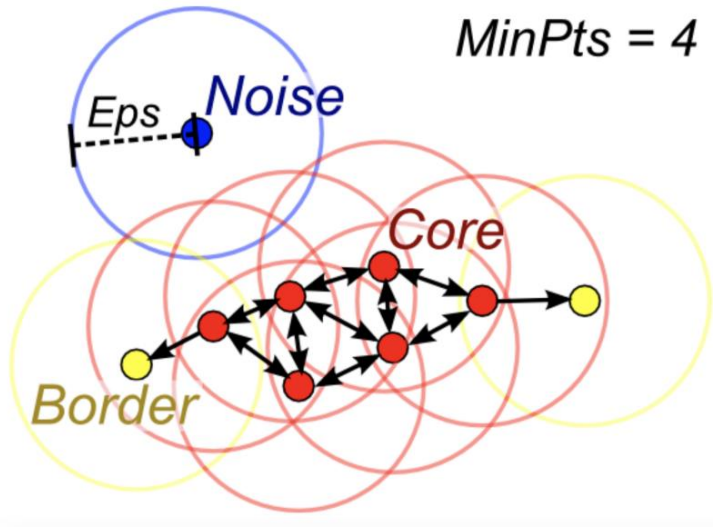

Figure 10: Point types in DBSCAN clustering. Source: Lutins (2017)

The DBSCAN algorithm can detect non-elliptical clusters, however, it is highly sensitive to the two input hyperparameters, ε and n^*. Finding these optimal parameters may not even be feasible if observation densities within a cluster are not uniform.

One Class SVM

The support vector machine algorithm is considered to be a supervised classification model, however, Schölkopf *et al.* (2000) proved that it can be modified to perform outlier analysis as well. One class SVM is an algorithm that learns a function with a returned value of +1 in a small region capturing a large portion of data points (called origin) and -1 everywhere else. To separate data from the origin, it solves the following quadratic problem:

$$\min_{w,\xi,\rho} \frac{\|W\|^2}{2} + \frac{1}{\nu n}\sum_{i=1}^{n}\xi_i - \rho$$

$$\text{Subject to } (W^T\Phi(x_i)) \geq \rho - \xi_i,$$

$$\xi_i \geq 0, \forall i = 1, \ldots, n$$

Note, that apart from a change to the C parameter, the problem definition remains largely the same. The $\nu \in (0, 1)$ parameter sets an upper limit on the fraction of outliers and a lower limit on the training examples used as support vectors simultaneously. The ρ parameter represents the margin separating outliers from the origin data, basically the distance of the separating hyperplane from the origin. The decision function can be determined by solving the Lagrange dual problem:

$$\min_{\alpha} \frac{1}{2} \alpha^T K(x_i, x_j) \alpha$$

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

Thus, the decision function will take the form of:

$$f(x) = sign(\sum_{i=1}^{n} \alpha_i K(x_i, x) - \rho)$$

One class SVM shares most advantages and disadvantages with the original SVM algorithm, the only major difference being that outlier classification is inherently a binary classification problem, therefore one class SVMs do not need adjustments for multiclass classification.

2.2.3. NEURAL NETWORKS

In this chapter I introduce the most common artificial neural network model, the feed-forward multilayer perceptron model. These models consist of parallel operating elements called neurons, each performing simple partitioning or fitting operations. Neural networks are powerful models, due to how, given enough neurons, they can approximate any arbitrary function. I based the first half of this chapter on the works of Rumelhart *et al.* (1988), Russel and Norwig (2010, pp. 727–737) and Kingma and Ba (2014). In the second half I introduce a specialized neural network architecture called autoencoder network. Autoencoder networks were designed to reconstruct their input data. A clever exploitation of this reconstruction on normal traffic makes autoencoders better suited for anomaly detection. My discussion on autoencoder networks is based on Ng and others, (2011), Kingma and Welling, (2013) and Sohn, Lee and Yan, (2015).

Artificial neural networks are designed to model the activity of the human brain, though this mathematical model cannot be claimed to be 100% accurate, as some operations in artificial neural networks were rather based on practical experiences. ANNs form networks of massively parallel distributed processing units called neurons. The schematic model for one neuron is presented in Figure 11. Each neuron is either connected with input observation values (x_i^S) or the outputs of other neurons ($a_q^{\ell-1}$, where ℓ refers to the layer the current neuron is part of and $q = 1 \ldots N^{\ell-1}$ iterates over the neurons of layer $\ell - 1$). Each connection has a weight $w_{qp}^{\ell} \in W$ (where $p = 1 \ldots N^{\ell}$ iterates over the

neurons of layer ℓ) associated with it determining the strength of the connection. The first weight ($w_{0p}^{\ell} \in \boldsymbol{W}$) refers to the bias value (b_p^{ℓ}), with its associated activation usually, but not necessarily equal to one. Activation a_p^{ℓ} of neuron p is calculated by aggregating the products of prior activations and their associated weights. This aggregation is marked as z_p^{ℓ} for convenience.

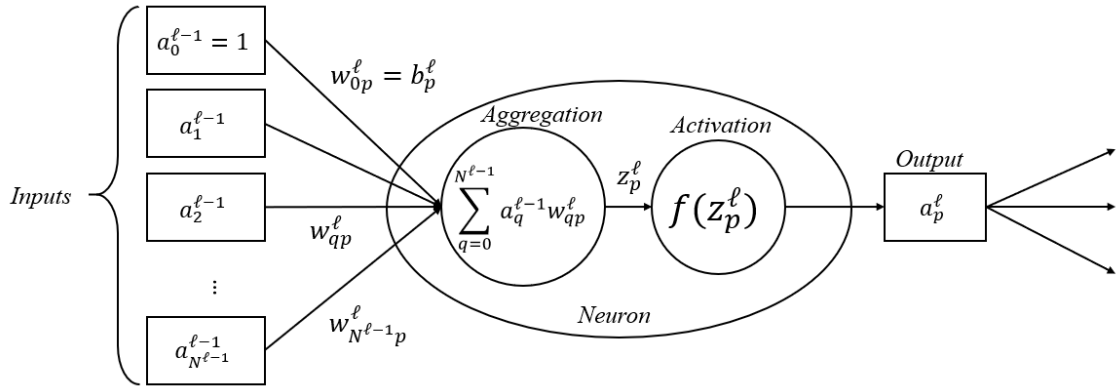

Figure 11: Simple mathematical model for a neuron. Based on Russel and Norwig (2010)

Mathematically, neurons perform a sum of products between activations of the previous layer and their respective weights, then apply a function on the aggregation:

$$z_p^{\ell} = \sum_{q=0}^{N^{\ell-1}} a_q^{\ell-1} w_{qp}^{\ell}$$

$$a_p^{\ell} = f(z_p^{\ell})$$

This function is the activation function (f). The most common are sigmoid, tangent hyperbolic, RELU and leaky RELU:

$$sigmoid: f(z) = \frac{1}{1 + e^{-z}}$$

$$tanh: f(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$RELU: f(z) = max(0, z)$$

$$Leaky\ RELU: f(z) = max(\varepsilon z, z); 0 < \varepsilon \ll 1$$

Each of these activation functions were based and developed on practical considerations, rather than on observed brain activity. This is one of the reasons why neural networks cannot be considered accurate mathematical models of the human brain. These activation

functions map a value from (-∞, ∞) to a set interval, and must be differentiable over the same interval. Differentiability is an important aspect of activation functions, which will become clear once back propagation is introduced.

In total, two neural network architectures can be developed:

1. **Feed-forward networks**: connections between neurons form a directed acyclic graph. They have no internal state other than their weights.
2. **Recurrent networks**: feeds output back into its own inputs. Their initial state depends on prior inputs as well as the weights, making them adept at modeling memory. They are better suited for problems possessing inherent sequential and temporal patterns, for example, text processing and NLP problems. They can be useful for intrusion detection if the intrusion detector is tasked to evaluate sequences of network packets. This temporal characteristic is not available for the benchmark datasets I used, therefore, though they have potential, this chapter will not discuss RNNs any further.

Feed-forward neural networks are structured into layers (Figure 12), collections of neurons taking inputs from neurons in a preceding layer and propagating their output to neurons in the following layer. A layer receiving inputs from the environment is called an input layer, a layer propagating its outputs to the environment is called an output layer. All the remaining layers between input and output layers are called hidden layers.

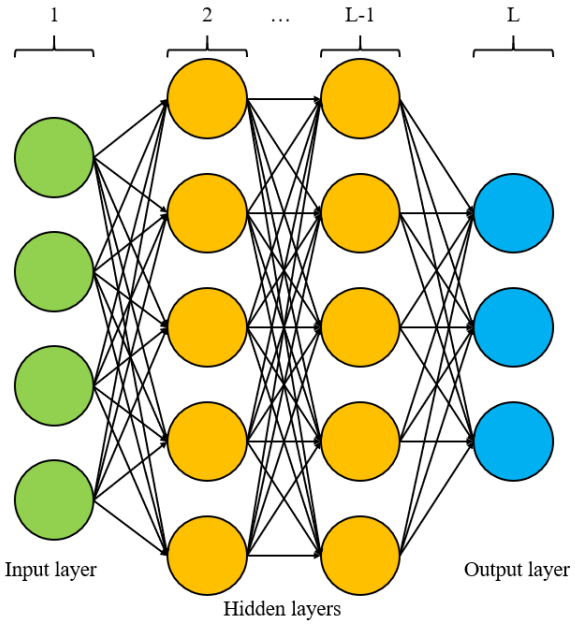

Figure 12: Architecture of a multilayer feedforward neural network. Source: own edit

Multilayer feed-forward neural networks are surprisingly flexible models. They can support both classification and regression problems. McCulloch and Pitts (1943) argued that a network constructed by a sufficiently large number of neurons is capable of approximating any desirable function with categorical or numerical output. As the primary use case of neural networks in intrusion detection is signature detection, primarily a classification task, the classification aspects of neural networks will be the primary focus of this chapter.

One challenge for multilayer feed-forward networks is how to produce more than a single output value. In this case, original expected output vectors are available, indicated as y_i. To quantify the performance of the network, the activations of the output layer ($a_i^L = \hat{y}_i$) are compared to this expected output. This is performed with the help of loss functions. In classification a particular loss function is the cross-entropy loss:

$$Loss = -\sum_{i=1}^{n} y_i log(a_i^L) + \frac{\alpha}{2}\|W\|_2^2$$

The point of a loss function is to take two vectors of class probabilities and compare them to one another. The difference between ground truth values and predictions is the loss for a given observation. Calculate the arithmetic mean of this loss for all observations to get the global loss of the network. Other popular loss functions include mean squared loss (or mean squared error, MSE) function, measuring ANN performance in regression tasks.

Calculating cross-entropy loss, however, requires class membership vector values to be interpretable on a (0, 1) interval each with one single value much closer to one than the rest. The softmax function can calculate vectors with such criteria:

$$softmax(a_i^L) = \frac{e^{a_i^L}}{\sum_{j=1}^{k} e^{a_{ij}^L}}$$

Where k stands for both the number of classes and the number of output neurons. The expected output y_i is a vector of length k, where $y_{ij} = 1$ if and only if the observation represented by y_i belongs to Y_j, otherwise 0. The softmax function takes a vector of length k and returns a vector at the same length with class membership probabilities. The probability located at index j will be the highest, if an observation belongs to Y_j. This way,

the loss for one observation and the overall loss for all observations can both be calculated.

One additional element in the loss function is $\frac{\alpha}{2}\|W\|_2^2$, commonly referred to as L2 regularization or ridge regression. The purpose of L2 is to penalize weight updates too great in magnitude, preventing the neural network model from overfitting the data. The magnitude of this regularization penalty is controlled by parameter α. Other regularizations are L1 and elastic net regularizations. L1 regularization (or lasso regression) is denoted as $\alpha\|W\|_1$. Its main purpose, like L2, is to prevent the network from overfitting, but it regularizes weights (w_{qp}^{ℓ}) to zero more, therefore it is suitable for feature selection or for enforcing weight sparsity. Finally, elastic net regularization combines the benefits of L1 (weight sparsity) and L2 (small coefficients) regularizations. In an elastic net α is multiplied by an additional component controlling the tradeoff between L1 and L2 regularization. Apart from L1, L2 and elastic net, other regularization techniques are available as well. One such example is the dropout rate (Srivastava *et al.*, (2014)), where, during training, a fraction of neurons are temporarily excluded from the model at each iteration, introducing randomness to neuron activations at each hidden layer, making the network overall more robust and generalize better on unseen data.

Learning in a neural network is synonymous with the minimization of the mean loss function. This optimization process involves the iterative incremental adjustment of W by taking the partial derivative of the loss function with regards to the weights. This is simple considering only the output layer; however, the true challenge lies in propagating the loss over to the hidden layers. This challenge has been solved when backpropagation was introduced.

Backpropagation propagates the loss measured at the output layer towards the input layer. To do this, backpropagation has to determine the sensitivity of the loss function to the weights. This is performed for each weight by applying the chain rule twice:

$$\nabla Loss_W \leftarrow \frac{\partial Loss}{\partial w_{p,N^{\ell}}^{\ell}} = a_{N^{\ell}}^{\ell-1} * f'(z_p^{\ell}) * \frac{\partial Loss}{\partial a_p^{\ell}}$$

Where

$$\frac{\partial Loss}{\partial a_j^{\ell}} = \begin{cases} \sum_{p=0}^{N^{\ell+1}} w_{p,N^{\ell+1}}^{\ell+1} * f'(z_p^{\ell+1}) * \frac{\partial Loss}{\partial a_p^{\ell+1}}, & if\ \ell\ is\ a\ hidden\ layer \\ a_i^L - y_i, & if\ \ell\ is\ an\ output\ layer \end{cases}$$

With the formula above, the algorithm calculates the gradient vector ($\nabla Loss_W$), holding information on how much each weight needs to change to minimize the loss function:

$$W^{t+1} = W^t - \eta \nabla Loss_W^t$$

Where t is the iteration step and η is a special parameter called learning rate. It is a model hyperparameter controlling the size of a step at each iteration to ensure the training reaches a global minimum. It is a sensitive value, set it too low and training will take a long time, set it too high and the model will fail to converge, or it will even diverge. More advanced optimization methods permit a dynamic learning rate, enabling the training process to start from higher learning rates (faster) and end on lower learning rates for better convergence. For example, inverse scaling learning rate reduces the initial learning rate by dividing it with the current iteration step (t) raised to a predetermined value.

An iteration can be one complete pass over all (X, y) input-output pairs. This is computationally expensive, other methods, like minibatch stochastic gradient descent (SGD) are preferred, where, the algorithm uses small slices of input observations before a single weight update, repeated for all X. A whole pass of the entire input in SGD is referred to as an epoch, which in turn repeats until a set number or convergence is reached. The other improvements to SGD involve Adam (Kingma and Ba (2014)), which introduced adaptive bias-corrected first and second moments to gradient descent for automated weight adjustments. Adam has been widely adopted as a solver for neural networks.

An advantage of neural networks is that they operate as universal function approximators. Given enough time and input, they can learn non-linear functions of any complexity.

The disadvantages of neural networks are:

- The algorithm has no guarantees to finding global optimum, neural network instances must be trained multiple times with different weight initializations.
- They tend to overfit presented data. This can be offset by applying L1 or L2 regularization.

- Neural networks require the tuning of several hyperparameters, such as learning rate, the number of hidden layers and the number of neurons per hidden layer. Hyperparameter optimization strategies are required to find an optimal value for each.
- Neural networks are sensitive to feature scaling, mitigated by feature normalization.
- Neural networks are often too complex for a human observer to understand, often referred to as black box systems.

Autoencoder Networks

Autoencoder networks are unsupervised neural network algorithms created when the target vectors are set to be identical to the input vectors. They are particularly useful in finding outlier patterns, a characteristic that can be exploited for anomaly detection. The architecture of a basic autoencoder network is available in Figure 13. More complex versions of this model have been designed, although all of them can be divided into an encoder, learning interesting patterns about the input data, a bottleneck creating a limited representation, and the decoder reconstructing the input from this limited representation.

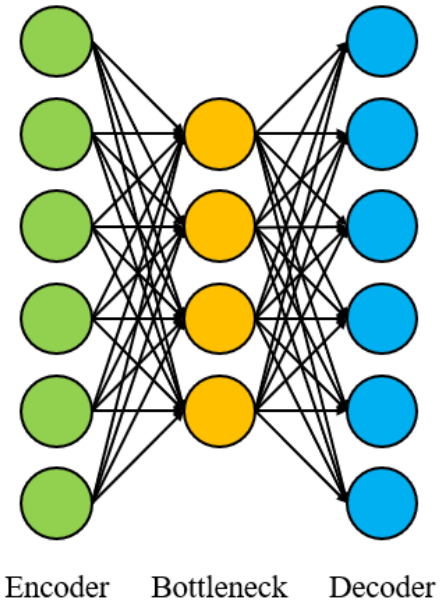

Figure 13: Architecture of an autoencoder network. Source: own edit

Training on an autoencoder is performed using the same backpropagation process used to train feed-forward neural networks. The most important differences lie in the network

architecture, the choice of true output to compare predicted outputs to and, in the case of intrusion detection, in which class of traffic is provided for the network to train on. With these considerations an autoencoder is trained the following way:

- The data is split to normal and anomalous (~attack) traffic
- The autoencoder is trained only using normal traffic, thus it learns patterns between normal connection features only
- Reconstruction loss on new normal connections is expected to be lower, and higher on attacks. This reconstruction loss, unlike with fully connected neural networks, is the half of the mean squared error function:

$$Loss_{AE} = \frac{1}{2n}\sum_{i=1}^{n}\|\hat{x}_i - x_i\|_2^2$$

Where x_i stands for the true input observations and $\hat{x}_i$ is the reconstructed input.

So far, I only introduced dense autoencoders with one hidden layer, but more complex autoencoder networks exist. By introducing additional hidden layers, one can construct an autoencoder capable of learning nonlinear relationships between the input features. These multilayer autoencoders are often referred to as deep autoencoders.

A second restriction imposed on deep autoencoders is that neuron counts in encoder layers must be monotonically decreasing and monotonically increasing in decoder layers. This restriction can be lifted, by introducing a sparsity constraint to the network (Ng and others, (2011)). Sparsity in all neural networks refers to the sparsity of activations when a selected x_i observation is fed to the network. Its main difference compared to dropout is that sparsity is maintained even after training has ended and a given q^* neuron might activate for some x_i input and not for others. Sparsity can be achieved by applying a constraint as regularization:

- Regularize the loss function with L1, as lasso regression encourages sparsity.
- Use Kullback-Leibler divergence (KL divergence). KL divergence is a measure of difference between two distributions. When used for sparsity in autoencoders, it penalizes the average activation of all neurons in all layers to a predetermined rate, indicated as ρ. The formula of KL divergence for hidden neuron q:

$$\sum_{q=1}^{N^{\ell}} KL(\rho||\hat{\rho}_q) = \sum_{q=1}^{N^{\ell}} \rho log \frac{\rho}{\hat{\rho}_q} + (1-\rho) log \frac{1-\rho}{1-\hat{\rho}_q}$$

Where $\hat{\rho}_q = \frac{1}{n}\sum_{i=1}^{n} a_q^{\ell}(x_i)$ is the average activation of q over all x_i inputs. This formula, like L1, is added as a regularization constraint to the loss function of the autoencoder:

$$Loss_{SAE} = \frac{1}{2n}\sum_{i=1}^{n} \|\hat{x}_i - x_i\|_2^2 + \beta \sum_{q=1}^{N^{\ell}} KL(\rho||\hat{\rho}_q)$$

Where β controls the effect of the KL divergence penalty on the loss function. The first part of the formula remained unchanged from ½ MSE. With sparsity introduced, neuron counts in encoder and decoder layers are permitted, and even encouraged, to increase beyond the number of preceding, as only a handful of them will be active at a time. An autoencoder regularized by sparsity constraints is called a sparse autoencoder (SAE).

The last variants of autoencoders I detailed are called variational autoencoders (VAE), developed by Kingma and Welling (2013). Variational autoencoders are created when, instead of learning an arbitrary function, the model learns the parameters of a multidimensional distribution. Compared to previous AEs, this model can not only reduce input dimensionality, but it is also capable of providing new samples itself. In this regard VAEs can be considered as generative models. This is achieved by dividing the bottleneck to mean and standard deviation vectors of neurons. The outputs of these two are used together with a random variable drawn from a predetermined distribution (usually normal) to generate new samples.

The loss of this model is the same as with sparse autoencoders: reconstruction loss regularized by KL divergence between the learned latent distribution and the prior distribution.

An extension of VAEs can also be fed with classification target class values (y_i) as a separate one hot encoded input. Then, the model is trained to learn not only a single latent distribution, but a set of latent distributions for each y_i. This model is called conditional variational autoencoder (Sohn, Lee and Yan, 2015). CVAEs allows more control over the generated samples, for example, generate observations per intrusion type to train an IDS.

I decided to use fully connected deep autoencoder networks with no regularization for the model introduced in chapter 4.2.4 as part of a hybrid intrusion detector.

2.2.4. ADDITIONAL TECHNIQUES USED IN DATA MINING

Machine learning algorithms form the core techniques data scientists use; however, they use other tools to assist them with data processing, model testing and model performance improvement. In this chapter I will briefly introduce these tools in more detail: model ensembles, synthetic sampling, hyperparameter optimization and model evaluation metrics.

Ensemble methods

The idea behind ensemble methods is to combine multiple machine learning models to get an aggregate prediction with the goal to provide better results than what any of them could achieve. In this chapter I use the term base models to describe the different machine learning models that contribute to the ensemble, and aggregate model to describe the ensemble. This chapter introduces the three most common ensemble models: bagging, boosting and model stacking, based on Smolyakov (2017) and Budzik (2019).

Bagging, or bootstrap aggregation aims to sample the training data with replacement (bootstrap sampling) to create an ensemble of models (Figure 14). This sampling process is repeated for each base model, and the final decision is calculated as either an arithmetic mean or a simple majority vote of base model predictions. Bagging is most effective when the base models have low bias but high variance, typically random forests.

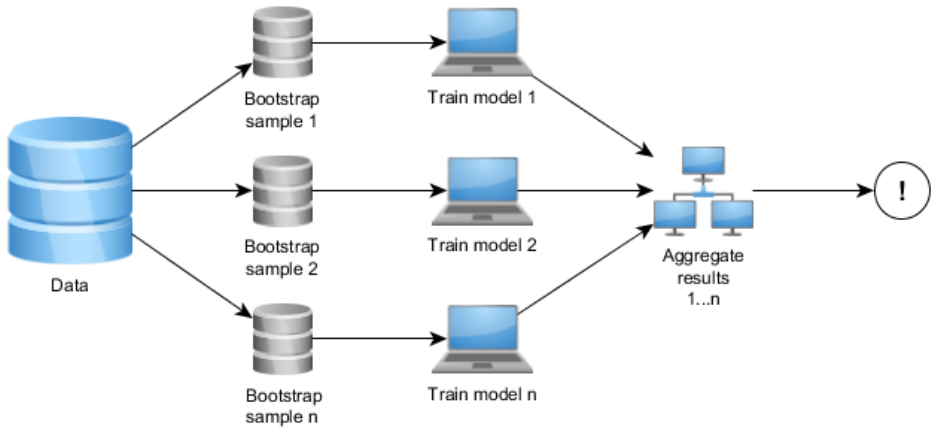

Figure 14: Bagging model training process. Based on Budzik (2019)

With **boosting**, performance is improved by concentrating modeling efforts on errors made by weak models (Figure 15). These base models are trained sequentially, where incorrectly predicted observations are weighted more than correct ones. The aggregate

boosting result is calculated either as a weighted arithmetic mean or by weighted majority voting. Models with low variance and high bias are well suited for boosting, for example, gradient boosting.

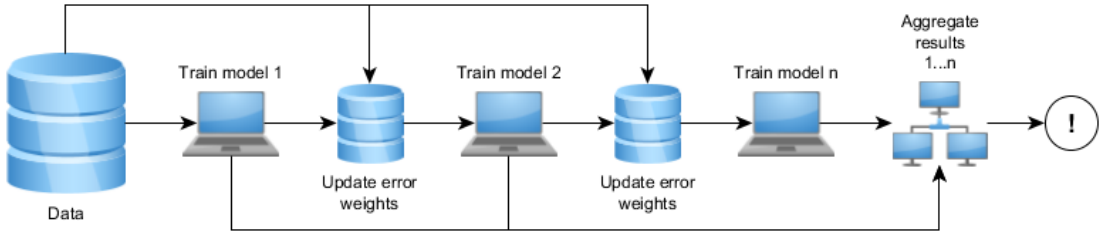

Figure 15: Boosting model training process. Based on Budzik (2019)

With model **stacking**, base model results are combined using a meta-model. This could be as simple as a linear function of the intermediary results, or a complex machine learning model itself (Figure 16). Stacking, compared to boosting and bagging, can reduce model variance and bias at the same time, providing powerful aggregate predictor models. This improvement stems from the heterogeneity of the base models, which could be achieved in two ways: by training models of the same kind, but on different feature sets, or by training different machine learning models (more common). Considering the advantageous property of simultaneously reducing variance and bias in model predictions, I decided to use this ensemble design for my intrusion detectors.

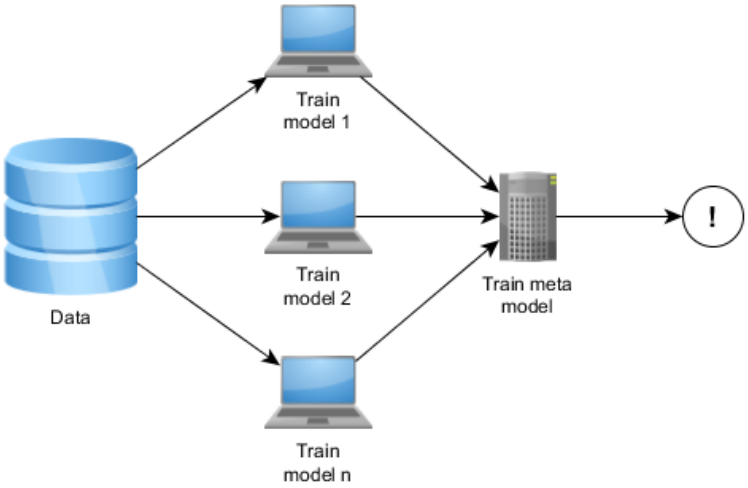

Figure 16: Stacking model training process with results combination. Based on Budzik (2019)

Ensemble models can improve results by reducing model variance, bias or both. Therefore, they are useful for creating aggregate models with improved classification or

regression performance. A drawback of model ensembles is an increase in complexity as multiple models have to be trained and maintained simultaneously.

Synthetic sampling methods

In classification, a way to combat imbalanced class values is to under-sample the majority class, or to over-sample the minority class. However, when class imbalance is too great, more sophisticated methods are needed. One such sophisticated method is synthetic sampling, where a machine learning model is trained to recognize relationships between data and a target feature with the goal to provide new artificial samples for minority classes, or to reduce the number of observations of the majority class, while maintaining patterns that make the target recognizable still. There are three methods for synthetic sampling:

- Over-sampling methods,
- Under-sampling methods and
- Combination of over- and under-sampling methods

The first oversampling method is SMOTE (synthetic minority over-sampling technique), developed by Chawla *et al.* (2002). In SMOTE, the minority class is over-sampled by selecting one observation from minority at a time and introducing new synthetic observations at random along the line connecting the selected observation and one of its k nearest neighbors from the same minority class. An advantage of SMOTE is that it forces the following machine learning model to create larger and less specific decision regions between classes forcing them to generalize better.

By itself, I used SMOTE only in one model, however it is important building block for more advanced synthetic sampling methods, like SVM SMOTE, recommended by Nguyen, Cooper and Kamei (2009). The core purpose of SVM SMOTE remains the same, but instead of using the k nearest neighbor algorithm only, it also applies the maximum margin classification of SVMs to sample observations from border regions only. The benefit compared to SMOTE that it samples the border regions between majority and minority classes, thus improving model generalization even further. The drawback is that neither k nearest neighbors, nor SVMs are recognized for their fast execution on large amounts of data. Later, it has been empirically proven (by Lopez-Martin, Carro and Sanchez-Esguevillas (2019), for example), that models that were fed with observations

by an SVM SMOTE sampler, achieved higher classification performance, than those that were fed by other synthetic samplers.

Turning to under-sampling methods, the first technique used to under-sample the training data is called edited nearest neighbors (ENN) created by Wilson (1972). It is based on the k nearest neighbor algorithm, however, instead of sampling from a line between two neighbors, it removes observations which do not "agree" with their neighborhood enough. This agreement can, for example, be described by the relative distribution of minority and majority class values within the neighborhood.

A second technique (Tomek (1976)) is based on identifying Tomek links within the dataset. Two observations in a dataset form a Tomek link, if they are nearest neighbors of each other. Under-sampling with Tomek links is the removal of such observations either from the majority class only, or from the minority classes as well.

I did not use edited nearest neighbors or Tomek links under-sampling for my intrusion detectors directly. However, they both played a role in creating combined under-, and over-sampling methods. I used both ENN and Tomek links in tandem with SMOTE, first over-sampling the minority classes, followed by under-sampling the majority class. Further details of how this combination works can be found in Batista, Prati and Monard (2004).

Evaluation metrics

Due to the comparability of the performance classification models, the data mining community developed several evaluation metrics. For almost every metric I used, the input has been provided by the confusion matrix, available in Han, Kamber and Pei (2011) and in Table 1 as well. It shows the predictions made by the classifier in rows and the ground truth class values in columns. The cells contain the true positive (TP), false positive (FP), false negative (FN) and true negative (TN) predictions. Table 1 is a confusion matrix for binary classification, although it can be extended for the multiclass case as well.

		Ground truth	
		+	-
Prediction	+	True positive (TP)	False positive (FP)
	-	False negative (FN)	True negative (TN)

Table 1: Confusion matrix for classifier performance. Source: Han, Kamber and Pei (2011)

The most common metric based on the confusion matrix is **accuracy**:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

It is sensitive to high imbalance between target feature values; therefore, it is less useful for evaluating intrusion detection performance. Other metrics to use to extend accuracy are **precision, recall, F_1-score**, the receiver operating characteristic (**ROC**) curve and the area under the ROC curve (**AUC**). The formula of the first three:

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

$$Recall = \frac{TP}{TP + FN}$$

$$F_1\ Score = \frac{2TP}{2TP + FP + FN}$$

Precision measures the exactness of positive labeling, the coverage of the correct positive labels among all positive-labelled samples. Recall measures the completeness of positive labelling, the fraction of correctly labelled positive samples among all positive samples. It is often referred to as sensitivity and detection rate. The F_1-score combines the two in a weighted harmonic mean. The weight is almost always set to one, meaning precision and recall are treated equally important.

Precision, recall and F_1-score are per-class measures, meaning they provide multiple values for each class value in multiclass classification. Sometimes, it is more desirable to calculate one single aggregate value describing the trained model. For these situations, three averaging schemes were constructed by Pedregosa *et al.* (2011):

- **Micro**: calculates metrics globally by counting total true positives, false negatives and false positives.
- **Macro**: calculates metrics for each label and calculate their unweighted arithmetic mean. This does not take class imbalance into account, which makes it easier to highlight performance on minority classes.
- **Weighted**: calculates metrics for each class value and averages them weighted by the number of observations that belong to that class. This method does take class imbalance into account.

The efficiency of an intrusion detector can be measured with **false positive rate** as well, indicating the percentage of observations misclassified as positive over all observations:

$$False\ Positive\ Rate = \frac{FP}{TN + FP}$$

The ROC curve mentioned earlier visualizes all possible cuts between positive and negative predictions. It is a measure based on sensitivity and false positive rate (or 1-specificity) (Figure 17). In a ROC curve a good prediction with a good cut value converges to the top right or the bottom left corner (in the latter case, changing the class labels is a viable option). The diagonal line represents the results of random guessing.

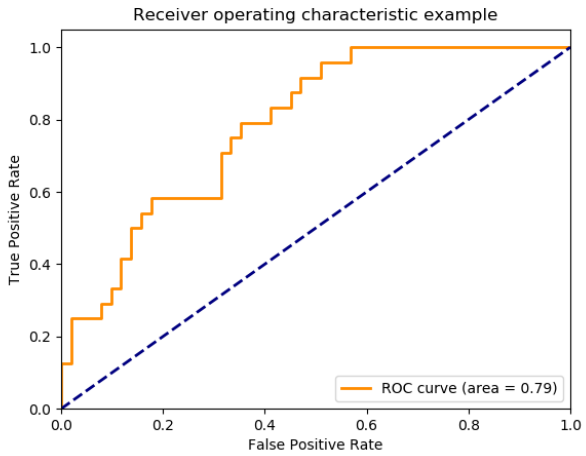

Figure 17: ROC curve. Source: scikit-learn developers (2018)

The formulas for sensitivity and specificity are:

$$Sensitivity = \frac{TP}{TP + FN}$$

$$Specificity = \frac{TN}{TN + TP}$$

ROC curve is a visualization technique, it is difficult to interpret quantitatively. AUC is a metric can be calculated from ROC curves, which is easier to interpret as a measurement of overall generalization ability. A 0.5 AUC score indicates random guessing, a value closer to 1 an almost perfect classification.

The metrics introduced so far are all used for evaluating classification performance. However, due to the application of autoencoder networks, and how they are evaluated, I found it useful to introduce one metric used for measuring regression performance. This

metric is the **mean squared error**, measuring the average squared difference of predictions from true values:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

Compared to the earlier formula shown in chapter 2.2.3, $\hat{y}_i$ and y_i refer to scalar ground truth and predicted target features, rather than vectorized features.

All these model performance metrics have their own unique characteristics, which makes choosing them more challenging. Intrusion detection is a classification task, where the difference in the representation of attacks compared to normal traffic can be uneven. Accuracy and precision are less useful metrics here, as both require class values to be equally distributed. Although, a case could be made for accuracy, due to how common it is even in papers studying intrusion detection. A second characteristic of intrusion detection, is how expensive the wrong classification of an attack as normal traffic is, compared to the reverse case. This calls for the importance of recall and false positive rate as values better characterizing this kind of error in face of imbalanced classes. Personally, I opted in to demonstrating the accuracy and recall achieved by my models, the latter for the reason I just described, and the former for its common appearance in intrusion detection literature. For aggregating recall, I decided to use macro averaging, highlighting the imbalanced nature of the dataset I worked with.

Hyperparameter optimization

Machine learning models require parameters set up prior to training. These parameters could directly influence the performance achieved by a model, therefore an automated approach for selecting these is crucial. This approach is called hyperparameter optimization, a method wrapped over regular train-test-evaluate process of machine learning. In this sense, the meaning behind the notations used for $\{X, y\}$ is slightly different for hyperparameter optimization: X indicates the hyperparameter space where optimization algorithms sample from, where X_s, s=1...m are the hyperparameters and $x_i, i = 1 \ldots n$ are samples from the hyperparameter space. The target feature (y) is an outcome performance metric of the internal machine learning model, where y_i is the metric achieved when the hyperparameter sample was x_i. The challenge is that no prior information is available on the value of y_i, but it can be estimated by calculating $\hat{y}_i$. The

goal of hyperparameter optimization is to find a x_i^* parameter combination, for which $\hat{y}_i$ reaches its maximum (or, in case y is a loss metric, its minimum).

First, I introduce the two most common methods for hyperparameter optimization, followed by more intelligent approaches. The common methods:

- **Grid search**: X is divided into equally sized segments (~grids) between $[x_{min}^s, x_{max}^s], \forall s = 1 \ldots m$. At each step a parameter combination from the grid is chosen for evaluation. The best performing combination of hyperparameters will be the final choice to train the machine learning model with. Grid search evaluates all grid combinations generated from X exactly once, in this sense it ensures optimal results; however, its time complexity increases exponentially with m and the number of grids selected for each X_s.

- **Random search**: randomly generates x_i values from X a set number of times. The final parameter combination (x_i^*) is determined by the best model performance. It is linear in the number of trials set in advance; therefore, it calculates faster compared to grid search strategy; however, it does not guarantee optimal hyperparameters.

Grid and random search both have their respective issues either with execution time or with performance. One idea to solve these issues is to find algorithms designed to optimize more intelligently, for example, Bayesian optimization with gaussian process priors (Brochu, Cora and De Freitas (2010) and Snoek, Larochelle and Adams (2012)) or tree-structured parzen estimators (Bergstra *et al.* (2011)).

Bayesian optimization is interested in finding the maximum of function $f(x) = y_i$ on a bounded set of hyperparameters. This function is expensive to evaluate; therefore, a probabilistic model is calculated to approximate it. Bayesian optimization uses all information (all earlier evaluations of $f(x)$) to approximate the target metric value. This results in a process that can find the maximum of $f(x)$ at additional computational cost, which is still lower than attempting to calculate an additional value of the function to be approximated. For Bayesian optimization to work, two choices must be made: first, a prior over function must be selected to approximate $f(x)$; second, one must chose an acquisition function to construct a utility function from the model posterior to calculate a new point in X to evaluate.

Gaussian processes (GP) are a good choice for the prior over function. GP is defined by the assumption that any finite set of points form a multivariate gaussian distribution. Each GP can be derived and are characterized by a mean function and a covariance (or kernel) function. The mean function can be set to return zero constantly for convenience. This leaves the covariance function, which is a choice between the squared exponential function (or RBF kernel), the Matérn kernel, the rational quadratic kernel, the exp-sine-squared kernel or the linear kernel.

Several popular choices are available for the acquisition function as well. These all determine which x_i in X should be evaluated next. In general, these functions depend on all previous $\hat{y}_i$ estimates, as well as the GP hyperparameters. This dependence on GP prior functions is characterized by the predictive mean function $\mu(x)$ and predictive variance function $\sigma^2(x)$. An additional formula is the best current value, which is denoted as $x_{best} = argmax_{x_i} f(x_i)$. The most common acquisition functions:

Probability of improvement:

$$PI(x) = \Phi(Z), \qquad where\ Z = \frac{f(x_{best}) - \mu(x) - \xi}{\sigma(x)}$$

Expected improvement:

$$EI(x) = \begin{cases} (\mu(x) - f(x_{best}) - \xi)\Phi(Z) + \sigma(x)\phi(Z), & if\ \sigma(x) > 0 \\ 0, & if\ \sigma(x) = 0 \end{cases}$$

GP upper confidence bound:

$$UCB(x) = \mu(x) + \kappa\sigma(x)$$

In the formulas above $\Phi(\cdot)$ and $\phi(\cdot)$ are the normal cumulative distribution (CDF) and probability functions (PF) respectively. ξ and κ are parameters controlling the tradeoff between exploitation and exploration for the two improvement functions and the upper confidence bound function.

In fact, exploitation (associated with the mean function) and exploration (associated with the variance function) are important concepts to Bayesian optimization. The first means that new x_i recommendations will be calculated where earlier evaluations yielded higher target metric values. Exploration on the other hand encourages the evaluation of regions

where uncertainty is high. Finding the balance between the two is one of the core elements of Bayesian hyperparameter optimization.

A more demonstrative example of Bayesian optimization can be seen in Figure 18. In the figure, solid back line denotes the prior (and posterior) mean, blue shaded areas the prior (and posterior) uncertainty and black dotted line shown the true mean of function $f(x)$. The acquisition function is denoted as a green line. The next x_i value to evaluate are proposed based on the maximum of the acquisition function.

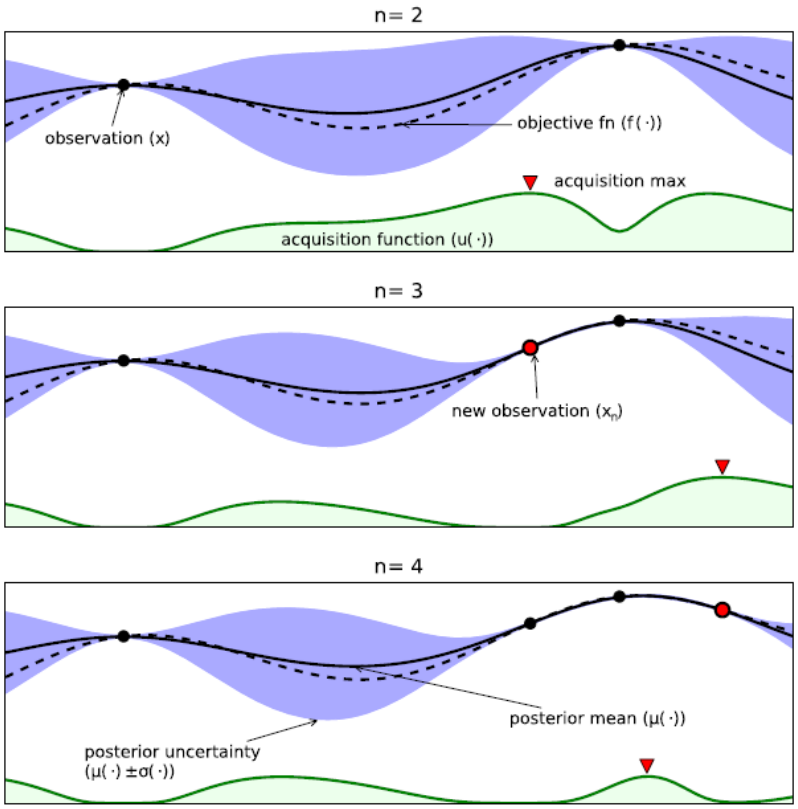

Figure 18: Illustration of the Bayesian optimization. Source: Brochu, Cora and De Freitas (2010)

Bayesian optimization is a more purposeful approach compared to random search and converges in less total iterations compared to grid search. However, it has drawbacks: the more purposeful hyperparameter recommendations require additional calculation overhead (cubic in the dimensions of the hyperparameter space (indicated as m)), which makes Bayesian optimization less suitable for simple machine learning models or when the model has a lot of parameters to set. Moreover, kernel functions and the tradeoff between exploration and exploitation are themselves hyperparameters to the optimization process itself. Finally, Bayesian optimization can only yield continuous numeric

hyperparameters and it cannot provide hyperparameter values dependent on other hyperparameter values.

These drawbacks have been highlighted and addressed by the tree-structured parzen estimators (TPE) approach by Bergstra *et al.* (2011). Where Bayesian optimization modeled $P(y|X)$ directly, TPE models $P(X|y)$ and $P(y)$. This $P(X|y)$ is modeled by transforming the tree-structured generative process by replacing the distributions with non-parametric densities. Using different observations in these non-parametric densities, these substitutions represent a learning algorithm that can produce a variety of densities over the space of hyperparameters. TPE defines two densities for $P(X|y)$:

$$P(X|y) = \begin{cases} h(X), & if\ y > y^* \\ g(X), & if\ y \leq y^* \end{cases}$$

Important to note that TPE is an algorithm which minimizes $\hat{y}_i$, rather than maximizing it. Though this change is only technical, it does affect function notation. In the formula above, $h(X)$ represents the density formed using x_i hyperparameter samples lower than a selected threshold y^*, while $g(X)$ represents distribution from all the remaining observations. Unlike GP, TPE supports a soft y^* threshold, in order to keep some x_i samples from X to formulate $h(X)$. For example, this y^* can be chosen to be some quantile γ of the observed $\hat{y}_i$ values. TPE itself optimizes the expected improvement acquisition function, formulated as:

$$EI_{TPE}(X) = \left(\gamma + \frac{g(X)}{h(X)}(1-\gamma)\right)^{-1}$$

To maximize improvement, x_i hyperparameter combinations which have high probability under $h(X)$ and low probability under $g(X)$ should be selected. The tree structure enables it to easily draw many x_i parameter combinations to evaluate. The ones with the highest expected improvement are selected at each iteration.

TPE addresses issues with numerical only and independent hyperparameter features as well. It permits sampling discrete numerical and categorical distributions and extends continuous numerical values sampling with more than one distribution type to sample from with the help of stochastic expressions shown in Table 2. The attribute value for *label* is common for all the expressions, indicating an internal name of a given hyperparameter for better tracking.

Expression	Description
hp.choice(label, options)	Returns an element from an *options* list. Elements can be nested stochastic expressions.
hp.pchoice(label, p_options)	Returns an element from a list of tuples in the format *(prob, option)*. Permits the user to enforce bias among the choices.
hp.uniform(label, low, high)	Draws uniformly between *low* and *high*.
hp.quniform(label, low, high, q)	Draws uniformly between *low* and *high*. Better suited for handling discrete values.
hp.loguniform(label, low, high)	Draws values that are uniform in their exponent, from the interval $[e^{low}, e^{high}]$.
hp.qloguniform(label, low, high, q)	Draws values that are uniform in their exponent, from the interval $[e^{low}, e^{high}]$. Better suited for handling discrete values.
hp.normal(label, mu, sigma)	Draws a real value from a normal distribution with *mu* mean and *sigma* standard deviation.
hp.qnormal(label, mu, sigma)	Draws a discrete value from a normal distribution with *mu* mean and *sigma* standard deviation.
hp.lognormal(label, mu, sigma)	Draws values whose exponent is normally distributed with *mu* mean and *sigma* standard deviation.
hp.qlognormal(label, mu, sigma, q)	Draws values whose exponent is normally distributed with *mu* mean and *sigma* standard deviation. Better suited for handling discrete values.
hp.randint(label, upper)	Returns a random integer in the range $[0, \text{upper})$. No additional correlation is assumed between closer integer values compared to distant values during optimization.

Table 2: Hyperopt (python implementation of TPE) stochastic sampling functions. Source: Bergstra, Yamins and Cox (2013)

Apart from these improvements to the hyperparameter value definitions, TPE also improved execution time from being cubic in m to being linear in both m and n. With these advantages and considering the availability of a python implementation, I decided to use TPE hyperparameter optimization to improve my intrusion detectors.

This concludes the data mining and machine learning context of my dissertation. In this chapter I introduced supervised and unsupervised learning and the most common algorithms in intrusion detection research from each. The introduction of neural networks and autoencoders, due to their importance to my work, received their own chapters. Finally, in the last part of this chapter I have shown additional techniques that I used to evaluate and improve the detection performance of my proposed models. Chapter 2.3 introduces the pivotal early works on intrusion detection and reviews the research conducted in the literature.

2.3. INTRUSION DETECTION RESEARCH – RELATED WORKS

The aim of this chapter is to briefly introduce the field of intrusion detection research with articles that studied it using machine learning models. I will start this chapter with the

most important studies in the field and follow it up with dedicated survey papers to give a quick look on which machine learning models can be used for intrusion detection. Then, I will discuss further articles that either focused on or provided single-model detectors as part of their research. Afterwards, I introduce papers evaluating ensemble models and followed by those evaluating hybrid models. Special cases of hybrid models were extended with variational autoencoders. I discussed them in connection with hybrid intrusion detectors. I highlighted papers that used at least one of two additional techniques for intrusion detection as well: synthetic sampling and hyperparameter optimization. Finally, I close this chapter by listing the issues in the field formulated by the survey papers introduced earlier.

One of the first studies of intrusion detection from a data mining perspective was Stolfo *et al.* (2000). They discussed the 1999 DARPA dataset for anomaly and misuse detection. They performed feature selection, classification, frequent pattern detection and sequence analysis. By the end of feature selection, the original traffic features were divided into four categories:

- **Intrinsic features**: features describing all network connections.
- **Time-based traffic features**: aggregate features describing connections that had the same destination host or service as a selected connection in the prior 2 seconds.
- **Host/service-based traffic features**: same as above, but instead of a 2 second aggregation window, the authors used the previous 100 connections.
- **Content features**: features describing the content of the traffic.

These four groups of features were used to train three machine learning models with the RIPPER algorithm for rule construction (RIPPER: a rule induction algorithm based on the “divide and conquer” principle). The target variable consisted of 5 classes: DoS (denial of service attacks), R2L (unauthorized access from a remote machine), U2R (unauthorized local access to superuser privileges), probe (traffic surveillance), and normal behavior.

Stolfo *et al.* (2000) expected the three models to perform better on different feature groups:

- **The time-based traffic model**: containing intrinsic and time-based traffic features. This proved to be the best for detecting DoS and probing attacks.

- **The host-based traffic model**: containing intrinsic and host-based traffic features, best for detecting slow probing attacks.
- **The content model**: containing intrinsic and content features, designed specifically for detecting R2L and U2R attacks.

The three models were then combined into a meta-learner, which decided on the best performing models for each connection. Though not explicitly stated in the paper, this approach can be considered as a model ensemble.

Stolfo *et al.* (2000) provided an important study of intrusion detection and one of the first benchmark datasets, the KDD Cup 1999, however the data they used were not without criticism. The most prominent of which can be read in McHugh (2000). His criticisms can be traced back to the unit of analysis problem: a single attack pattern can be tied to a single connection package, or to multiple packages over time, formulating a flow. This causes issues with evaluation methods used by Stolfo *et al.* (2000) and other participants analyzing the KDD Cup 1999 dataset. A second criticism of the dataset by McHugh (2000) complained about the underlying taxonomy: it has been developed from the attacker's perspective. This provides additional information for detection algorithms that may not be available in a realistic scenario. Instead, McHugh (2000) proposed a classification scheme based on the protocol layer and the protocols used, or whether a completed protocol handshake is required to carry out an attack. Attack distributions were unrealistic as well, which have been noted first a decade later by Tavallaee *et al.* (2009).

Both the training and test datasets of KDD Cup 1999 contained a large number of redundant records (78% and 75%, respectively), which caused machine learning algorithms to have biased predictions, first highlighted by Tavallaee *et al.* (2009). Instead, they proposed a new dataset, the NSL-KDD dataset having better balanced target classes, no redundancy and less observations overall.

The works of McHugh (2000), Stolfo *et al.* (2000) and Tavallaee *et al.* (2009) were pivotal, but not the only ones in intrusion detection. In their literature review, Tsai *et al.* (2009), for example, wrote about intrusion detection research between 2000 and 2007. The authors reported that single model classifiers were used the most, however by 2008, hybrid classification techniques also began to gain attention. Ensemble models were not analyzed in depth, partly due to how these works contributed only a small fraction in the evaluated literature (only ~11%).

Moreover, Tsai *et al.* (2009) have taken a look at two additional characteristics of intrusion detection literature: the datasets used and whether feature selection was considered by a paper or not. They, just as Bhuyan, Bhattacharyya and Kalita (2014), determined that most of the available literature used one of three datasets: KDD Cup 1999, DARPA 1998 and DARPA 1999, being the few available benchmark datasets at the time.

Bhuyan, Bhattacharyya and Kalita (2014) identified six methods used for network anomaly detection: statistical methods, classification, clustering and outlier detection, soft computing, knowledge-based models and combination learners (Figure 19) Out of them, classification, clustering, outlier analysis, soft computing algorithms (specifically artificial neural networks) and combination learners were the most researched areas.

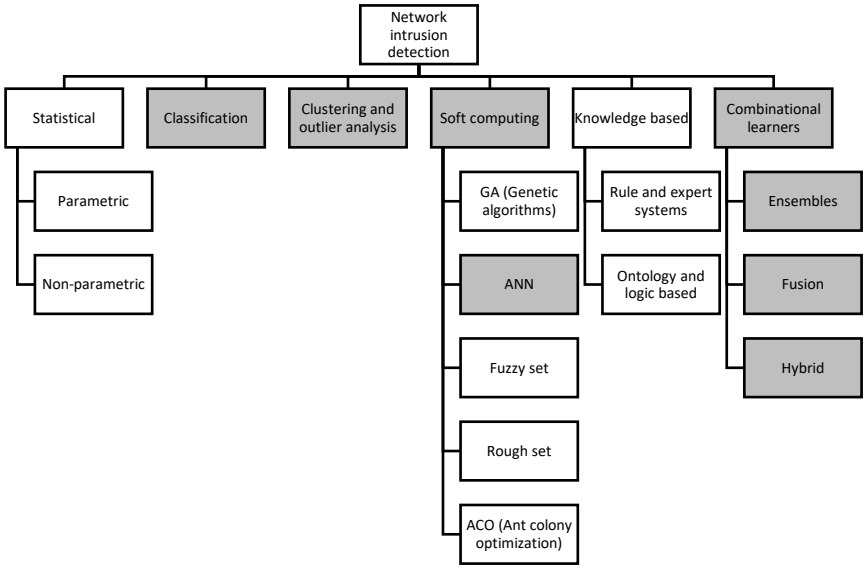

Figure 19: Classification of network anomaly detection methods. Source: Bhuyan, Bhattacharyya and Kalita (2014)

Figure 20 shows the model classification scheme set up by of Ippoliti (2011). Compared to Bhuyan, Bhattacharyya and Kalita (2014), he grouped classification, clustering and outlier analysis under machine learning, distributed elements of soft computing between the remaining four categories and identified knowledge based and combination learners as rule based and hybrid approaches.

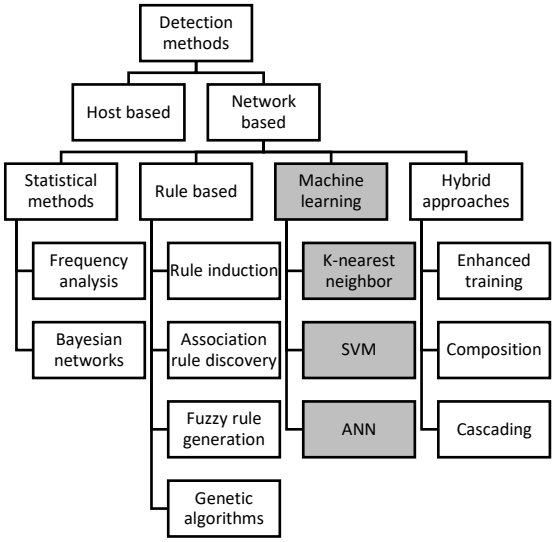

Figure 20: Relationship between detection methods. Source: Ippoliti (2011)

Buczak and Guven (2015) discussed the potential of using data mining and machine learning algorithms for intrusion detection, more particularly for signature detection, anomaly detection and hybrid approaches, the latter two combined into one category due to their low representation in the studied literature. The covered algorithms can be seen in Figure 21.

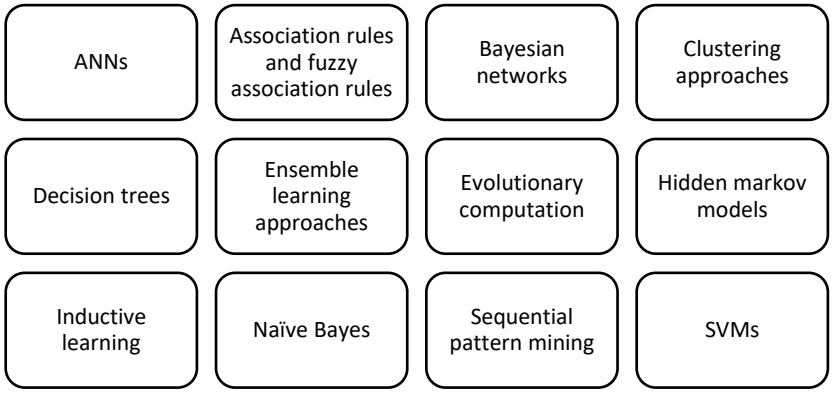

Figure 21: Machine learning approaches in intrusion detection. Coverage of Buczak and Guven (2015)

The most common metrics for classifier evaluation in the literature were accuracy, detection rate (often referred to as recall or sensitivity), false alarm rate (or false positive rate) and AUC based on ROC curves. Recall and false positive rate better describe model performance in intrusion detection, as attacks that remain undetected are more harmful for an organization, as legitimate connections being detected as attacks. Accuracy for intrusion detection, due to the unbalanced nature of attack classes, is far less informative.

A more recent survey created by Molina-Coronado *et al.* (2020) evaluated intrusion detection research from the perspective of the KDD process of Fayyad, Piatetsky-Shapiro and Smyth (1996). Therefore, they provide a more holistic review:

- **Data selection**: the system from which data is collected matters a lot. One can collect data from root network devices, covering a broad range of network devices at the cost of having less data on horizontal traffic, or from access devices, which provides more data on horizonal communication, but for less network devices. Network traffic itself can be interpreted on packet and flow level, which complicates analysis further.

- **Construction of data features**: it is tasked to acquire features from the captured raw traffic data. This includes both explanatory and target features, each having unique challenges associated. Explanatory features can come from the various packet headers, or from the content, and can describe one or more connection flows. Features from each must be collected if the goal is to create an intrusion detector for a wide array of attack patterns. Additionally, labelling attacks might be an even greater challenge.

- **Data preprocessing and transformation:** involve feature noise reduction (outlier and missing value imputations), categorical feature encoding, continuous feature discretization and numerical feature scaling.

- **Data reduction**: data reduction can be achieved by reducing the number of features or of traffic observations. The former can be achieved by selecting useful features or by projecting explanatory variables into a lower dimensional space. For example, PCA and AE can be used for dimensionality reduction. Sample dimensionality reduction (where an observation in the sample represents more than one observation from the old dataset) is less researched within the field.

- **Data mining**: data mining for intrusion detection can take the form of misuse, anomaly and hybrid detection. In misuse detection, the most common algorithms used were ANNs, SVMs, k-nearest neighbor, naïve Bayes algorithms and decision trees. The combination of these methods into ensemble models was also proposed by multiple papers. Hybrid detection was divided into four categories, shown in Figure 1 and discussed in chapter 2.1. A second, less common taxonomy divided intrusion detection to batch and incremental learning. Batch learning is more

common, while incremental learning is better suited for stream data processing architectures.

- **Evaluation**: prediction performance is only one of many evaluation criteria an intrusion detector can face, however papers published study this characteristic almost exclusively. Furthermore, not every metric is equally useful, due to the target class imbalance experienced in the data and the fact that attacks might have more severe consequences for the victim.

The most used single-model methods for intrusion detection were ANNs, SVMs, decision trees, k-nearest neighbor algorithms and naïve Bayes algorithms. ANNs, decision trees and SVMs performed well as intrusion detectors, not without drawbacks though: in general, ANNs and SVMs are time intensive to train, while ANNs and decision trees are more susceptible to overfitting. I found the following papers studying single model signature detection either as their pronounced focus, or as part of a comparison with more advanced ensemble or hybrid detectors: Bouzida *et al.* (2004), So-In *et al.* (2014), Elhag *et al.* (2015), Petersen (2015), Aghdam and Kabiri (2016), Hasan *et al.* (2016), Almseidin *et al.* (2017), Yin *et al.* (2017), Ingre, Yadav and Soni (2017), Divekar *et al.* (2018), Parampottupadam and Moldovann (2018), Sakr, Tawfeeq and El-Sisi (2019), Sapre, Ahmadi and Islam (2019), Mahfouz, Venugopal and Shiva (2020).

Aghdam and Kabiri (2016) performed feature selection on the NSL-KDD and KDD Cup 1999 datasets using ant colony optimization, a special metaheuristic approach mimicking the foraging behavior of real-life ants. The authors mentioned no explicit classification algorithm, although ant colony optimization could be utilized as one, the means of which I studied in Brunner (2019) as well.

Almseidin *et al.* (2017) compared several models on the KDD Cup 1999 dataset, reporting random forest classifiers, an ensemble method, having the best overall performance in terms of precision, recall and AUC. Their work is not the only one which, either on purpose or by accident, compared single model approaches with model ensembles. The ensembles outperformed the single models in every case.

Bouzida *et al.* (2004) experimented with k-nearest neighbor and decision tree approaches augmented by principal component analysis on the 10% sample of the KDD Cup 1999 dataset. They reported good classification performance on as low as four components for

both models. They also reported predictions on R2L and U2R classes to be the most difficult.

Elhag *et al.* (2015) proposed a genetic fuzzy system for classification in one versus one pairwise classification models rained on the 10% sample of the KDD Cup 1999 dataset. Their goal with it was to improve prediction performance on minority classes. The method they used was initially designed for association rule mining, but they extended it for classification. During training, just like Tavallaee *et al.* (2009), they removed duplicate observations. Performance evaluation shown comparable results to other fuzzy rule generation algorithms and to decision trees. Their proposed genetic fuzzy system performed well with underrepresented classes as well, while maintaining low false alarm rates.

Hasan *et al.* (2016) studied the intrusion detection performance of SVM classifiers under different kernels. They found the Laplace kernel to provide the best performance on the NSL-KDD dataset, though they highlighted that SVM model performance is dependent on the used dataset.

Ingre, Yadav and Soni (2017) used correlation-based feature selection and CART decision trees to perform predictions on the NSL-KDD dataset. They reported good classification performance on both binary and 5-class classification.

Mahfouz, Venugopal and Shiva (2020) trained naïve Bayes, logistic regression, neural network, SVM, k-nearest neighbor and decision tree models in three setups on the NSL-KDD dataset. The first and second setups were performed with and without feature selection. The third setup involved data resampling: random under sampling was used for majority classes, and SMOTE oversampling for minority classes. Models trained in setup three provided the best predictions.

Parampottupadam and Moldovann (2018) used the H2O.ai implementation of artificial neural networks on a cloud architecture. Their model performed binary classification between normal and attack traffic on the NSL-KDD dataset, then a second neural network classified the attacks into multiple classes. The authors compared the performance of their neural network architecture with SVM, random forest, linear regression and naïve Bayes models. Overall, the proposed neural network architecture provided the best classification performance.

Petersen (2015) used the NSL-KDD dataset and four machine learning algorithms (ID3 and CART decision trees, k-nearest neighbor and naïve Bayes) to perform three hypothetical experiments with binary, five class and 22-class classification schemes. A secondary analysis evaluated feature importance. The results provided shown that k-nearest neighbor and ID3 decision tree algorithms had the best overall prediction performance. Out of the classification schemes, binary classification models performed better, however, Petersen (2015) noted, that a case for a five class classification could be created, as it provides additional clues for the intrusion detection system to act on.

So-In *et al.* (2014) manually extended the KDD Cup 1999 dataset with a new class based on botnet signatures. Their model comparison covered decision trees, sequential rule construction, artificial neural networks, naïve Bayes, k-nearest neighbor algorithms and SVMs in different setups. Setup one involved binary classification between normal and attack traffic, setup two was 5-class multiclass attack detection and scenario three was 6-class multiclass attack detection with the new botnet class. The authors reported good prediction performance, with the best performing models being decision trees, neural networks and k-nearest neighbor algorithms.

Yin *et al.* (2017) used recurrent neural networks on the NSL-KDD dataset. Their choice of approach is interesting, as the original KDD Cup 1999 dataset does not contain any feature that is explicitly temporal, only implicitly temporal features in the form of time-based traffic features. This lack of a temporal feature has been inherited by the NSL-KDD dataset and it makes sorting observations difficult be used for training RNN models, despite the potential gains of detecting attacks tied not only to a single traffic packet, but also to a flow of traffic.

Divekar *et al.* (2018) used naïve Bayes, SVM, decision trees, random forests, neural networks and k-means clustering with majority voting over the clusters. The authors applied synthetic sampling and improved the models with grid search hyperparameter optimization as well. They reported model performances in terms of F_1-score, where random forests performed slightly better than other, single-model classifiers.

Sapre, Ahmadi and Islam (2019) Studied naïve Bayes, SVM, random forest, and neural network models for binary and multiclass classification. They reported artificial neural networks as the best, outperforming even random forests in some classes and setups.

Sakr, Tawfeeq and El-Sisi, (2019) combined binary-, and standard-based particle swarm optimizations (BPSO and SPSO) with support vector machines. First, feature selection was performed by BPSO, then a support vector machine was trained on the NSL-KDD dataset. SPSO played a part in the optimization of the SVM model, which managed to achieve good performance.

A second group of researchers studied ensemble models with the intent of increasing overall intrusion detection performance by aggregating the results of multiple classifiers. Papers written on ensemble modeling are Chebrolu, Abraham and Thomas (2005), Folino, Pizzuti and Spezzano (2005), Mukkamala, Sung and Abraham (2005), Abadeh *et al.* (2007), Tian, Liu and Xiang (2009), Kevric, Jukic and Subasi (2017), Latah and Toker (2018) and Cavusoglu (2019). In the majority of these papers result combination has been based on a simple function of predictions, such as simple majority vote, average vote, rule-based evaluation, etc. The most common ensemble model used was the random forest algorithm due to its popularity. More complex boosting and stacking approaches were studied less in the intrusion detection literature.

Abadeh *et al.* (2007) presented a parallelized fuzzy rule generation approach, each rule built using a genetic local search algorithm. Each set of fuzzy rules were later aggregated to perform ensemble classification. This approach was compared with other rule-based learning algorithms, where it achieved best performance.

Tian, Liu and Xiang (2009) created a distributed learning model using artificial neural networks in a two-staged approach: in the first stage, a network learned a random subset of the KDD Cup 1999 10% dataset's features. In the second phase, the class predictions of these models were collected by a final classifier improving prediction performance. Conceptually, this model is the most similar to a combination of ideas used for random forests, neural networks and stacking model ensembles.

Chebrolu, Abraham and Thomas (2005) used a three-phased approach. They first performed feature selection on a sample created from the 10% sample of the KDD Cup 1999 dataset. In the second phase, they created a Bayesian network and a CART decision tree, tested separately. Later, the two models were combined into a bagging classifier with improved overall detection performance compared to each base model.

Folino, Pizzuti and Spezzano (2005) used distributed parallel genetic programming to train decision trees. These trees were then combined in an ensemble by using simple

majority vote. The solution was tested using the 10% sample of KDD Cup 1999 dataset. The proposed model performed well on normal, DoS and proba attacks, but struggled on minority classes.

Latah and Toker (2018) experimented with decision trees, random forests, bagging trees, multiple boosting algorithms, k-nearest neighbor algorithms, extreme learning machines, neural networks, SVMs, linear discriminant analyses and naïve Bayes algorithms. Some of these are single-model methods, others ensemble models. The authors achieved the best performance on LogitBoost out of the listed detectors.

Mukkamala, Sung and Abraham (2005) discussed three different artificial neural networks (different in their optimization algorithms), SVMs and multiple adaptive regression spline (MARS) models. These results were improved further on when the authors aggregated the results with majority voting. The authors created two stacking models: one that combined the three ANNs, and a second adding the SVM and MARS models to the stack. The best performing ANN was the one with back propagation, though both stacking models improved on the results further.

Cavusoglu (2019) used naïve Bayes, random forest, decision tree and k-nearest neighbor algorithms as single-model classifiers and as candidate base models for stacking classifiers combined with logistic regression. The author grouped the NSL-KDD data into multiple samples comparing each attack class to normal traffic. Each model was trained and evaluated on these with the option of formulating ensembles as well. Though not a conscious attempt at studying ensemble models, the resulting detectors of Cavusoglu (2019) all ended up being random forests or stacking classifiers.

Kevric, Jukic and Subasi (2017) compared different models based on decision trees, then combined them into majority voting ensembles. One example is the NBtree model, which is a specialized decision tree with naïve Bayes classifiers at each leaf of the tree.

More sophisticated models combine signature and anomaly detection, resulting in hybrid intrusion detectors. Only a few research papers evaluated hybrid detection. These had shown a lot of variations on how models can be combined. Studies evaluating hybrid detection were Zhang and Zulkernine (2006), Zhang, Zulkernine and Haque (2008), Kim, Lee and Kim (2014), Parsaei, Rostami and Javidan (2016) and Yao *et al.* (2017).

Zhang and Zulkernine (2006) and Zhang, Zulkernine and Haque (2008) demonstrated the applicability of random forest algorithms for hybrid intrusion detection. In their papers, signature detection is performed using random forests, and anomaly detection with outlier detection techniques applied on each leaf of every decision tree in the forest. For example, outlier detection can be based on a similarity score between two network traffic observations which appear in the same leaf for a large enough number of trees. The authors profiled the records for outlier detection not by attack class, but by network service, which is available in KDD Cup 1999.

Similarly, Kim, Lee and Kim (2014) used decision trees and one-class SVMs for their hybrid intrusion detector. Their approach constructed a decision tree first to classify attacks present in the training dataset. Attacks unknown to the model were used to train one-class SVM models, one for each leaf of the decision tree having unknown attack classes. In this combination, the model managed to achieve good predictions with a low false positive rate.

Yao *et al.* (2017) proposed a new hybrid multi-level data mining system for intrusion detection. The system consists of three components. The multi-level hybrid data engineering component is tasked with data preprocessing and with splitting the data to one versus rest samples. Then performs feature selection on the samples. The second component is called multi-level hybrid machine learning, and it is responsible for model training by clustering each data group first, then classifying each cluster using either an SVM model, an artificial neural network, a decision tree or a random forest. These are not evaluated immediately, because the next component, micro expert modify generates "impurity data" from the misclassified traffic, then trains a new decision tree model on this misclassified dataset to improve predictions further. The hybrid multi-level data mining system achieved better performance using the KDD Cup 1999 10% sample than many non-ensemble and ensemble approach used before, even on the more challenging minority classes.

Parsaei, Rostami and Javidan (2016) focused their efforts on the minority classes of NSL-KDD. They used a combination of k-means and k-nearest neighbor algorithms. They first clustered the training data, and calculated two distances, one from the cluster centroids and one from the neighborhood of each traffic record. They used this aggregate feature for dimensionality reduction to a single explanatory feature. This feature was used

to train k-nearest neighbor algorithm. To increase performance, the authors used SMOTE sampling as well.

So far, I have excluded one type of hybrid intrusion detection from the previous paragraphs due to their unique nature. The following authors all combined autoencoder and variational autoencoder networks with signature detectors to achieve an even greater level of prediction performance: Javaid *et al.* (2016), Al-Qatf *et al.* (2018), Lopez-Martin, Carro and Sanchez-Esguevillas (2019) and Yang *et al.* (2019). The utility of autoencoders comes from how they can be viewed as dimensionality reduction algorithms. On top of that, VAEs and CVAEs also perform well as data generative models, therefore, they can replace synthetic sampling techniques, as well as help machine learning models acquire more knowledge on minority classes.

Al-Qatf *et al.* (2018) combined sparse autoencoders with SVM classifiers. This has been achieved by training the SAE on unlabeled data to generate a low dimensional representation. Following this, new data with target labels are fed to the encoder layers only. The reduced dimension explanatory features are then fed to the SVM classifier. The authors did not only report improved performance, but also improved the memory footprint and lowered training time for the SVM model. Similarly, Javaid *et al.* (2016), combined an autoencoder with multiclass logistic regression. Both reported classification performance greater than ensemble models.

Lopez-Martin, Carro and Sanchez-Esguevillas (2019) used different types of VAE models. The first model was a standard VAE conditioned by target labels at encoder input. It used cross-entropy loss regularized by KL divergence compared to standard normal distribution. The second variation split explanatory features at the output layer to numerical and categorical. The loss for numerical features was MSE, whereas the loss for categorical features remained the cross-entropy loss. KL divergence was not changed. The third model changed the conditioning: instead of the encoder input, it was applied on the decoder input. Out of the three models, the outputs of the third provided the best predictions, outperforming those using synthetic sampling. For their performance tests the authors used random forests, linear SVMs, logistic regression and neural networks, although their main focus was on sampling, rather than on prediction capabilities.

Yang *et al.* (2019) combined improved CVAEs with neural networks. The improvement in their model was a target conditioning applied on the decoder layer only, which makes

their intrusion detector similar to the third model of Lopez-Martin, Carro and Sanchez-Esguevillas (2019). The main difference was that Yang *et al.* (2019) re-used the encoder for weight initialization in the detector neural network. They carried out numerous performance comparisons with single-model, ensemble and other AE supported hybrid models as well, reporting their CVAE + NN model achieving the highest detection performance.

I also identified two techniques that could increase detection performance regardless of the model type used. One was synthetic sampling and the other was hyperparameter optimization, both used infrequently in the articles I researched. Synthetic sampling has been utilized by Parsaei, Rostami and Javidan (2016), Divekar *et al.* (2018), Lopez-Martin, Carro and Sanchez-Esguevillas (2019), Yang *et al.* (2019) and Mahfouz, Venugopal and Shiva (2020). These papers compared SMOTE with their respective VAE variations or used to improve detections of their model. As SMOTE and VAE fulfill the same purpose, it is highly discouraged to use them at the same time. Hyperparameter optimization was used by Zhang, Zulkernine and Haque (2008), Hasan *et al.* (2016), Yin *et al.* (2017), Al-Qatf *et al.* (2018), Divekar *et al.* (2018), Sakr, Tawfeeq and El-Sisi (2019) and Yang *et al.* (2019). The most used optimization strategy was grid search.

Based on my review of the literature, I experienced a hierarchy between the studied techniques, starting from single-model signature / anomaly detection, followed by ensemble models, then by hybrid models, and finally, new data generative approaches, like VAE models.

Each review article I presented earlier in this chapter provided challenges and open questions in intrusion detection. Bhuyan, Bhattacharyya and Kalita (2014) brought up the following issues, questions and research topics:

- The nature of attacks keeps changing over time; therefore, adaptability of models is a necessity.
- A high rate of false alarms should be avoided; however, it cannot be eliminated completely.
- There is an overarching need for benchmark intrusion datasets.
- A fast and appropriate feature selection for all attack classes is needed.
- Selection of non-correlated classifiers for building an effective ensemble approach.

Buczak and Guven (2015) advised the following criteria to compare machine learning algorithms with each other:

- Performance measures do not work for comparison, as the trained machine learning algorithms were tested using different samples of the same dataset.
- Due to the ever-changing nature of network attacks, intrusion detectors need to adapt quickly. IDS model training, however, is performed when traffic is the lowest, usually at night. It is expected form the training process to not take 24 hours. A relatively low training time therefore is key to evaluation.
- Intrusions should be detected fast. Quick classification of network traffic can improve reaction time and shows the processing capability of the system.
- To help administrators examine model characteristics and update the system more easily, a model with lower complexity is preferred, though not mandated.

Buczak and Guven (2015) furthermore gave the following advice on creating machine learning models for intrusion detection:

- Intrusion detection is a field with a rapidly changing environment. Models must be trained on a daily basis, or when a new intrusion is discovered. To adopt faster, the whole model should not be retrained again, but incrementally as the administrators feed it with new data.
- The KDD Cup 1999 dataset, as a benchmark, is widely accepted and used, however it has its own flaws. It contains too many redundant observations and the target class is unevenly distributed. Many tried to combat both by sampling the dataset, which makes performance comparisons complicated. The creators of the NSL-KDD dataset addressed this redundancy; therefore, it is a preferable alternative.

Dua and Du (2016) identified multiple challenges for data mining algorithms in intrusion detection:

- Modeling large-scale networks and creating graphs based on large networks is a difficult task.
- The volume of heterogenous data, the dynamic threats, and the severe imbalance between normal and attack classes complicate threat detection.

- New data mining methods and adaptive systems are necessary to predict future attacks.
- Use of online learning methods for dynamic modelling of network data.
- Modelling data with skewed class distributions to handle rare event detection. There is a fundamental asymmetry in anomaly detection problems between normal activities and attacks. Classification should be more focused on classifying minority classes as attacks or anomalies.
- One of the biggest challenges in anomaly detection is the selection of features that best characterize the user, or the system usage patterns. This is often carried out to reduce data dimensionality.

Molina-Coronado *et al.* (2020) provided the following open issues in their review article:

- Most papers provided insufficient information on the techniques applied for intrusion detection which hurt reproducibility.
- They found issues with publicly available datasets, too. A large portion of data preprocessing has been carried out in them in advance. The authors recommend not to rely on a single benchmark dataset but to use two or more instead. Furthermore, encrypted data is increasingly prevalent, which is not present in these public datasets at all.
- They highlighted the importance of dimensionality reduction, due to the large number of features, which is further increased when categorical features are encoded.
- Instead of batch learning, incremental learning should receive more attention in the future.
- The temporal nature of network traffic is underused, despite having a lot of potential.
- Intrusion detection has many more characteristics apart from detection performance. However, only the latter is studied in the field.

To summarize, I identified the following areas in need of substantial attention:

- Design hybrid detection approaches and/or ensemble models for comprehensive, unbiased intrusion detections.
- Mind the data: if the KDD Cup 1999 dataset is used, then an appropriate sample, and a good set of features should be selected. With NSL-KDD dataset, sampling

can be omitted, but some form of feature selection should be performed, nonetheless.

- When measuring performance, false alarm rate and recall are more important than accuracy.

3. RESEARCH OVERVIEW

This chapter provides an overview of the research I have conducted, demonstrated with the tools, techniques and considerations of the design science methodology and the CRISP-DM process. These two have many intersections, as some steps in the CRISP-DM process supports design science activities.

3.1. CONTEXT

The context of intrusion detection, apart from the details discussed in chapter 2.1, were elaborated in Ahamad *et al.* (2009). They identified five reasons for developing intrusion detection systems:

- **Threats from malware:** hackers use malware to steal private information. They leverage the vulnerabilities of web site structures, social networks and document transmissions not scanning for malware. Once and intrusion is successful, the malware will track the user's keystrokes, spy on the users browsing habits and send the user's personal information to the attacker.
- **Threats from botnets:** botnets are groups of hijacked machines coordinated by attackers. Bots in a botnet are controlled by a hidden master computer. Computer and internet users suffer privacy breaches or financial losses, loss of valuable data, and damage to computer systems caused by botnets.
- **Threats from cyber warfare:** cyber-attacks are critical military actions. The increasing dependence of traditional infrastructure on cyberinfrastructure leaves many vulnerabilities for cyber warriors to exploit. Cyber defense is an inevitable, challenging goal of military forces around the world. An efficient cyber defense requires conscious effort from multiple countries, states, institutions and industry members, as attacks can affect all of them.
- **Threats from mobile communication:** the development of mobile communication caused the proliferation of reliable services. Investigations shown

that even financial transactions appeared in mobile services, which draws the interest of hackers as well. The mobile infrastructure and devices provide multiple opportunities to steal valuable information. Institutions are developing new ways to protect against fraud and phishing.

- **Cyber-crimes:** different jurisdictions define cyber-crime depending on how it correlates to local situations. Prospering e-commerce entices cyber criminals, many purchase attack platforms to carry out their activities. These are carried out by exploiting vulnerabilities in the e-commerce industries. Countering these activities is difficult as they do not leave traces behind. Combating cyber-crimes requires effort in two perspectives: first, uniform cyber laws need to be enacted. Second, advanced intrusion detection technology needs to be developed to defend against criminal activities.

More recent developments within the context of intrusion detection are the ongoing monitoring and reporting on the development of malicious activities. One example is the McAfee Labs Threats report (Beek *et al.* (2019)). This report drawn attention to the increase of ransomware attacks, the increase of data dumps (release of sensitive customer data to the dark web), the increase of cyber-attacks exploiting vulnerabilities in remote desktop applications and in the HTTP protocol. Two attacks mentioned in the report were social engineering, which is still as prevalent as ever, and an increase in attacks exploiting the vulnerabilities of IoT devices. Many of these are not necessarily network intrusions themselves, more the results of a successful intrusion.

All the above and more fuel the efforts aimed at creating new and better intrusion detection systems. The goals of actors in the social context can be summarized in the following points:

- Risk mitigation: reduce the chance of intrusion, information loss, or fines in the form of potential lawsuits. Reduce system downtime due to DDoS attacks, by installing a traffic reduction service supported by an intelligent intrusion detector.
- Infrastructure and national security: prevent the sabotage of key infrastructural elements, such as electricity and water supply, increasingly reliant on information infrastructure.
- Protection of private information: restrict access to sensitive information, such as credit card numbers, bank account and personal information.

- Protection of government secrets: as an extension of the point above, departments store information not meant for a public audience. The exposure of these can have far-reaching consequences.

I have already discussed the knowledge context of intrusion detection in chapter 2. Out of them I found signature and hybrid NIDS interesting to be studied deeper using ensemble techniques and artificial neural networks. To frame my research, I decided to use the CRISP-DM process model, displayed in Figure 3.

3.2. RESEARCH GOALS

The goal and design problem of this dissertation is to provide a novel intrusion detection solution applying machine learning methods. Accordingly, the two research goals I set to achieve are:

RG1. To create an intrusion detection model that can compete with the ones introduced in related scientific literature, measured by detection performance metrics. Performance in this context is described as the portion of attacks correctly and incorrectly classified as being part of normal activity and vice versa.

RG2. To identify machine learning methods that can improve performance on complex event detection problems where target features have a high degree of class imbalance. Intrusion detection fits this description, as the available data is heavily skewed towards the more common normal, rather than the rarer malicious activity. Some of these candidates are synthetic sampling to feed more balanced training data, hyperparameter optimization to find the overall best performing parameters for a machine leaning model, and ensemble techniques, creating composite models for improved predictions.

Based on these research goals I formulated the research questions of the next chapter.

3.3. RESEARCH QUESTIONS

RQ1. Is machine learning a suitable approach for intrusion detection? If machine learning is a proper technique for intrusion detection, which are the appropriate models?

Finding the right machine learning model is a challenging task. It is affected by the selected intrusion detection method (signature detection or anomaly detection) as well as the available dataset and the sampling method chosen for that dataset.

The most common and best working non-ensemble machine learning algorithms in intrusion detection are decision trees, artificial neural networks and k-nearest neighbor algorithms for signature detection. Each has drawbacks though:

- Decision trees are prone to overfitting, unstable (a small change in training data can cause entirely different decision trees) and perform poorly on unevenly distributed training classes.
- Artificial neural networks, like decision trees, are prone to overfitting, and generally have long training times.
- K-nearest neighbor algorithms are fast to train, but need all data for accurate predictions, therefore they scale poorly.

Countless studies in the literature have proven that a good combination of machine learning algorithms can detect intrusions well with few false alarms.

Predictive performance is, however, not the only characteristic for intrusion detectors to be compared by. Training time, prediction time and model portability are three additional characteristics to consider. Under portability I mean how well can one move the detection model between two systems and how much computational resource do they require from the operator. However, I kept the evaluation of these aspects out of scope of this dissertation in favor of a more thorough study of predictions.

I answered this research question throughout the dissertation with different machine learning models, most prominently in chapters 4.2.1 and 4.2.2, where I provided the designs of two intrusion detectors and in chapters 5.1 and 5.2, where I described the achieved performances of the same detectors. In addition, the analysis of the related literature in chapter 2.3 already provided context for this research question.

RQ2. Which type of intrusion detection method is more effective from the following ones: misuse detection by classification, anomaly detection by outlier analysis or a combination of the previous ones?

This is a more recent question in the field of intrusion detection, also highlighted by Dua and Du (2016). On one hand, signature detection can have high recall and low false

positive rate, is easy to implement, and provides predictions quickly. However, it is incapable of detecting new, unknown attacks. On the other hand, anomaly detection aims at building a profile of normal traffic, and then detects anomalous or attack traffic based on the difference from this normal profile. Anomaly detection captures unknown attacks better; however, it is more difficult for it to set apart attacks and anomalous traffic, as the latter might include unusual, yet normal connections as well, highlighted in Ippoliti (2011, 2013), therefore, anomaly detection will have high false positive rates. In a good intrusion detector, recall is high and false positive rate is low. Signature and anomaly detectors use compensatory detection approaches; therefore, it is a good idea to combine them into new hybrid detectors.

A simple combination of the two techniques is not enough though, a more purposeful approach must be followed. For a hybrid detector to work, one must make two decisions:

- Find the best candidate algorithms for the individual signature and anomaly detector.
- Find a way to integrate the two detection approaches to achieve the best balance of recall and false positive rate.

Good candidates for hybridization are models that do not perform conflicting operations on the data, for example, decision trees and one class SVM models or any autoencoder combined with fully connected artificial neural networks. The choice of integration can be simplified to one of the four alternatives shown in Figure 1 as well.

The chapters intended to provide answers to this question are 4.2.4 and 5.4 where I design and evaluate a neural network stacking ensemble as a signature detector enhanced by deep autoencoder networks as an anomaly detector. Chapter 5.4 in particular evaluates the composite performance of the two models and the anomaly detection capabilities of the autoencoder.

RQ3. What is the level of model performance that can be expected in an intrusion detection task?

Based on reviewing the related literature, contemporary intrusion detection research is facing the following challenges:

- Predominant use of the accuracy measure for performance evaluation on data with unevenly distributed classes.

- Different articles created their own samples of a chosen dataset, making performance comparisons between them and the proposed models difficult, if not impossible.
- Focus mostly on signature detection, less on other techniques.
- Intrusion detection is always involved with detecting minority classes.

There is a high variation on possible model performance measurements. Therefore, I set up two criteria for selecting papers from the related literature to compare the proposed models with, in order to test the assumptions of this dissertation.

- Emphasis on recall / detection rate: although accuracy is the most common metric, it is inappropriate for performing detections on imbalanced data. A better alternative is recall. Throughout the dissertation I favored literature with recall as the model performance indicator compared to those with accuracy, though, due to how common it is, I could not ignore accuracy completely. Moreover, I had to take the alternative names of recall, like detection rate and sensitivity, into account as well. To make the search more difficult, some papers claimed to use detection rate, when in reality, the definition and provided formula fitted accuracy instead.
- Data sampling is the second source of complexity and prediction variance in the literature. Different samples result in different models with different performance measurements. Therefore, I attempted to look for papers that validated their model proposals with the complete test samples of the datasets they used. Similarly, I set up the intrusion detectors of this dissertation in the following way: I tested them on the complete test sample of the respective dataset, regardless of what data I used for training. This covered data preprocessing as well: transformations were performed on the test sample using calculations from the training data to avoid information leakage.

I used these requirements as filters on the research papers to be used in the final performance comparisons in chapters 5.5 and 5.6. Apart from that, I also aimed to test techniques like synthetic sampling, particularly with models demonstrated in chapter 4.2.3 and evaluated in 5.3; and advanced hyperparameter optimization with models in chapters 4.2.3, 4.2.4, 5.3 and 5.4 to achieve increased model prediction performance.

3.4. METHODOLOGY

The methodology I used for designing, executing and evaluating my models followed a top-down pattern shown in Figure 22. As the goals I set can be achieved by creating and evaluating an algorithmic artifact, I found design science research to be a fitting methodology. Furthermore, this algorithmic artifact is in fact a machine learning model, therefore the concepts and considerations of the CRISP-DM process model for planning, implementing and deploying machine learning models can be applied as well, forming the second methodological pillar. Finally, the designs in chapter 4 outline the exact process of model creation, with the necessary data preprocessing, training and evaluation steps involved, forming the lowest level of methodological abstraction.

Figure 22: The methodological abstraction levels followed in this dissertation. Source: own edit.

To further clarify the connection between design science research and the CRISP-DM process model, one must first evaluate the engineering cycle (Wieringa (2014)). The engineering cycle is a rational problem-solving process consisting of 5 tasks, each displayed in Figure 23, and described in detail together with the CRISP-DM tasks in Table 3.

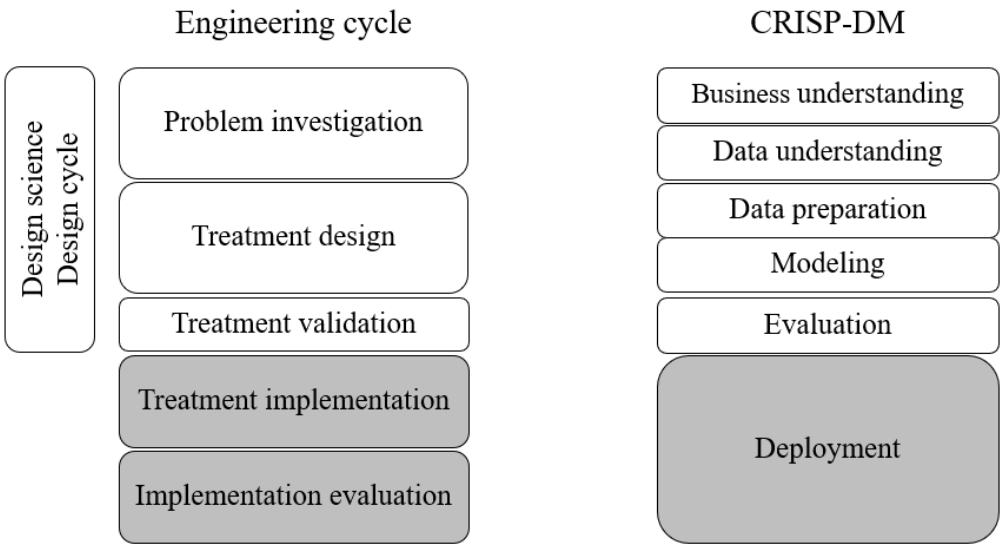

Figure 23: The relationship between the Engineering Cycle and CRISP-DM. Based on: Chapman *et al.* (2000) and Wieringa (2014)

Engineering cycle (design science)	CRISP-DM
Treatment: interaction between the artifact and the problem context.	In CRISP-DM, a treatment can be an implemented **machine learning model** and its effect on the decision-making process.
Problem investigation: to prepare the researcher for designing a treatment, by learning more about the problem to be treated.	**Business understanding**: to understand the business background / context / problem **Data understanding**: to evaluate the available data sources and to understand the meaning and utility of data for machine learning applications.
Treatment design: design is a decision about what the researcher is going to do. A specification is the documentation of this decision.	Multiple methods exist to express design and specification within the field of machine learning. **Data preparation**: ~ transform data for machine learning. In some contexts, this is referred to as data preprocessing. **Modeling**: find applicable model for the problem context, design model architecture, design model optimization process. **Evaluation**: the design of model evaluation is tied to this step (train/test split, CV, nested CV, choice of performance metric(s))
Treatment validation: the goal is to predict how a designed treatment will perform within context without it being observed in said context. As such, the evaluation is performed under artificial conditions.	**Modeling, evaluation**: executing the training and evaluation processes on separate training data.
Treatment implementation: implementation and use of treatment in the original problem context.	**Deployment**: live implementation of the machine learning model.
Implementation evaluation: evaluate how the implemented artifact interacts with its real context.	**Deployment**: performance monitoring of the machine learning model. Retrain in case of performance degradation.

Table 3: Comparison of engineering cycle and CRISP-DM tasks. Based on: Chapman *et al.* (2000) and Wieringa (2014)

The two methodologies are connected by their logically corresponding tasks, for example, problem investigation in the engineering cycle involves activities that are similar to activities performed during the business understanding and data understanding tasks of CRISP-DM. However, design cycle, the focus of design science, consists of only the first three tasks of the engineering cycle. Therefore, this dissertation will only discuss CRISP-DM tasks leading up to and including model evaluation. Deployment, although an important task, will only be discussed tangentially in chapter 6.

The two methodologies have differences as well. The goals of the two methodologies is one. The main goal of design science research is not only to deliver a well-designed, working artifact, but also to answer scientific questions about the artifact, at the context or at the relationship between the two. Comparatively, the goal of CRISP-DM is more practical. It is interested in delivering a machine learning algorithm, preferably as a part of a working business solution or service, delivering value to both the customers and the organization. The CRISP-DM approach therefore is more focused on evaluating the

business context and the effects on the business context, rather than on answering research questions.

Some personalization of the CRISP-DM process model, hence, will be necessary. These adjustments are not only permitted, but also encouraged by the designers of CRISP-DM, as they intended it to be a collection of best practices within the field of data science, rather than a rigid standard. Some of the changes compared to the CRISP-DM process model are:

- Greater emphasis on the wider context of the intrusion detection model: this includes both the social scientific context via literature reviews and the knowledge context by covering the data scientific tools and techniques in use. I covered them both already in chapter 2.
- More emphasis on model evaluation involving the comparison of model performances: On one hand, comparisons are conducted between the different of intrusion detection models I delivered as part of the design process. This supports the disclosure rule of the design science process as well. On the other hand, I compared the best performing detector to other works available in the field of intrusion detection, placing a higher emphasis on detection rate.

4. PROPOSED MODEL DESIGNS

This chapter describes the design and creation of the machine learning model-based intrusion detection architectures. In the chapter I introduce the datasets used for model training and evaluation first, followed by the detailed description of model architecture designs. Throughout this chapter and chapter 5, I followed the CRISP-DM process, creating four intrusion detection model variations.

4.1. INPUT DATASETS

After reviewing the literature, particularly Stolfo *et al.* (2000), McHugh (2000) and Tavallaee *et al.* (2009) the most common datasets for intrusion detection in use were: DARPA 1998 & DARPA 1999, KDD Cup 1999 and NSL-KDD. These datasets are all the products of an experiment conducted in 1998 by MIT Lincoln Labs to survey the state of the art in intrusion detection at the time. During the experiment, about 5 million records were collected in 5 weeks in the form of raw tcpdump logs. The data simulated the traffic

of a typical Air Force LAN, while the researchers carried out multiple network attacks against it.

The first iteration of these experiments were the DARPA 1998 & DARPA 1999 datasets. These were highly criticized, particularly by McHugh, (2000). These issues have already been discussed in chapter 2.3. The main criticisms were the unit of analysis problem, the question of attack distribution and the large level of redundancy among the records.

The unit of analysis criticism has been resolved by the KDD Cup 1999 dataset, by fixing the unit of analysis in network connections. The dataset itself consists of ~5 million network connection record for training, and another ~3 million record for testing intrusion detection models. The altogether ~8 million records might be too difficult for an intrusion detection system to handle; therefore, the authors of the KDD Cup 1999 dataset provided a 10% stratified sample of both the training and test datasets. The total number of features available is 41, with 40 explanatory and 1 target feature. Designed primarily for signature detection, the target feature contains numerous attack types each belonging to five distinct attack classes:

- **DoS**: denial of service attacks aimed at disabling crucial systems or system components.
- **R2L**: unauthorized access from a remote machine.
- **U2R**: unauthorized access to local superuser (~admin) privileges by a local unprivileged user.
- **Probe**: surveillance and probing, not attacks by themselves but could be used to prepare for future attacks.
- And **normal** legitimate behavior.

The assignment of each detailed attack type to their respective class is shown in Table 4. Some of the detailed types are only available in the test dataset of KDD Cup 1999. The training attack types were well documented by Stolfo *et al.* (2000); the test attack types, however, were not, which caused some confusion in the studied literature. I have determined a final detailed attack type to high level attack class assignment shown in Table 4 using a simple majority vote between relative class frequencies based on the assignment tables published in 10 different articles. The exact process of this is further demonstrated in Appendix A.

Class	Train	Test
Normal	normal	normal
DoS	back, land, neptune, pod, smurf, teardrop	apache2, mailbomb, processtable, udpstorm, worm
Probe	ipsweep, nmap, portsweep, satan	mscan, saint
R2L	ftp_write, guess_passwd, imap, multihop, phf, spy, warezclient, warezmaster	httptunnel, named, sendmail, snmpgetattack, snmpguess, xlock, xsnoop
U2R	buffer_overflow, loadmodule, perl, rootkit	ps, sqlattack, xterm

Table 4: Classification of attack types. Source: own edit (see Appendix A for details).

Compared to the previous DARPA 1998 and 1999 datasets, the features of KDD Cup 1999 are better organized and described, and, as a part of data preprocessing, new derived features were created based on domain knowledge by Stolfo *et al.* (2000). These features can be grouped into four categories:

- **Intrinsic features**: features describing all network connections, regardless of user intentions.
- **Content features**: capturing information on the content of each network connection.
- **Time-based traffic features**: features aggregating the connections that had the same destination host or service as the selected connection in the prior 2 seconds.
- **Host-based traffic features**: as a counterpart to time-based traffic features, host-based traffic features were created to capture aggregate data not over the prior 2 seconds, but over the previous 100 connections.

Tavallaee *et al.* (2009) identified an issue with the KDD Cup 1999 dataset: a large number of redundant observations (Table 5 and Table 6). About 75% of the test set and 78% of the training set is duplicated. This redundancy often caused research papers prior to 2009 to have biased intrusion detectors towards duplicate records. To alleviate this issue, Tavallaee *et al.* (2009) proposed the new NSL-KDD dataset. The authors provided two datasets: the first with binary labels and the second with 5-class labels, both having their respective training and test sets.

	Original records	Distinct records	Reduction rate	Final records
Attacks	3,925,650	262,178	93.32%	58,630
Normal	972,781	812,814	16.44%	67,343
Total	4,898,431	1,074,992	78.05%	125,973

Table 5: Statistics of redundant records in the KDD train set. Source: Tavallaee *et al.* (2009)

	Original records	Distinct records	Reduction rate	Final records
Attacks	250,436	29,378	88.26%	12,833
Normal	60,591	47,911	20.92%	9,711
Total	311,027	77,289	75.15%	22,544

Table 6: Statistics of redundant records in the KDD test set. Source: Tavallaee *et al.* (2009)

Apart from reducing the level of redundancy in both sets of KDD Cup 1999, Tavallaee *et al.* (2009) introduced two additional changes to the original DARPA 1998 data: first, they trained 21 machine learning classifiers on the reduced redundancy KDD Cup 1999 dataset. Each record has been grouped based on how many models predicted its class correctly. This information was then stored as a complexity feature in addition to the rest of the data and provided an input to the next change. In order to make the size of the dataset more manageable, a random sampling has been performed, stratified by class, and the new complexity feature. This resulted in the creation of the NSL-KDD dataset, with final class membership counts shown in the final records column of Table 5 and Table 6.

The final issue was the class distribution. Figure 24 shows this for the 10% sample of the KDD Cup 1999 dataset, while the same for the NSL-KDD training dataset is shown in Figure 25. There are too many records for DoS attacks, and not enough for the remaining classes. This distribution is unrealistic, a real-life environment can have a ratio closer to 98-95% to 2-5% between normal traffic and any attacks. Nonetheless, class imbalance persists, the only factor that has changed is the class in majority. Neither of the two studied datasets proposed solutions to handle class imbalance, finding them is up to the person conducting research. Inequalities in class distribution in general can be corrected by using one of following strategies recommended by Brownlee (2015):

- **Collect more data**. Because the research that produced the DARPA 1998 data has concluded a long time ago, this alternative is improbable.
- **Change the performance metric** from accuracy to something different, discussed in chapter 2.2.
- **Resample the dataset**: one can use oversampling on the less represented classes, and under sampling on the better represented ones. This serves no benefit by itself, as some of the minority classes have <100 observations. Therefore, even if the minority classes are 100% oversampled, their number is still insufficient when compared to majority classes, and if the majority classes were to be under sampled, then the size of the training data will be too small for any meaningful model to be trained.

- **Generate synthetic samples**: the idea behind synthetic sampling is to generate samples where the records are not necessarily from the original dataset but were created with some randomness involved, based on statistical distributions of the class they originate from. Some methods of generating synthetic samples are reversed Naïve Bayes algorithm, SMOTE, and more recently, variational autoencoders. This approach could work by itself, or as the second phase of a process aimed at creating a balanced training sample.
- **Try different algorithms**: use not just one data mining algorithm on a dataset but try out more and see which works best. This is the thought process behind model ensembles.
- **Use penalized models**: penalized classification imposes an additional cost factor to misclassification. In short, the cost of making a mistake is set to be higher for the minority class, compared to the majority class.
- **Use a different perspective**: view the dataset from the perspective of the area studying it. This usually involves different machine learning algorithms, for example, clustering or outlier analysis instead of classification. This is the idea behind anomaly and hybrid detection models.

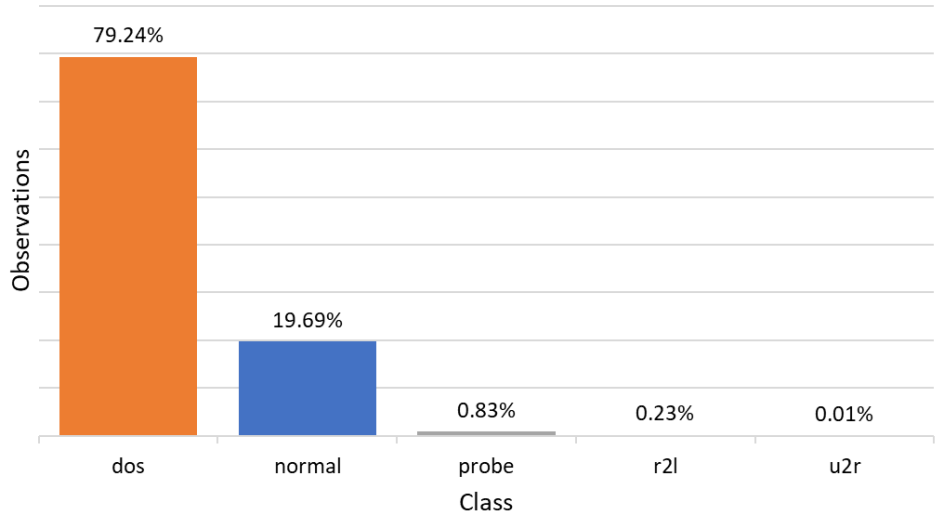

Figure 24: KDD Cup 1999 class distributions on the 10% training sample. Source: own edit.

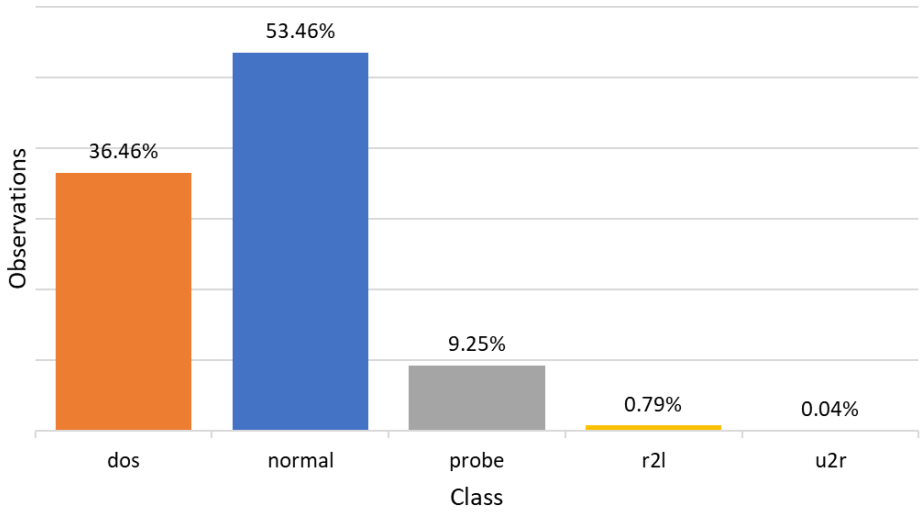

Figure 25: NSL-KDD train dataset class distributions. Source: own edit.

Despite the criticisms formulated, a large portion of the literature still use the KDD Cup 1999 and NSL-KDD datasets. Therefore, I decided to prefer the NSL-KDD dataset, in contrast to newer datasets. In return, these datasets have been evaluated many times before, and now work as benchmarks for intrusion detection models. Additionally, according to Stolfo *et al.* (2000), the core idea behind the KDD Cup 1999 dataset (and NSL-KDD dataset) is that training data contains one set of attack patterns, while test data contains a different set of attack patterns, some unavailable in the training data. These test attack patterns are impossible for machine learning models to learn, emulating the appearance of new attack types. This makes KDD Cup 1999, and NSL-KDD as an extension, conceptually similar to the newer intrusion detection datasets.

4.2. Model evolution

I studied intrusion detection models created by combining machine learning algorithms in an ensemble. The design and implementation of one model, however, was not an easy task due to the specifics of the dataset. I had to perform multiple iterations to find an appropriate model. I discuss further elements of the CRISP-DM process in terms of these iterations, where each produced a new, better refined version of an intrusion detector. Figure 26 shows the detection models created in this iterative process:

- **Version 0** (prototype): the first prototype of the model is outlined and evaluated in Brunner (2017), where I published a decision tree bagging classifier trained on

a map-reduce-like architecture. I trained this model only on the KDD Cup 1999 dataset.

- **Version 1** (neural network stacking ensemble): I created a stacking ensemble from neural networks trained on different features. I managed to improve performance by using a more robust sampling process and grid search hyperparameter optimization. In this model I transitioned between KDD Cup 1999 and NSL-KDD, sampling both differently.
- **Version 2** (migration to TensorFlow): I moved the neural network ensemble over to a TensorFlow + Keras platform achieving faster training. I expected further improvements in prediction performance by using TPE hyperparameter optimization. My second goal with this iteration was to evaluate different variations of SMOTE sampling, namely SMOTE ENN, SMOTE Tomek, and SVM SMOTE. In this iteration I used the NSL-KDD dataset only.
- **Version 3** (extension with autoencoders): where I extended the best performing elements of earlier iterations (like SVM SMOTE sampling and TPE optimization) with deep autoencoder networks trained on normal traffic, creating a true hybrid intrusion detection approach. For training, I kept the NSL-KDD dataset.

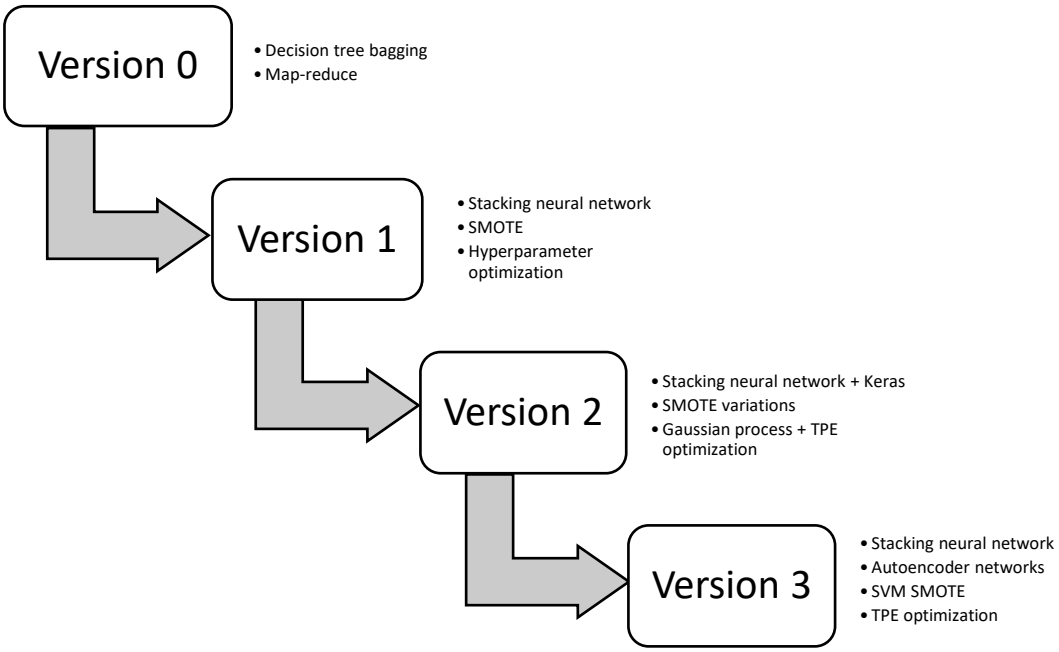

Figure 26: Iterations on the studied detection model. Source: own edit.

Further chapters show how the modeling and model evaluation steps of CRISP-DM were implemented throughout the different iterations of the proposed intrusion detector. I

describe implementation details by how data preparation and model training were performed, the plans for performance evaluation and techniques how I attempted to improve the design of the next model with.

4.2.1. THE DECISION TREE BAGGING MODEL

The first machine learning model used for intrusion detection was built using decision trees organized into a bagging ensemble on a parallel map-reduce environment. I discussed this model in detail in Brunner (2017). I used Java and the WEKA API to implement this machine learning model. In further chapters I will refer to this intrusion detector as V0 model, due to it being the first model I created. This is a naming I will follow consistently throughout the dissertation for the other intrusion detectors as well.

Data preparation

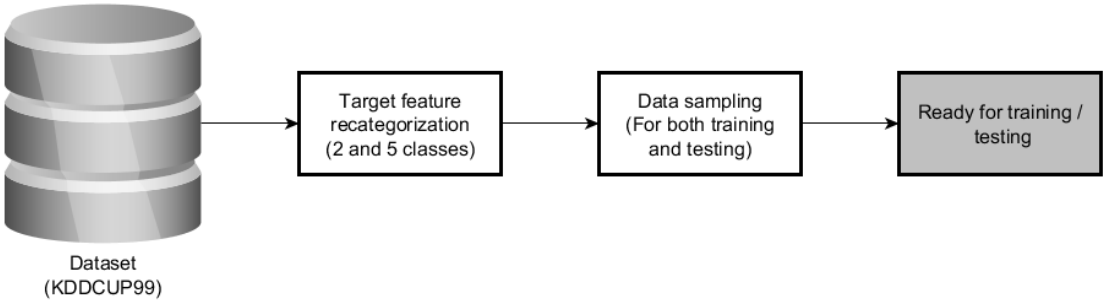

Figure 27: Data preprocessing for the detection model prototype. Source: own edit.

The steps of data preparation are outlined in Figure 27. I had to organize unique attacks into their respective classes first. To achieve the desired outcome I used an earlier conceptual hierarchy from which I created the categorization scheme in Table 4 and in Appendix A.

Next I performed stratified sampling on the 10% sample of the KDD Cup 1999 training dataset. Altogether I created 4 datasets with different target features iteratively changing the following settings:

- Target feature: during this first iteration, I performed binary and five-class classifications. I based the classes for multiclass classification on the early conceptual hierarchy. Binary classification was a choice between normal traffic and one of the four attack classes.

- Sample size: small or large. This, together with map-reduce parallelization had implications on training time only. The exact sample sizes are available in Table 7.
- Intent with the sample: I prepared a test and a training sample. However, due to frequent memory overflow errors of the Java platform, I had to swap training and test datasets around for the binary and multiclass classification tasks. Table 7 shows how I performed this exactly. Training and test columns show the number of observations available in a given sample.

Target variable	Training	Test	Sample size
5 classes	3,000	5,000	S
2 classes	5,000	3,000	
5 classes	6,000	10,000	L
2 classes	10,000	6,000	

Table 7: Sampling setup of the prototype intrusion detector. Source: Brunner (2017)

Apart from target recategorization and data sampling, I performed no feature selection or feature grouping. Moreover, I transformed no numerical or categorical features either.

Modeling

I set up the model to work in a parallel map-reduce environment in three different architectures (Figure 28) each different in the number of CPUs and CPU cores used: 1 and 2 CPUs and 2-4-8 cores. Out of these architectures my goal with the 1 processor, 2 cores architecture was to train a benchmark classifier, to provide simple results for comparison with the later ensemble models trained on 4 and 8 processing cores.

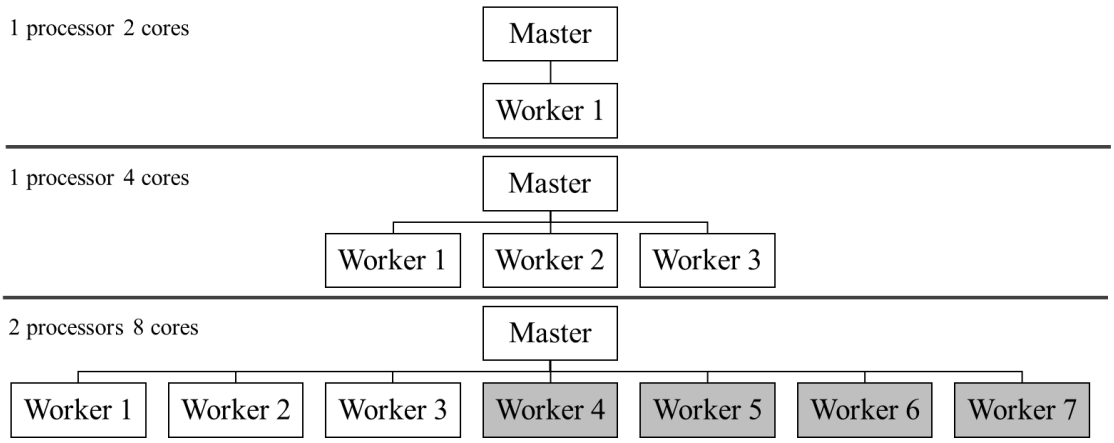

Figure 28: Experimental execution architectures of the V0 intrusion detector. Source: own edit.

The first available core was reserved for a master thread tasked to distribute the stratified subsamples to all the remaining threads, each training a decision tree (Figure 29). When

done, each thread calculated predictions on the test sample. These were sent back to the master, where the final class of each observation was decided based on a majority vote between the decision tree predictions.

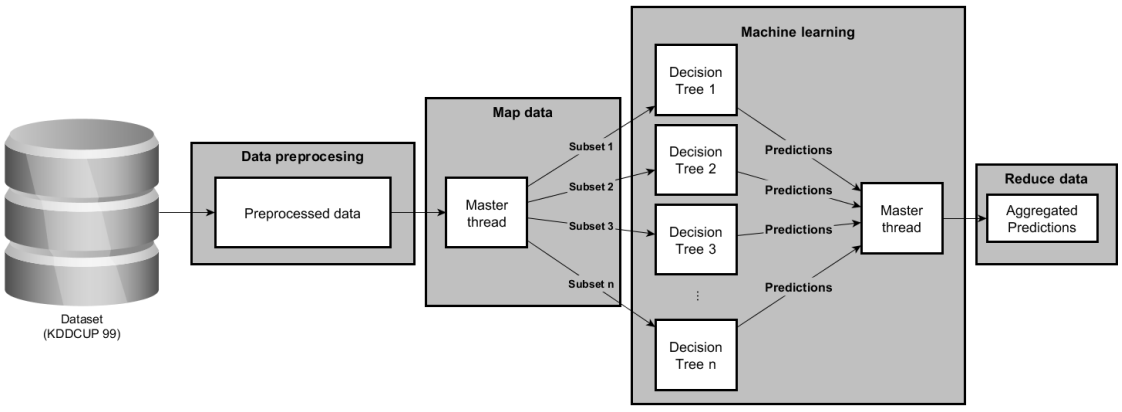

Figure 29: The model creation and prediction process of the V0 intrusion detector. Source: own edit

To mitigate the level of uncertainty caused by random elements of the process, I repeated training three times for each sample size (small or large), architecture setup (1 processor 4 cores, 2 processors, 8 cores) and target feature kind (binary and five-class) combination. Altogether, including the benchmarks, I repeated the training and testing processes 28 times.

Evaluation

Performance measurement and data collection were determined by target feature kind. I collected accuracies and macro-averaged precisions, recalls and F_1-scores for five-class classification. I expanded these metrics with ROC AUC scores for binary classification. Due to my lacking understanding of model ensembles at the time, I only collected base classifier performance metrics, but no metrics measuring aggregate classification performance. Moreover, in some instances the base classifiers were unable to detect minority attack types, thus I had to set precision, recall and F_1-score values to zero. This is a common behavior in many programming frameworks for machine learning. I had to follow this in a manual post-processing step, as the WEKA API at the time did not support it.

Due to the parallelization on the map-reduce architecture, I measured execution time as well, although I dropped this in later models, so I could focus more on detection performance.

Potential improvements of the model

This first version is best described as a prototype intrusion detector. It had many flaws:

- I only measured the classification performance for each individual base model, the aggregate performance of the ensemble could only be evaluated as the arithmetic mean of each base classifier, which does not reflect the real classification capabilities of a bagging ensemble.
- Java and the WEKA API, though useful on their own right, have counterparts that are better at performing data preprocessing, model training and testing. Three examples are Python, R and the KNIME Analytics Platform. The first two are programming languages less susceptible to malfunctions and are easier to maintain. Additionally, Python has readability advantage over most other programming languages as well. KNIME Analytics Platform is a free environment for developing and maintaining data workflows. Written in Java and originating from the WEKA API, it is an ideal choice for someone who prefers using the two.
- More robust sampling methods are to be explored, having a large effect on model performance.
- New machine learning models were recommended for use in detection models, particularly artificial neural networks.
- Opportunities related to feature group creation were not explored.
- Out of binary and five-class classification, only the latter should be kept, being greater challenge. This is supported by the nature of network intrusions as well, after all, different mitigation controls should be applied to DoS attacks than to R2L or U2R attacks.

However, some findings of this early version are undeniably valuable. For example, the application of model ensembles was a forward-looking idea. With all the above considered, I designed the next experiment.

4.2.2. THE STACKED NEURAL NETWORK MODEL

The next intrusion detection model has been implemented using the Python scientific stack (a collection of Python modules designed for data manipulation and data scientific tasks, the core modules being pandas + numpy + scikit-learn + matplotlib). I created a new stacking ensemble of artificial neural networks and evaluated it for detection performance. In the following chapters I will refer to this model as V1.

Data Preparation

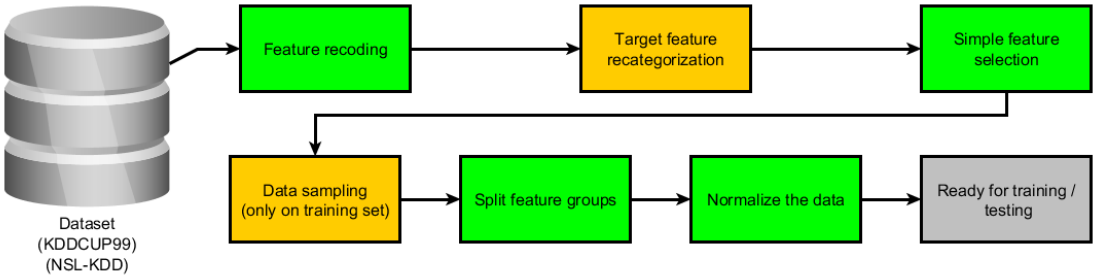

Figure 30: Data preprocessing for the V1 detector. Source: own edit.

Figure 30 shows the modifications to data preprocessing performed on the 10% sample of the KDD Cup 1999 dataset first, then later on the NSL-KDD dataset. I highlighted the new steps compared to the preprocessing of the V0 model in green, and the two altered steps in orange:

- Some categorical features were recognized incorrectly as numerical by the Python interpreter, I corrected these in the first preprocessing step. Furthermore, as the second part of this process, I encoded all categorical features using one-hot encoding to be more appropriate for processing by the neural networks.
- The target feature was created using the class assignments described in Table 4. I dropped binary classification in this iteration, however.
- I performed a simple feature selection to remove explanatory features with no variance (equivalent to not having information content). I based this feature selection on relative deviation.
- I fundamentally redesigned sampling generate balanced samples more efficiently. This process was different for the two datasets. For the 10% sample of KDD Cup 1999, it was performed in two stages. Stage one performed a balancing stratified

split, where minority classes had higher probability to be selected in the sample, the exact sampling fractions are in Table 8. The second stage balanced the sample further by performing SMOTE sampling on the intermediate sample. This two-stage approach yielded a completely balanced sample. For the NSL-KDD dataset, due to its more manageable size, I only used SMOTE.

Class	Normal	DoS	Probe	R2L	U2R
Sampling fraction	2.50%	0.50%	50%	100%	100%

Table 8: Sample fractions to balance class distributions in the 10% KDD Cup 1999 sample before SMOTE resampling. Source: own edit

- I split the data to feature groups according to the findings of Stolfo *et al.* (2000). The sample was grouped into intrinsic, content, time-based traffic and host-based traffic feature groups.

- As the last step of data preprocessing, I normalized the training sample with min-max normalization for the neural networks to reach meaningful results.

One might ask whether the sample created from the 10% KDD Cup 1999 data is a valid representation of the original. I have validated this in a separate experiment where I repeated the proposed sampling process 150 times, then compared them to the original training dataset. I based this evaluation on the nonparametric two-sample Kolmogorov-Smirnov test from statistics. The null hypothesis of the K-S test states that the two samples were drawn from the same statistical distribution. These K-S tests were then performed for each class, feature and sampling iteration. The result is a per class aggregation of the acceptance or rejection of the null hypothesis, where acceptance counts as 1 and rejection as 0. My goal with this test was to provide insights into how well the sampling matched the original data.

For the NSL-KDD data, as SMOTE is guaranteed to yield a synthetic sample with a distribution matching the original data closely, answering the above question has no additional benefit.

Modeling

I trained multiple neural networks, one for each feature group and one as a final aggregator model. The modeling setup is visible in Figure 31.

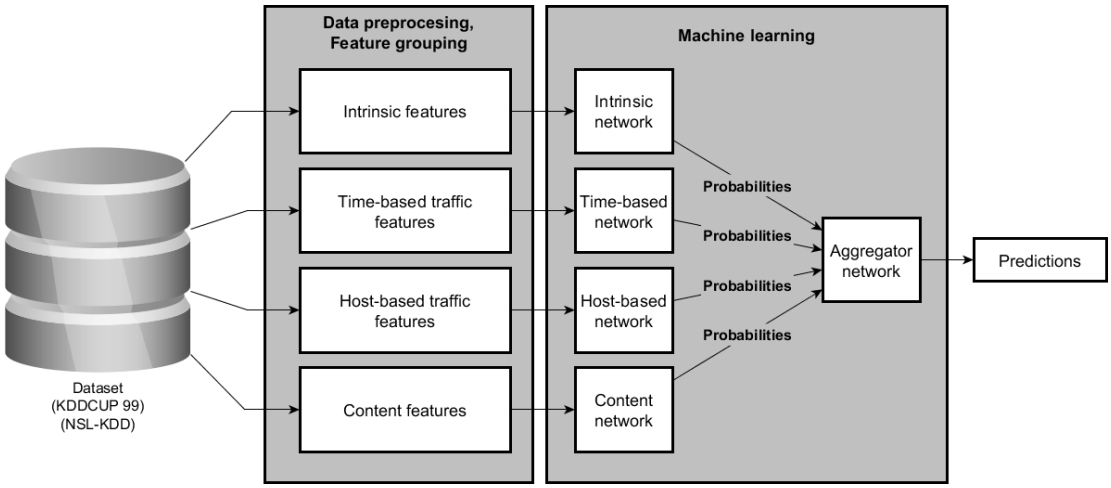

Figure 31: The model creation and prediction process of the V1 model. Source: own edit

An important element to stacking ensembles is the variance of the base models, which is usually achieved by different base models. In V1, I tried to achieve variation by feeding different data features to the base classifiers instead.

I trained every model in the ensemble in a similar process. First, I optimized each with grid search hyperparameter optimization with five-fold CV for more stable results. The target metric I optimized for was recall, the hyperparameters I changed are visible in Table 9. The optimization process altered only the initial learning rate, the exponent for the decaying learning rate and the momentum for every neural network. Further settings available were, for example, hidden layer and neuron per hidden layer counts. In those cases however, grid search would have taken too long to conclude and yield an optimal neural network architecture. As a compromise, I gave each model a fixed architecture. The base classifiers were trained on three hidden layers with 40, 20 and 10 neurons respectively, whereas the aggregator was trained only on two hidden layers with 10 and 5 neurons. I base my argument for the smaller architecture for the aggregator model on that it received only $k \cdot 5$ features as input, one for each target class value per base classifier.

Parameter	**Base models**	**Aggregator model**
hidden layer	(40, 20, 10)	(10, 5)
activation	RELU	
solver	Adam	
alpha (L2 regularization)	0.0001	

Parameter	Base models	Aggregator model
learning rate type	inverse scaling	
initial learning rate	0.01, 0.03, 0.05, 0.07, 0.1, 0.15, 0.3, 0.5, 0.7, 0.9	
LR decay power	0.25, 0.5, 0.6, 0.7, 0.8, 0.9	
momentum	0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9	

Table 9: Hyperparameter settings for the V1 detector. Source: own edit

Evaluation

I used the standard measures of accuracy, precision, recall and F_1-score for performance evaluation, obtained from testing the ensemble with the dedicated test datasets of KDD Cup 1999 and NSL-KDD respectively. I performed transformations using the same one-hot encoding, feature group splitting rule and normalizer objects I fitted using the training datasets to limit the effect of information leakage. Moreover, I performed no sampling on the test datasets either.

Potential improvements of the model

This iteration has taken a major step forward in terms of quality and classification performance compared to the prototype V0 model. However, I identified new issues as well:

- Although I did not measure with research intent, the training process with grid search hyperparameter optimization took a significant amount of time, which was a result of multiple factors: the notoriously long training time of neural networks, the grid search algorithm itself and the cross-validation iterations. I found TensorFlow + Keras with GPU acceleration capabilities a good candidate to improve this training time, with the potential benefit of improving model performance further.

- I performed hyperparameter optimization using grid search, though I considered random search at some point as well. Both have flaws, grid search takes a long time, while random search is not guaranteed to find global optimum. Gaussian and tree-structured parzen estimator hyperparameter optimization both evaluate a small number of combinations, but they do it more intelligently, thus converge faster to global optima. Moreover, they can search in larger parameter spaces, therefore more parameter dimensions could be evaluated.

- Later in the iteration, as I evaluated synthetic sampling for utility, I discovered multiple modifications to the SMOTE sampling algorithm. Some of these variations had the potential to further improve prediction performance, therefore I found it useful to include them in the next iteration.

4.2.3. NEURAL NETWORKS ON TENSORFLOW AND KERAS

The next iteration was a natural evolution of the V1 model. I created it by implementing two major changes to the training process shown in chapter 4.2.2: first, I changed the models from scikit-learn MLPClassifiers to Keras models on TensorFlow backend. Second, I used TPE hyperparameter optimization instead of grid search during model training. I refer to models created this way as V2 detectors, with variations in naming suggesting variations in the sampling I applied on data the models were trained with.

According to the Google Brain Team (2015) *"TensorFlow is an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researchers push the state-of-the-art in ML and developers easily build and deploy ML powered applications"*. Developed by Google, it has quickly obtained popularity in the fields of machine learning and AI research.

The latest release of TensorFlow has native support for Keras, a high-level API for neural network architecture development. According to the Keras documentation (Chollet (2015)): "*Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation*".

Recently, Google released TensorFlow 2.0, which tied the Keras API closer, which, at the same time discontinued support for Theano backend. To keep my models up to date, I used TensorFlow 2.0 as well.

Data Preparation

The data preparation step in Figure 32, as the majority of my changes affected modeling, remained largely unchanged compared to the previous iteration (shown in Figure 30), except for two. First, I experimented with multiple variations of synthetic sampling, namely SMOTE ENN, SMOTE Tomek and SVM SMOTE, the former two being

combinational over and under sampling approaches, while the latter being a strictly over sampling approach. My second change was added to meet a requirement of Keras regarding the target class: the target needs to be represented in a number encoded format in order for cross entropy loss to work, which I added as the last step to preprocessing.

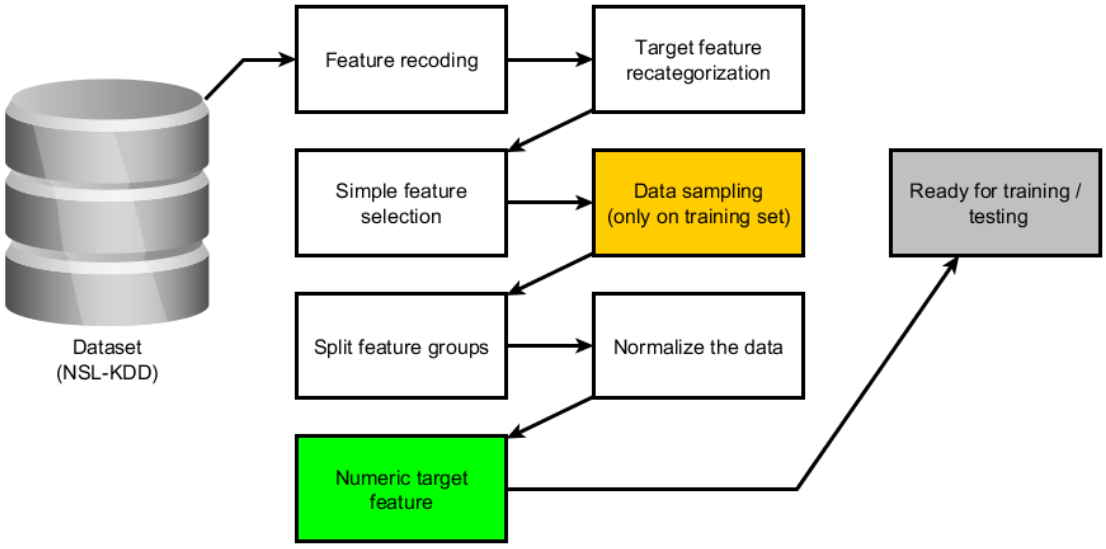

Figure 32: Data preprocessing for the V2 models. Source: own edit

Regarding execution speed, although more manageable, the NSL-KDD training dataset still contained enough observations and features to make synthetic resampling a slow process to execute. After taking the recommendations of (scikit-learn developers, 2018), the following adjustments were made to the synthetic samplers:

- I set all their n_jobs parameter to -1. This setting enables multi-threaded execution during resampling, the -1 value tells the code to use all available CPU cores for execution, thus enabling it to take advantage of all available resources.
- The SVM classifier used by SVM SMOTE has no n_jobs parameter, instead, it is optimized with the cache_size parameter. Adjusting this from the default 200 MB to 4096 MB enabled faster execution for SVM SMOTE as well.

Modeling

In the modeling phase I changed the backend and the API of the neural networks and introduced the new optimization strategy. The model architecture is presented in Figure 33.

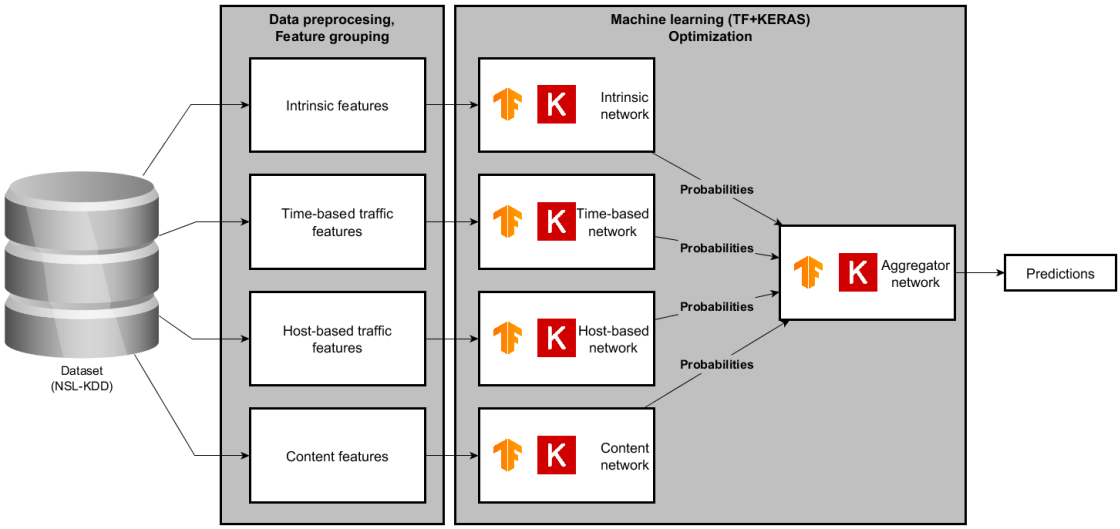

Figure 33: The model creation and prediction process of the V2 models implemented in Keras on TensorFlow backend. Source: own edit

The architecture setup shares a lot in common with the V1 model: I trained and optimized each base model, then trained the aggregator on the class probability predictions of the base models. The differences were in the backend and the hyperparameter optimization strategy I used. One of the many reasons for choosing TensorFlow was a better access to computational resources, notably the potential to access the GPU of the computer modeling is performed on. A question is the degree of benefit from doing so, as GPU training involves a computational overhead to set the data up for processing.

Furthermore, Bayesian model optimization together with the flexibility of TF and Keras, allowed training to explore a wider range of hyperparameters, for example, the number of hidden layers, the number of neurons per hidden layer and the activation function per hidden layer together with the already explored learning rate and learning rate decay over time parameters. This expanded optimization has the potential of finding more accurate predictions. I chose TPE algorithm for hyperparameter optimization, as it possessed advantageous properties compared even to gaussian process optimization. The target measure to optimize for has been the sparse categorical cross entropy loss function of the Keras API.

The parameterization of TPE, however, is different than that of grid search, visible in Table 10. I defined the parameter settings in accordance with Bergstra *et al.* (2011), Bergstra, Yamins and Cox (2013). The details of this is visible in Table 2. Distributions to sample from were log uniform for learning rate and dropout rate and uniform for learning rate decay. I set the number of hidden layers to be chosen from a list of values,

in this case, integers between 1 and 5 inclusive. The number of hidden layers parameter also determined the number of neurons and activations per layer parameters (one for each hidden layer), each sampling from a quantized uniform distribution converted to integer value and a choice between sigmoid, RELU and tanh functions, respectively. This dependent hyperparameter value selection is one of the many advantages of the TPE algorithm over gaussian processes. The settings in Table 10 enabled a simple neural network architecture search for each base and the aggregator model alike.

Parameter	Generator function
Learning rate	hp.loguniform(10^{-3}, 10^{1})
Dropout rate	hp.loguniform(10^{-3}, $5 * 10^{-1}$)
Learning rate decay	hp.uniform(0.1, 0.5)
Hidden layer number	hp.choice(1, 5)
Neurons per layer	hp.quniform(5, 50, q=1) converted to integer
Activations per layer	hp.choice(sigmoid, RELU, tanh)

Table 10: TPE hyperparameter settings for the V2 intrusion detectors. Source: own edit

Other parameters important to neural networks were not optimized. These were the number of epochs during training (set to 100), batch size (set to 1024) and a lower bound for learning rate reduction (set to 10^{-3}). The learning rate reduction, together with an early stopping criterion with patience set to the square root of the number of epochs were added as callback policies expanding the capabilities of the training process and reducing execution time. Another unaffected parameter was L2 regularization, the coefficient of which I fixed at 10^{-3}. Finally, I used the Adam solver of Kingma and Ba (2014) for training, just like with the V1 model of chapter 4.2.2.

Evaluation

The evaluation process is the same as it was in previous experiments, however, I altered the scope of measures as I only examined accuracy and recall. My choice for these two measures was influenced by their widespread use and general recommendations in the literature.

The main benefit I expected from using TensorFlow and Keras were the potential of better optimized neural network training algorithms, which can exploit the capabilities of multi core CPUs as well as GPUs. Moreover, Keras models are more flexible when it comes to parameter settings, enabling per-layer activation functions, neuron counts, regularization, etc. With TPE optimization, the key advantage is that it performs a limited set of trials, just like how random search works, however it converges on good results faster compared to grid search, and, in some cases, it can even outperform it.

As a secondary goal, I evaluated more advanced synthetic sampling approaches as part of this iteration. These included SMOTE Tomek (Batista *et al.* (2003)), SMOTE ENN (Batista, Prati and Monard (2004)) and SVM SMOTE ((Nguyen, Cooper and Kamei 2009)). Based on empirical results, I expected models trained on samples generated by SVM SMOTE to perform slightly better, due to how it samples from border regions.

Potential improvements of the model

The models trained in this iteration only performed signature detection. I planned to evaluate at least one hybrid intrusion detector in this dissertation, providing more insight to the second research question I formulated in chapter 3.3. I found a good candidate in the shape of autoencoder networks. Two benefits from using them are:

- First, the least complicated single layer autoencoder networks can be viewed as a nonlinear extension of the PCA algorithm, therefore, they are inherently capable of dimensionality reduction.
- Second, being neural networks themselves, I could integrate them into the V2 stacking neural network architecture demonstrated in this chapter.

4.2.4. AUTOENCODER ENHANCED STACKING NEURAL NETWORK

The key improvement of this model version over V2 is the extension of base classifiers with deep autoencoder networks trained only on normal traffic. Changes to data preparation and modeling processes were only minor, mostly involving the usage of the best performing elements described in chapter 4.2.3. My goal with the addition of autoencoders was increasing the quality of model predictions, justifying the ordinal increase in the naming convention to V3 in the following chapters.

Data Preparation

The data preprocessing in Figure 34 only saw minor changes, aimed at optimizing the workflow and at synthetic sampling. I implemented the former by using new preprocessing tools offered by the latest release of the scikit-learn API (Pedregosa *et al.* (2011)), and by merging logically similar transformations in a single step. I managed to join one-hot encoding, min-max normalization and target feature numerical encoding together in the same preprocessing step. The second change I implemented was the use of SVM SMOTE sampling, as it proved to be the best performing synthetic sampling

process according to the studied literature, particularly to Lopez-Martin, Carro and Sanchez-Esguevillas (2019).

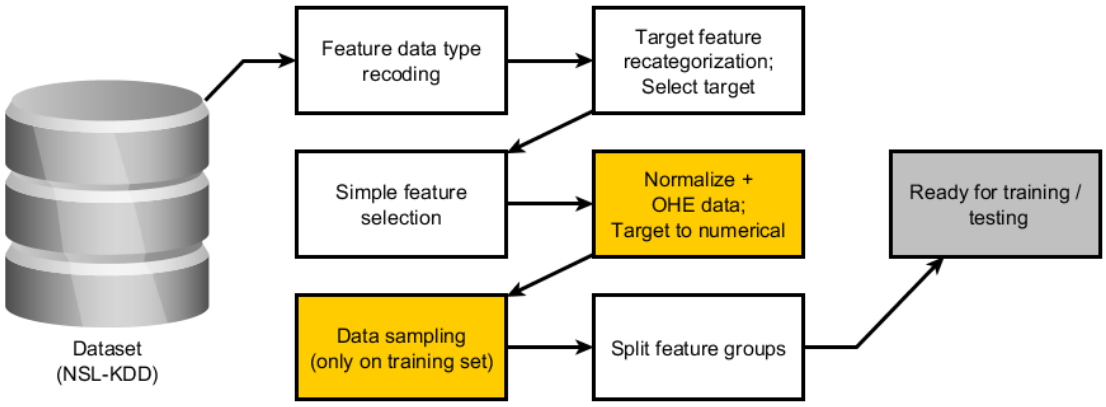

Figure 34: Data preprocessing for the V3 architecture. Source: own edit

Modeling

Model training (Figure 35) received a major update when I added deep AEs to the base classifier levels. I trained and saved each of these autoencoders only on normal traffic in a separate process, then, before training the base models of the neural network stacking model, I loaded and used these autoencoders to predict all connection data. Attack connections are predicted as if they were normal traffic, therefore I expected the squared difference between the actual and predicted features to be higher for attacks than for normal traffic. This difference can be calculated for each observation and feature, yielding new datasets to train and test with. I performed the rest of model training as I described in chapter 4.2.3, I used the TPE algorithm for hyperparameter optimization with the same hyperparameter space definitions I shown earlier in Table 10.

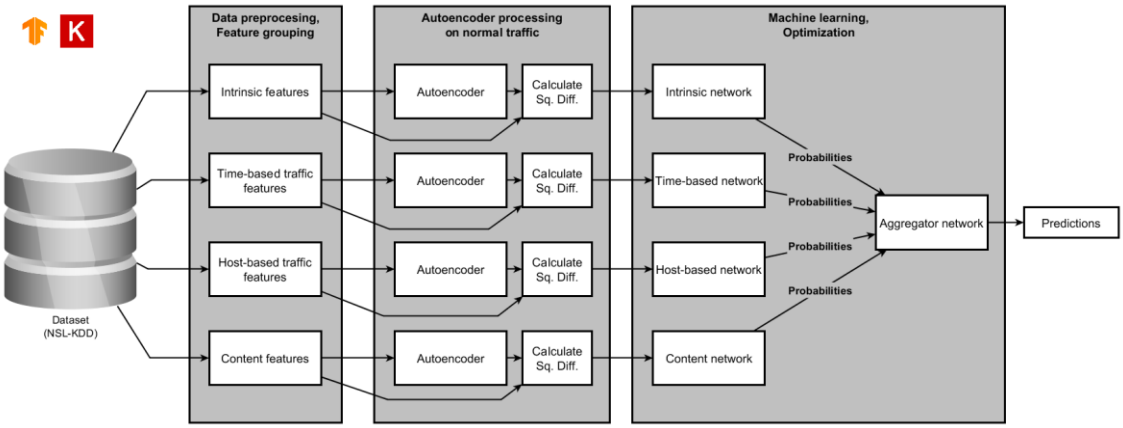

Figure 35: The model creation and prediction process of the V3 model. Source: own edit

I set up the architecture of the deep autoencoders differently than the architecture of the stacking neural network models, due to their different purpose. These different settings are shown in Table 11. As part of the study I performed on deep AE networks, I experimented with linear, sigmoid, RELU and tanh activation functions applied on all hidden layers of the autoencoder with the intent of using the activation which provided the lowest MSE on the target features. Further parameters I used were the Adam solver with default learning rate, and early stopping criterion, which was a policy shared between the autoencoders and the models of the stacking neural network with identical parameters. I did not perform regularization on the hidden layers of the autoencoder.

Parameter name	**Parameter setting**
Activation	Linear
Layer reduction rate	2
Optimizer	Adam (LR=0.001)
Bottleneck	Round($\sqrt{\lvert input \rvert}$)
Epochs	100
Early stopping patience	$\sqrt{Epochs}$

Table 11: Autoencoder parameter settings. Source: own edit

To better understand layer reduction rate and bottleneck parameters of Table 11, the setup process of one autoencoder must be understood first. I divided this process into two stages:

1. **Encoder construction**: the input feature count is taken to be the neuron count for the first hidden layer. For each subsequent hidden layer, the used neuron count is saved to a list and the neuron count for the subsequent layer is divided by the layer reduction rate. Then, the next iteration is performed with the new calculated neuron count. This iteration continues while the current neuron count is larger than a predefined bottleneck parameter, set to the square root of the input feature count.

2. **Decoder construction**: the decoder network is constructed from the reversed list of encoder neuron counts.

Evaluation

I did not change the stacking neural network performance evaluation process from the previous iterations described in chapter 4.2.3, I kept accuracy and recall as the core metrics to demonstrate. I did so to maintain the ability to compare results achieved by this variant with the results of previous variants.

I tested the autoencoders separately, on training and test datasets of NSL-KDD. The predictive performance of autoencoders is defined on how closely they can reconstruct data from a low-dimensional representation. This is best characterized using the MSE function. I performed these comparisons for each class value and feature group, expecting different results by both values and groups, but not much different between the training and test datasets. A further evaluation of autoencoders was the small optimization of the activation functions, which I already described.

Potential improvements of the model

Although this iteration had the potential for the most promising results, I still found opportunities for adjustments. Particularly, the following could be improved on:

- Although NSL-KDD (and KDD Cup 1999) is a widely studied and accepted benchmark for comparing intrusion detectors, its source is one of the oldest in the field, dating back to 1999. Although both NSL-KDD and KDD Cup 1999 simulate the appearance of new attacks by excluding some attack categories from the training set, new datasets created since then may have new insights useful for machine learning algorithms. New candidate datasets include Kyoto 2006 (Song *et al.* (2011)), UNSW-NB15 (Moustafa and Slay (2015)) and CSE-CIC-IDS2018 (Sharafaldin, Lashkari and Ghorbani (2018)). UNSW-NB15 looks to be the most promising, it is more up to date, has dedicated training and test datasets and both packet and flow features.

- Deep autoencoders are just one type of autoencoders that can be effective in an intrusion detection environment. Newer autoencoders include sparse autoencoders and variational autoencoders, both used effectively for improving intrusion detector systems in Al-Qatf *et al.* (2018), Lopez-Martin, Carro and Sanchez-Esguevillas (2019) and Yang *et al.* (2019).

- I used the results of the autoencoders as the squared difference between the original and the predicted features. This approach is uncommon, most studies (Al-Qatf *et al.* (2018), Lopez-Martin, Carro and Sanchez-Esguevillas (2019) and Yang *et al.* (2019)) used the latent representation of a variational autoencoder to generate new outputs or used the reduced dimensional representation. When applied to the intrusion detector demonstrated in this chapter, this change would likely reduce model complexity to a level, where even the use of model ensembles

could be omitted entirely, and instead, a neural network could be trained directly on reduced dimensional set of features. Furthermore, it is also possible that, being generative models, variational autoencoders could replace synthetic sampling altogether.

- Finally, the current and previous iterations all relied on feature groups proposed primarily by Stolfo *et al.* (2000). The original purpose of these feature groups was to provide feature subsets better suited at detecting specific classes of traffic. A similar process could be developed to calculate feature importance per class in a one vs rest setting and take the top performing features to further modeling. This would create an ensemble built from base classifiers performing well for on specific class each. For example, these base classifiers could be the encoders of trained autoencoder networks, although it is to be determined whether a single variational autoencoder would provide better predictions.

5. RESULTS

In this chapter I summarize the detection results that the models achieved. I organized this chapter similarly to the model outline in chapter 3.4 and Figure 26: I discuss the results of each model in the order of their creation, going from V0 to V3. Following the separate evaluation of these detectors, I discuss their results in relevance to each other. Finally, in chapter 5.6, I take these results an compare them with results provided by several papers in the field of intrusion detection research to see if my detectors can compete in a wider scientific context.

5.1. DECISION TREE BAGGING RESULTS

The results of the V0 detector can be seen in Table 12 and Table 13 . The metrics I used were accuracy, precision, recall, F_1-score and, in the case of binary classification, AUC, all calculated using custom samples from the 10% KDD Cup 1999 dataset. In the original article of Brunner (2017) I evaluated the effects of parallelization on classification performance using differently sized training and test samples as well. There, I concluded that map-reduce parallelization and different sample sizes had no effect on the prediction performance of the base classifier models. Therefore, when I aggregated the performance metrics for demonstration in this dissertation, I filtered the effects of both characteristics out of the aggregate results calculation. For those interested in the original measurements, I made the source tables available in Appendix B.

Binary classification (Table 12) achieved moderate accuracy and AUC at 78.8% $\pm$ 1.23% and 0.773 $\pm$ 0.0237 respectively. The $\pm$ components were due to the aggregation; they do not indicate cross validation folds. Precision was the highest metric at 0.925 $\pm$ 0.0372 and recall and F_1-score were the lowest at 0.513 $\pm$ 0.0332 and at 0.659 $\pm$ 0.0266.

Measurement	Value
Accuracy	78.8% (±1.32%)
Precision	0.925 (±0.0372)
Recall	0.513 (±0.0332)
F_1-score	0.659 (±0.0266)
AUC	0.773 (±0.0237)

Table 12: Aggregate measurements for the binary classification case of the V0 model. Based on Brunner (2017)

Comparing binary with five-class classification (Table 13), where accuracy was the highest at 97.9% $\pm$ 1.24% with the remaining precision (0.491 $\pm$ 0.0944), recall (0.458

$\pm$ 0.0582) and F_1-score (0.473 $\pm$ 0.0708) measurements showing worse performances. The high accuracy in this case is misleading, as training and test samples were not balanced at the time, and accuracy tends to rate classifier models better when they assign most observations to the majority class.

Measurement	Value
Accuracy	97.9% (±1.24%)
Precision	0.491 (±0.0944)
Recall	0.458 (±0.0582)
F_1-score	0.473 (±0.0708)

Table 13: Aggregate measurements for the five-class multiclass classification case of the V0 model. Based on Brunner (2017)

Binary classification achieved better results. This is understandable as binary classification only requires models to make a choice between normal and attack traffic, also highlighted by Petersen (2015). This also means that the misclassifications made between the different attack classes remain masked. Depending on the intrusion detection controls requested, this masking may be unacceptable, as different control policies need to be applied for different attack patterns. Therefore, I considered studying intrusion detection performance as a multiclass classification problem more favorable in further intrusion detectors. One candidate algorithm recommended were artificial neural networks, preferably joined in an ensemble, for example, in the form of stacking classifiers.

The results of this chapter provide one example answer to my first research question in chapter 3.3.

5.2. STACKING NEURAL NETWORK RESULTS

The datasets I used to train the V1 ensemble were KDD Cup 1999 and NSL-KDD. Due to differences in dataset sizes and certain steps of data preprocessing I applied, it would be a mistake if I was to aggregate or even compare results achieved on the two datasets with each other. Therefore, I decided to show the results separately, first on KDD Cup 1999, where I tested the stability of the custom sampling process I designed, and prediction performance of the models. This time I included detection performance metrics at base classifier and aggregator levels both, an improvement over the V0 model of chapter 5.1. On NSL-KDD, I evaluated detection performance only, as I simplified the data sampling process to only include SMOTE sampling.

KDD Cup 1999 - Sampling stability

Sampling stability evaluation was designed to test the specialized two-stage sampling process, repeated 150 times. I used Kolmogorov-Smirnov tests at each sampling iteration to compare the original 10% KDD Cup 1999 dataset and the generated samples for each class and feature, including one-hot encoded categorical columns. I set a special decision column to 0 if I could not reject the null hypothesis of K-S test, 1 otherwise. The result is an aggregation of these decisions to target class values. Table 14 shows these aggregate results, containing mean rejection rate with ~95% confidence interval estimated from standard deviation.

H_0	Normal	DoS	Probe	R2L	U2R
Rejected %	0.16%±0.13%	0.02%±0.04	0.00%±0.00%	0.00%±0.00%	2.73%±0.76%

Table 14: V1 sampling validation results on KDD Cup 1999 data. Source: own edit

Sample testing results show that probe and R2L classes matched the original data perfectly for all explanatory features, while DoS and normal categories matched their respective distributions in the majority of tests. The only exception was the U2R class, where 2.73% (with a confidence interval of ±0.76%) of tests rejected the possibility that the sample has been drawn from the same distribution as the 10% KDD Cup 1999 dataset. It is likely that this has been caused by how underrepresented U2R class was in the original dataset.

KDD Cup 1999 - Model performance

I trained each model (the four base classifiers and the aggregator model) using the preprocessed training sample and grid search hyperparameter optimization. I tested 540 different combinations in total with exact hyperparameter values I shown earlier in Table 9.

I performed model testing using the dedicated test dataset of KDD Cup 1999. The results of this can be viewed in Table 15 and Table 16. Table 15 shows achieved model accuracies for each class and base model as well as accuracy achieved by the ensemble under the aggregator model. The best base models were those trained on intrinsic and host-based traffic for normal, host-based traffic for DoS, time- and host-based traffic for probe and R2L and host-based traffic and content for U2R classes. Overall, content model performed the worst, however, I cannot say it was completely redundant, as it still contained useful information about U2R attacks, partially confirming the findings of Stolfo *et al.* (2000).

The aggregator model improved accuracy further compared to the base models. The aggregator model improved detection accuracy for nearly every class, the only exception being probe detection on the model trained on host-based traffic data (99.16% against 99.07% of the aggregator), but even there the difference is minor.

The final row of Table 15 shows overall accuracies for each model. Despite how it seems, this overall accuracy value has no connection with the per-class values. Per class accuracies were meant to measure model performance in detecting that one class, while overall accuracy is measuring the performance of a model in general. Based on overall accuracies, the aggregator managed to improve the detections of all models, with the base model trained on host-based traffic features being the closest in detection.

	Intrinsic	Time-traffic	Host-traffic	Content	Aggregator
Normal	91.76%	79.54%	92.10%	63.26%	92.13%
DoS	83.56%	85.11%	96.71%	17.16%	96.74%
Probe	85.27%	99.05%	99.16%	15.86%	99.07%
R2L	93.83%	93.36%	94.17%	74.41%	94.70%
U2R	98.62%	86.50%	99.32%	99.77%	99.92%
Overall	78.77%	74.94%	91.03%	15.31%	91.52%

Table 15: Aggregate V1 model accuracy with base model accuracies measured on KDD Cup 1999. Source: own edit

Table 16 shows the remaining aggregate classification measures for base and aggregator models. Class recall, precision and F1-scores were all macro-averaged to calculate the results shown in Table 16. As I mentioned before, the primary measure I evaluated was recall, which shown promising results with the aggregator and one base model trained on intrinsic features as well. However, only the aggregator model could achieve consistently high recall together with high precision (achieving the highest F_1-score as a result).

	Intrinsic	Time-traffic	Host-traffic	Content	Aggregator
Recall	0.668	0.635	0.595	0.470	0.665
Precision	0.476	0.447	0.555	0.333	0.626
F_1 score	0.402	0.442	0.525	0.269	0.582

Table 16: Macro-averaged precision, recall and F_1-score of the V1 model measured on KDD Cup 1999. Source: own edit

I also attached a more detailed version of Table 16 showing base and aggregator model performances on intrusion class value level in Appendix C.

NSL-KDD – Model performance

The performance comparison on NSL-KDD dataset is available in Table 17. Out of the base classifiers, intrinsic model performed the best on normal class, time-based traffic

model detected DoS, probe and U2R attacks well and host-based traffic model was the best on R2L class. Content model was the worst at detecting attacks except R2L and U2R. The final aggregator improved on almost every class, except normal traffic, where it could not achieve better per class accuracy than the base classifier trained on intrinsic data.

Based on overall accuracy, I set up the following ranking from worst to best model: content, intrinsic, host-traffic, time-traffic and aggregator. Performance improvement achieved by the aggregator model is understandable as it uses knowledge and patterns acquired earlier by the base classifiers.

The results the models achieved when I used the NSL-KDD dataset for training are worse than the results achieved when I used the KDD Cup 1999 dataset. Due to redundancies, certain observations received a higher representation in training and test datasets. Correctly classified, these redundant records have a stronger representation in Table 15, compared to Table 17, where each observation is equal in importance. This is a possible reason why accuracies trained on KDD Cup 1999 seem to be better.

	Intrinsic	Time-traffic	Host-traffic	Content	Aggregator
Normal	84.71%	80.30%	79.44%	81.58%	82.30%
DoS	81.73%	88.14%	86.97%	70.29%	91.00%
Probe	79.74%	94.44%	92.00%	89.26%	93.64%
R2L	88.56%	86.46%	89.18%	89.09%	90.09%
U2R	96.98%	99.35%	97.65%	98.07%	99.18%
Overall	65.86%	74.34%	72.62%	64.15%	78.11%

Table 17: Aggregate V1 model accuracy with base model accuracies measured on NSL-KDD. Source: own edit

In Table 18 recall, precision and F1-score are visible, calculated on the test dataset of NSL-KDD. Based on recall, the intrinsic model performed best out of the base models, even outperforming aggregate results. With precision, it was the base model trained on the host-based traffic features that provided the best result, closely followed by the model trained on time-based traffic features, which simultaneously provided the best F_1-score. The aggregator model managed to improve precision and F_1-score compared to the base models.

	Intrinsic	Time-traffic	Host-traffic	Content	Aggregator
Recall	0.576	0.609	0.515	0.453	0.558
Precision	0.512	0.580	0.607	0.451	0.668
F_1 score	0.483	0.584	0.509	0.372	0.566

Table 18: Macro-averaged precision, recall and F_1-score of the V1 model measured on NSL-KDD. Source: own edit

As with the KDD Cup 1999 measurements, I published a detailed version of Table 18 in Appendix D for those who are interested. As I considered aggregator model performance to be much more important to analyze in this dissertation, I decided to exclude base model performance measurements from further chapters. Moreover, instead of providing class-specific recalls, precisions and F_1-scores for each model, I only shared confusion matrices in further appendices tied to chapters 5.3 and 5.4. I found this an easier approach to follow, while allowing the reader to calculate additional performance metrics as they see fit.

The results of this and the previous chapter confirm that machine learning works as an intrusion detector, answering the first research question in chapter 3.3.

5.3. KERAS AND TENSORFLOW STACKING NEURAL NETWORK RESULTS

I compared V2 models for performance in groups determined by the synthetic sampling approach. I executed all experiments according to details I explained in chapter 4.2.3, with 50 hyperparameter optimization iterations at first. However, with 50 iterations, the aggregator models started to show signs of overfitting, therefore, later I reduced the number of hyperparameter optimization iterations to 25. Results in this and the following chapters were provided by the models performing best out of these 25 iterations.

Earlier I mentioned that a key advantage of TensorFlow is the potential execution time improvement on computers equipped with a GPU. This improvement is conditional, requiring a setup overhead from the TensorFlow backend and depends on the dimensions of the weight matrix. For example, a more complex model with weights in the millions, GPU utilization is highly beneficial, as it was determined by Lind and Pantigoso Velasquez, (2019) as well. As none of the models trained in this dissertation reached such complexities, I decided to drop GPU utilization and work with CPU only instead.

Model prediction performances are visible in Table 19 for accuracy. The models performed well on each class, regardless of the sampling approach used. The class all models had difficulty predicting was normal, which indicates that a large portion of attacks were classified as normal traffic incorrectly. I excluded overall accuracies from Table 19, for the same reasons I highlighted while discussing the accuracies in Table 15 (per-class and overall accuracies are different metrics). Overall accuracies were 77.09% for SMOTE ENN, 78.34% for SMOTE Tomek and 77.75% for SVM SMOTE. In this

regard SMOTE Tomek provided the best predictions, although, the differences between the sampling methods are rather small.

Accuracy	SMOTE ENN	SMOTE Tomek	SVM SMOTE
Normal	80.67%	82.38%	80.78%
DoS	90.16%	90.52%	90.74%
Probe	93.01%	93.14%	93.57%
R2L	90.68%	91.02%	90.73%
U2R	99.66%	99.64%	99.68%

Table 19: Aggregate V2 model accuracies. Source: own edit

The recall values of Table 20 provide more information on predictions. The models provided the best results on the majority normal and DoS classes and predicted probe, R2L and U2R classes worse as they started belonging more and more to minority. Moreover, the sampling methods provided similar macro-averaged recall values ranging within one percentage point, with the minor advantage of SMOTE ENN sampling.

Recall	SMOTE ENN	SMOTE Tomek	SVM SMOTE
Normal	0.9255	0.9198	0.9140
DoS	0.8259	0.8592	0.8438
Probe	0.5225	0.5580	0.5944
R2L	0.3258	0.3289	0.3109
U2R	0.3731	0.2985	0.2985
Average	0.5946	0.5929	0.5923

Table 20: Aggregate V2 model recalls. Source: own edit

Based on the data I collected, I cannot state with certainty which synthetic sampling of the three evaluated can improve model performance the most, therefore, I compared the results with the models discussed in chapter 2.3. Lopez-Martin, Carro and Sanchez-Esguevillas (2019) reported SVM SMOTE models to give a small advantage, therefore I used this sampling approach for the model described in chapter 4.2.4. To assist with further performance analysis, I attached the confusion matrices for all the V2 base and aggregator models to this dissertation in Appendix E, Appendix F and Appendix G.

As part of my third research question, I set the goal of finding additional techniques that could help an intrusion detector in providing more accurate predictions. This chapter provided models enhanced with two such techniques: synthetic sampling and more advanced hyperparameter optimization.

5.4. AUTOENCODER ENHANCED STACKING NEURAL NETWORK RESULTS

My goal with the autoencoder iteration was to evaluate the predictive performance of a hybrid intrusion detection solution. I implemented this hybrid detector by extending the V2 model of chapter 4.2.3 with autoencoders trained on normal traffic. This process required evaluations on two artefacts: the autoencoders themselves, and the extended stacking model.

Autoencoder model results

I conducted two analyses on the autoencoder models: the first involved the testing and evaluation of the activation functions. I set this up in a way similar to grid search hyperparameter optimization. I measured the results of these experiments using the mean squared error function common in regression tasks. The measured MSE values can be seen in Table 21. All autoencoders, except the one trained on intrinsic data, performed the best with linear activation functions, while intrinsic AE performed better with RELU activations. Based on these results, I decided to use linear activation for each autoencoder.

Feature Group	Linear	RELU	Sigmoid	Tanh
Intrinsic	0.00060	0.00012	0.00205	0.00013
Content	0.00016	0.00031	0.00186	0.00018
Host-traffic	0.00304	0.00461	0.01012	0.00335
Time-traffic	0.00442	0.00609	0.01096	0.00447

Table 21: Autoencoder MSE per feature group and activation. Source: own edit

My second analysis of autoencoders measured their performance. Just like with the previous analysis, I used the MSE between the original explanatory features and the predictions the AE models made. With this analysis I aimed to prove the usefulness of training autoencoders on normal traffic only. Because the models only saw normal traffic, their reconstruction error would be much higher on attack classes. the results of this analysis can be seen in Figure 36.

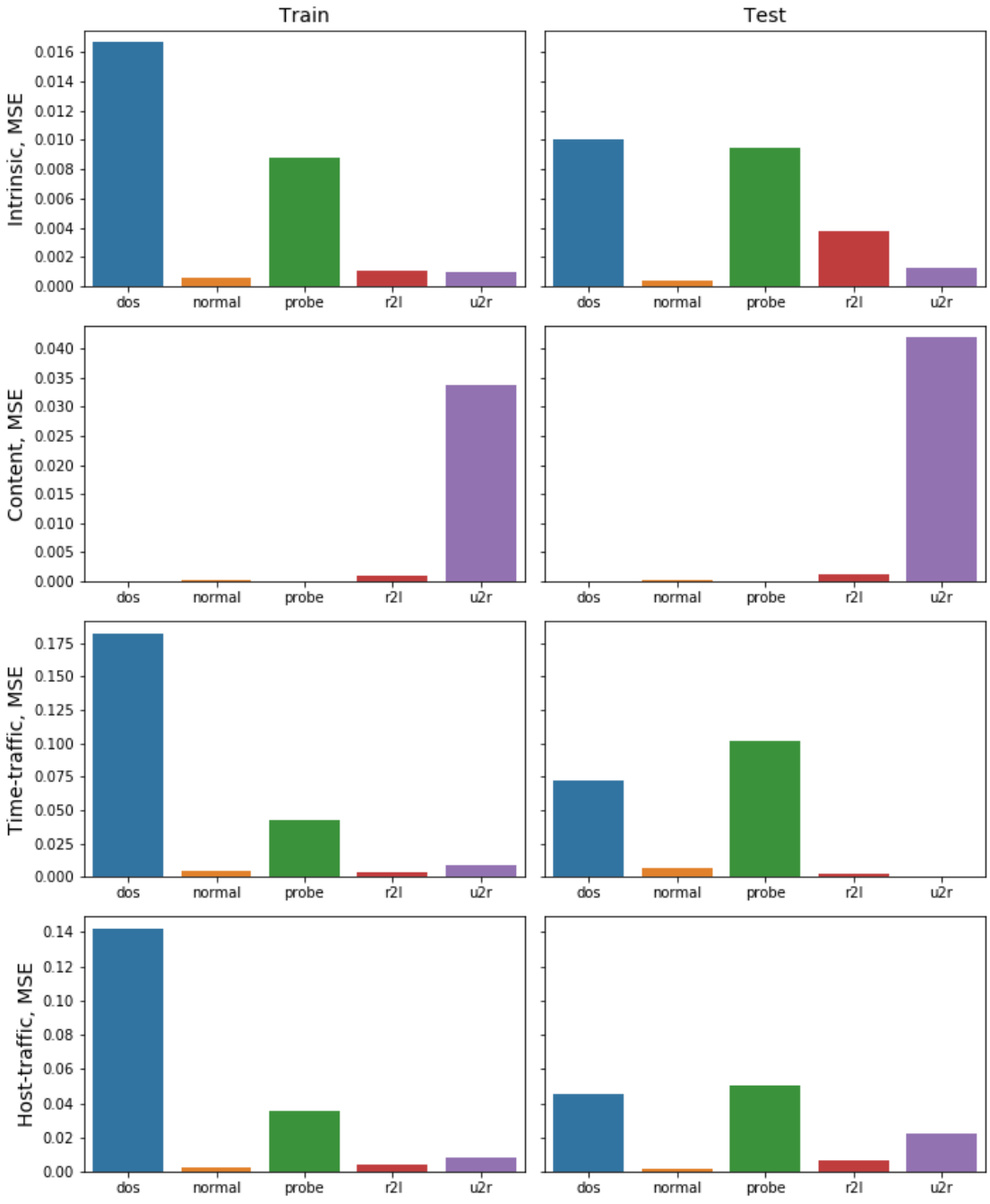

Figure 36: Per-class autoencoder model errors on NSL-KDD dataset. Source: own edit.

Figure 36 shows the per-class and per feature group performance of the autoencoder models. The intrinsic and the two traffic AEs were good at differentiating DoS and probe attacks, R2L and U2R classes were more challenging, though: MSEs for these categories were similar, or even lower than what the three models achieved on normal traffic. This similarity, together with their minority state makes these classes even harder for machine learning models to detect. The last remaining content autoencoder was unique in how well it managed to find differences between normal and U2R traffic. This is another proof,

that, no matter how poorly models trained on content features perform, they still contribute to the overall performance.

Based on the results in Figure 36, an autoencoder network can work as an anomaly detector, partially answering my second research question.

Model performance

Following autoencoder model predictions, the stacking neural networks were trained as usual. As in chapter 5.3, I show two tables, one for the accuracy measures and the other for recall values, both describing only the aggregator model predictive performance.

Table 22 shows per class accuracies. The model provided the best accuracy for U2R, while DoS, R2L and normal also maintained high detection accuracies. Probe class provided the worst per-class accuracy at only 84.95%. Overall accuracy of the aggregator model was 74.26%, a surprisingly decreased value compared to the results published in chapter 5.3.

Normal	DoS	Probe	R2L	U2R
87.20%	90.82%	84.95%	88.84%	96.72%

Table 22: Aggregate V3 model accuracies. Source: own edit

Viewing the analysis from a different perspective, Table 23 shows the recalls of the aggregator model. The best recall values were achieved on normal, DoS and probe traffic. The worst achieved recall was on R2L. Their average improved compared to the earlier iterations.

Normal	DoS	Probe	R2L	U2R	Average
0.8367	0.7728	0.7732	0.3262	0.5821	0.6582

Table 23: Aggregate V3 model recalls. Source: own edit

As in the previous chapters, I published the confusion matrices of all the trained neural networks in Appendix H.

This chapter proved that hybrid models work as viable intrusion detector models. Additionally, the V3 model further utilized synthetic sampling and advanced hyperparameter optimization, providing more evidence to answer my third research question.

5.5. COMPARISON OF EXPERIMENTAL RESULTS

The most important conclusions can be drawn when one compares the results of each model variant. In this chapter, I performed this comparison between these variants on a per-class value basis trained on the NSL-KDD dataset. These comparisons encompass two metrics, accuracy and recall. To make these comparisons easier, I created a ranking scheme inspired by the ranking ideas outlined in Kovács and Kő (2018), adjusted to benefit from the characteristics of confusion matrices.

Table 24 demonstrates the aggregator model accuracies. As in previous chapters, V1 stands for the scikit-learn-based NN staking ensemble, V2 for the TF + Keras stack and V3 for the model enhanced by autoencoders. Interestingly, according to per-class accuracies, the V1 model outperformed the more complex Keras models, particularly in detecting DoS and probe attacks. Moreover, the V3 model is not necessarily the best performing model either, only providing better results for normal traffic. I explain this with the autoencoder training process and how I trained them on normal traffic. Following neural networks saw explanatory features that were less different from normal traffic, therefore they have gotten better at detecting that exact class.

Based on overall accuracies, V2 SMOTE Tomek performed best with 78.34%, followed by the V1 model at 78.11% and V2 SVM SMOTE at 77.75%. The last two models were the V2 SMOTE ENN and the V3 detectors, both achieving 77.09% and 74.26%, respectively.

	V1	V2			V3	Support
		SMOT ENN	SMOTE Tomek	SVM SMOTE		
Normal	82.30%	80.67%	82.38%	80.78%	87.20%	9711
DoS	91.00%	90.16%	90.52%	90.74%	90.82%	7460
Probe	93.64%	93.01%	93.14%	93.57%	84.95%	2421
R2L	90.09%	90.68%	91.02%	90.73%	88.84%	2885
U2R	99.18%	99.66%	99.64%	99.68%	96.72%	67

Table 24: Accuracy table for all model variants. Source: own edit.

Model performance shifts when I include the recall measures of Table 25. With recall, I observed that the V3 model traded performance on majority classes (normal and DoS) for performance on minority classes, especially U2R. This caused a significant increase in the macro-averaged recall.

	V1	V2			V3	Support
		SMOT ENN	SMOTE Tomek	SVM SMOTE		
Normal	0.9452	0.9255	0.9198	0.9140	0.8367	9711
DoS	0.8126	0.8259	0.8592	0.8438	0.7728	7460
Probe	0.6898	0.5225	0.5580	0.5944	0.7732	2421
R2L	0.2400	0.3258	0.3289	0.3109	0.3262	2885
U2R	0.1045	0.3731	0.2985	0.2985	0.5821	67
Average	0.5584	0.5946	0.5929	0.5923	0.6582	22544

Table 25: Recall table for all experiments. Source: own edit

I also included support values to Table 24 and Table 25. This support stands for how many records from the test dataset belongs to a specific class. I used these support values for a performance ranking as inverse class weights. First, I ranked the models for each class and metric, extending this ranking to the overall accuracy average precision values as well, assigning them artificial 50% weights. I distributed the remaining 50% between the classes according to the following formula:

$$class\_weight_j = (1 - \frac{support_j}{\sum_{j=1}^{k} support_j})/(k-1) * 2$$

My goal with this formula was to penalize the effect of majority classes more, and the multiplication by 2 in the denominator was needed to adjust the sum of weights to 50%. The end results of this ranking process are visible in Table 26. When ranked according to accuracy, the V2 SMOTE Tomek model performed best. With recall rankings, the V3 model proved to be the best alternative.

	V1	V2			V3
		SMOT ENN	SMOTE Tomek	SVM SMOTE	
Accuracy	2.34	3.80	1.79	2.60	4.46
Recall	4.30	2.53	2.79	3.66	1.73

Table 26: Model rankings in terms of accuracy and recall. Source: own edit

With all the above considered, determining the best intrusion detector still remains a challenging task, influenced by the problem the models were created to address. As the cost of predicting a false negative is greater, going for a high recall is preferable. Based on this line of thought, the V3 stacking neural network model extended with autoencoders is the correct model to choose.

The models' comparisons in this chapter provided additional information on the performance levels of several intrusion detectors necessary to answer the third research question in chapter 3.3. These models include ensemble signature detectors supported by

different variants of synthetic sampling and a hybrid detector model based on a deep autoencoder network. Depending on the choice of performance metric, stacking models enhanced by autoencoder networks can provide an improved detection performance.

5.6. COMPARISON TO EXTERNAL RESULTS

To truly understand how the proposed model predictions performed, I compared them to results I found in the wider intrusion detection context. This information gathering proved to be a more challenging task than I anticipated at first. For example, most intrusion detection papers published accuracy only as the primary metric for intrusion detection. Accuracy alone can be a misleading measure when class imbalance is high. Recall, a better metric for intrusion detection is seldom published, and even if it is, it is often referred to by different names, like detection rate or sensitivity, or defined incorrectly with the formula of accuracy or other metrics. The fact that there are three different averaging methods for recall (macro, micro and weighted) did not help either. The third challenge I faced was with class assignments. Separate studies assigned detailed attack categories to different classes. Luckily, this issue was only prevalent in attack categories present in the test samples of the DARPA 1998 family of datasets, attack classes in the training datasets were sufficiently described in Stolfo *et al.* (2000). Nonetheless, this issue with class assignments was the main reason why I decided to conduct the analysis of Appendix A. With these difficulties and my earlier results in mind, I created a comparison table based on the key metrics I collected.

Table 27 shows the results comparison with intrusion detection papers. I collected most of these from papers studying autoencoder network performance and included the performance of non-ensemble models as well. The mean accuracy of the available models was 77.72%; V2 SMOTE Tomek, V2 SVM SMOTE and V1 models managed to outperform this from my proposed models. Yang *et al.* (2019) also published model recalls, the average of which was 51.23%. All models published in this dissertation managed to perform above this value. In fact, the autoencoder enhanced model achieved the best recall, even in comparison to the best models in the intrusion detection literature.

Model	**Accuracy**	**Recall**
KNN (Yang *et al.*, (2019))	76.51%	48.3%
Multinomial NB (Yang *et al.*, (2019))	78.73%	47.69%
RF (Yang *et al.*, (2019))	76.49%	48.84%
SVM (Yang *et al.*, (2019))	72.28%	45.88%

Model	Accuracy	Recall
DNN (Yang *et al.*, (2019))	80.22%	52.77%
DBN (Yang *et al.*, (2019))	80.82%	53.61%
ROS-DNN (Yang *et al.*, (2019))	78.26%	49.59%
SMOTE-DNN (Yang *et al.*, (2019))	81.16%	51.49%
ADASYN-DNN (Yang *et al.*, (2019))	80.1%	51.47%
ICVAE-DNN (Yang *et al.*, (2019))	85.97%	62.66%
VGM + RF (Lopez-Martin, Carro and Sanchez-Esguevillas, (2019))	73.61%	N/A
VGM + Logistic Regression (Lopez-Martin, Carro and Sanchez-Esguevillas, (2019))	77.29%	N/A
VGM + Linear SVM (Lopez-Martin, Carro and Sanchez-Esguevillas, (2019))	77.23%	N/A
VGM + MLP (Lopez-Martin, Carro and Sanchez-Esguevillas, (2019))	79.26%	N/A
SVM SMOTE + RF (Lopez-Martin, Carro and Sanchez-Esguevillas, (2019))	74.25%	N/A
SVM SMOTE + Logistic Regression (Lopez-Martin, Carro and Sanchez-Esguevillas, (2019))	76.29%	N/A
SVM SMOTE + Linear SVM (Lopez-Martin, Carro and Sanchez-Esguevillas, (2019))	77.99%	N/A
SVM SMOTE + MLP (Lopez-Martin, Carro and Sanchez-Esguevillas, (2019))	77.98%	N/A
Decision Tree (Yin *et al.*, (2017))	74.6%	N/A
NB (Yin *et al.*, (2017))	74.4%	N/A
RF (Yin *et al.*, (2017))	72.8%	N/A
NB Tree (Yin *et al.*, (2017))	75.4%	N/A
MLP (Yin *et al.*, (2017))	78.1%	N/A
RNN (Yin *et al.*, (2017))	81.29%	N/A
SAE + SMR (Javaid *et al.*, (2016))	79.1%	N/A
AE + SVM (Al-Qatf *et al.*, (2018))	80.48%	N/A
Proposed V3 (AE + Stacking NN)	74.26%	65.82%
Proposed V2 + SMOTE ENN	77.09%	59.46%
Proposed V2 + SMOTE Tomek	78.34%	59.29%
Proposed V2 + SVM SMOTE	77.75%	59.23%
Proposed V1 (Stacking NN)	78.11%	55.84%

Table 27: External comparisons in terms of accuracy and recall. Source: own edit

The authors of Yang *et al.* (2019) published per-class recalls, enabling a more detailed comparison. In fact, global macro recalls in Table 27 were calculated from the per-class recalls shown in Table 28. The mean recall values based on the collected data were 95.5% for normal, 77.44% for DoS, 64.52% for probe, 13.84% for R2L and 4.85% for U2R classes. My proposed models performed under average for normal classes, above average for DoS, with the exception of V3, above average for probe, except for the V2 models, and all proposed models performed above average for R2L and U2R classes.

Model	Normal	DoS	Probe	R2L	U2R
KNN (Yang *et al.*, 2019)	92.78%	82.25%	59.4%	3.56%	3.5%

Model	Normal	DoS	Probe	R2L	U2R
Multinomial NB (Yang *et al.*, 2019)	96.03%	37.1%	82.61%	22.22%	0.5%
RF (Yang *et al.*, 2019)	97.37%	80.24%	58.53%	7.55%	0.5%
SVM (Yang *et al.*, 2019)	92.82%	74.85%	61.71%	0%	0%
DNN (Yang *et al.*, 2019)	96.1%	85.4%	65.3%	14.56%	2.5%
DBN (Yang *et al.*, 2019)	97.04%	83.11%	69.85%	12.56%	5.5%
ROS-DNN (Yang *et al.*, 2019)	92.61%	80.32%	56.26%	12.75%	6%
SMOTE-DNN (Yang *et al.*, 2019)	96.59%	82.19%	56.75%	10.93%	11%
ADASYN-DNN (Yang *et al.*, 2019)	96.43%	83.28%	59.81%	9.84%	8%
ICVAE-DNN (Yang *et al.*, 2019)	97.26%	85.65%	74.97%	44.41%	11%
Proposed V3 (AE + Stacking NN)	83.67%	77.28%	77.32%	32.62%	58.21%
Proposed V2 + SMOTE ENN	92.55%	82.59%	52.25%	32.58%	37.31%
Proposed V2 + SMOTE Tomek	91.98%	85.92%	55.80%	32.89%	29.85%
Proposed V2 + SVM SMOTE	91.40%	84.38%	59.44%	31.09%	29.85%
Proposed V1 (Stacking NN)	94.52%	81.26%	68.98%	24.00%	10.45%

Table 28: Recall comparison per class. Source: Yang *et al.* (2019) & own edit

The autoencoder enhanced model proposal provided the worst recall on normal connections and performed bad on DoS attacks compared to the measurement of Yang *et al.* (2019). The V3 model performed better, however, at predicting probe and U2R attacks and not much worse with R2L classes. It can be said that the V3 model traded good performance on majority classes for better classifications on minority classes, which also explains the performance degradation experienced with accuracy metrics.

This chapter summarized several works from the related literature and compared their reported performance with the models' performances I proposed in my research. Based on certain per-class and aggregate measures, at last one of the proposed models (V3) can compete and outperform works in the related literature, answering my third research question in chapter 3.3.

6. CONCLUSION

The main goal of my dissertation was to provide a novel intrusion detection solution applying machine learning methods. I have introduced the field of, the data science and machine learning tools and techniques used for, and the literature studying intrusion detection first. Then, based on the design science methodology and the CRISP-DM process model I have designed, implemented and evaluated four intrusion detector models. For evaluation I compared the four models with one another first, then with additional model proposals from the related literature. I discussed three research questions in my dissertation.

The first research question dealt with the suitability of machine learning models. Based on the literature review and the machine learning models I created, I proved that machine learning is a suitable approach for detecting intrusions. It is easy for machine learning models to provide accurate predictions when detecting DoS, probe and normal activity. Minority classes, like U2R and R2L attacks are more complicated. To overcome this challenge, the right course of action is not necessarily the choice of a new model. There is no "free lunch" in data science, there is no single best model which can give perfect predictions. Instead, a viable approach is to strive for ensemble models. The comparisons of chapter 5.2, Appendix C and Appendix D between base classifiers and aggregate classifiers have pointed out the usefulness of this approach.

Speaking of ensembles, the second research question put misuse detection with ensembles and hybrid detection into perspective. Misuse detection can achieve good results, especially when the models are combined into ensembles, but it does have its limits. To test the magnitude of detection performance increase from hybrid detection, I created the model described in chapter 4.2.4. My expectation based on the literature was an intrusion detector that achieved an even better classification performance. The results were, however, more nuanced. The V3 model did achieve the best overall recall, even when it was compared to the related literature, but its overall accuracy suffered for it. Other models, like a more advanced conditioned variational autoencoder could help clarify the results.

My third research question was about the levels of model performance. Primarily based on the related literature, I can set up a form of hierarchy between the known intrusion detection techniques. Single-model misuse detection can achieve acceptable detection

results, but better detections can be generated by ensemble models, and even better by hybrid approaches. Currently, data generative models, like CVAE and generative adversarial networks (GANs), formulate the cutting edge in intrusion detection. The question is whether there is a significant difference between the two approaches in terms of prediction performance, which is a potential topic for a future study. The dissertation also highlights two techniques important from a detection performance evaluation perspective: hyperparameter optimization and synthetic sampling. The former is underutilized in intrusion detection research, the most common technique used was grid search, when more advanced ones exist, like Bayesian optimization and tree-structured parzen estimators. For synthetic sampling, the most common method was SMOTE. The majority of the models discussed in the literature apply only the base variant of SMOTE, while I took a step further and compared SMOTE ENN, SMOTE Tomek and SVM SMOTE, However, I did not find significant differences in the predictions among the three.

A different angle intrusion detection systems can be evaluated from is more practical. Ahamad *et al.* (2009) and Beek *et al.* (2019) reported an increase in volume and complexity of cyber-crimes in the last decade, showing no signs of slowing down. By providing inputs to alerting and prevention systems, intrusion detection could play an important role in a holistic information security system. My research can additionally provide guidance on what models do and do not work for detecting malicious activity.

NIDS is not a one size fits all solution, though. There are many attacks, like social engineering that exploit the weakest link in an information security system: the human. An algorithm, no matter how well designed and implemented it might be, will not stop someone who looks like a janitor if the security guards let them in without supervision. The key to preventing such events is the application of defense in depth, meaning that a host of different controls are applied at different layers of an information system. Intrusion detection itself, for example, can protect the network layer or the host layer.

Furthermore, I left the discussion of the last steps of the design science methodology and CRISP-DM process (Table 3 and Figure 22) out of scope for my dissertation, as that would have required model implementation into a live environment, which depends on the social context of the research process. However, if implementation is the intended purpose, then a more detailed context study involving the study of stakeholder goals,

information system entry points, particularly the review of the network protocols and potential open ports, the information infrastructure, and an information security audit. This latter shall be repeated annually to not only implement, but also maintain a high-profile security infrastructure.

A different approach to intrusion detection deals with its big data nature, particularly the velocity of modern network traffic. Here, the recommendations of Molina-Coronado *et al.* (2020) and ideas from stream processing can be applied. The recommendation is an intrusion detector that learns not large amounts of data at rest, but continuously as new observations and patterns are being provided. This concept is called incremental learning, and it combines well with stream processing. Furthermore, stream processing enables the system to benefit from the otherwise underutilized temporal nature of network traffic better.

A final topic to consider during deployment is simplicity of the deployment process itself. Recent advances like containerization, and tools like Docker, can help tremendously with operation in a live environment and model maintenance. Splitting up the software environment to development, test and production enables the developers to find potential mistakes and bugs in the code, before the models get to be used.

7. REFERENCES

Abadeh, M. S. *et al.* (2007) “A parallel genetic local search algorithm for intrusion detection in computer networks,” *Engineering Applications of Artificial Intelligence*, 20(8), pp. 1058–1069. doi: 10.1016/j.engappai.2007.02.007.

Aghdam, M. H. and Kabiri, P. (2016) “Feature Selection for Intrusion Detection System Using Ant Colony Optimization.,” *IJ Network Security*, 18(3), pp. 420–432.

Ahamad, M. *et al.* (2009) “Emerging Cyber Threats Report for 2009,” *Georgia Tech Information Security Center (GTISC)*, 34, pp. 1–9.

Al-Qatf, M. *et al.* (2018) “Deep learning approach combining sparse autoencoder with SVM for network intrusion detection,” *IEEE Access*. IEEE, 6, pp. 52843–52856.

Almseidin, M. *et al.* (2017) “Evaluation of Machine Learning Algorithms for Intrusion Detection System,” in IEEE (ed.) *Intelligent Systems and Informatics (SISY), 2017 IEEE 15th International Symposium on*. IEEE, pp. 000277–000282. Available at: http://arxiv.org/abs/1801.02330.

Anderson, J. P. (1972) *Computer security technology planning study. volume 2*.

Anderson, J. P. (1980) “Computer security threat monitoring and surveillance,” *Technical Report, James P. Anderson Company*.

Batista, G. E. *et al.* (2003) “Balancing Training Data for Automated Annotation of Keywords: a Case Study.,” in *WOB*, pp. 10–18.

Batista, G. E., Prati, R. C. and Monard, M. C. (2004) “A study of the behavior of several methods for balancing machine learning training data,” *ACM SIGKDD explorations newsletter*. ACM, 6(1), pp. 20–29.

Beek, C. *et al.* (2019) *McAfee Labs Threats Report August 2019*. Available at: https://www.mcafee.com/enterprise/en-us/threat-center/mcafee-labs/reports.html.

Bergstra, J. S. *et al.* (2011) “Algorithms for hyper-parameter optimization,” in *Advances in neural information processing systems*, pp. 2546–2554.

Bergstra, J., Yamins, D. and Cox, D. D. (2013) “Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms,” in *Proceedings of the 12th Python in science conference*, pp. 13–20.

Bhuyan, M. H., Bhattacharyya, D. K. and Kalita, J. K. (2013) “Network Anomaly Detection: Methods, Systems and Tools,” *IEEE Communications Surveys & Tutorials*. IEEE, 16(1), pp. 303–336. doi: 10.1109/SURV.2013.052213.00046.

Bodon, F. and Buza, K. (2014) *Adatbányászati algoritmusok*.

Bouzida, Y. *et al.* (2004) “Efficient intrusion detection using principal component analysis,” in *3éme Conférence sur la Sécurité et Architectures Réseaux (SAR), La Londe, France*, pp. 381–395.

Breiman, L. *et al.* (1984) *Classification and Regression Trees*. 1st ed. New York: Routledge.

Brochu, E., Cora, V. M. and De Freitas, N. (2010) “A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning,” *arXiv preprint arXiv:1012.2599*.

Brownlee, J. (2015) *8 Tactics to Combat Imbalanced Classes in Your Machine Learning Dataset*. Available at: https://machinelearningmastery.com/tactics-to-combat-imbalanced-classes-in-your-machine-learning-dataset/ (Accessed: March 31, 2018).

Bruneau, G. (2001) “The history and evolution of intrusion detection,” *SANS Institute*, 1.

Brunner, C. (2017) “Processing Intrusion Data with Machine Learning and MapReduce,” *Academic and Applied Research in Public Management Science*, 16(1), pp. 37–52.

Brunner, C. (2019) “A comparative study of Antminer+ and Decision Tree classification performances,” *SEFBIS Journal*, 13, pp. 15–23.

Buczak, A. L. and Guven, E. (2015) “A survey of data mining and machine learning methods for cyber security intrusion detection,” *IEEE Communications Surveys & Tutorials*. IEEE, 18(2), pp. 1153–1176. doi: 10.1109/COMST.2015.2494502.

Budzik, J. (2019) *Many Heads Are Better Than One: The Case For Ensemble Learning*. Available at: https://www.kdnuggets.com/2019/09/ensemble-learning.html (Accessed: September 29, 2019).

Cavusoglu, Ü. (2019) “A new hybrid approach for intrusion detection using machine learning methods,” *Applied Intelligence*. Springer, 49(7), pp. 2735–2761.

Chapman, P. *et al.* (2000) “CRISP-DM 1.0: Step-by-step data mining guide,” *SPSS inc*, 16.

Chawla, N. V *et al.* (2002) “SMOTE: Synthetic Minority Over-sampling Technique,” *Journal of Artificial Intelligence Rearch*, 16, pp. 321–357.

Chebrolu, S., Abraham, A. and Thomas, J. P. (2005) “Feature deduction and ensemble design of intrusion detection systems,” *Computers & security*. Elsevier, 24(4), pp. 295–307.

Chollet, F. (2015) *KERAS Documentation*. Available at: https://keras.io (Accessed: September 15, 2019).

Cortes, C. and Vapnik, V. (1995) “Support-vector networks,” *Machine learning*. Springer, 20(3), pp. 273–297.

Damiani, J. (2019) *A Voice Deepfake Was Used To Scam A CEO Out Of $243,000*, *Forbes*. Available at: https://www.forbes.com/sites/jessedamiani/2019/09/03/a-voice-deepfake-was-used-to-scam-a-ceo-out-of-243000/#3efedc652241 (Accessed: June 7, 2020).

Divekar, A. *et al.* (2018) “Benchmarking datasets for anomaly-based network intrusion detection: KDD CUP 99 alternatives,” in *2018 IEEE 3rd International Conference on Computing, Communication and Security (ICCCS)*, pp. 1–8.

Dua, S. and Du, X. (2016) *Data mining and machine learning in cybersecurity*. CRC press.

Elhag, S. *et al.* (2015) “On the combination of genetic fuzzy systems and pairwise learning for improving detection rates on Intrusion Detection Systems,” *Expert Systems with Applications*. Elsevier Ltd, 42(1), pp. 197–202. doi: 10.1016/j.eswa.2014.08.002.

Ester, M. *et al.* (1996) “A density-based algorithm for discovering clusters in large spatial databases with noise.,” in *Kdd*, pp. 226–231.

Fayyad, U., Piatetsky-Shapiro, G. and Smyth, P. (1996) “From data mining to knowledge discovery in databases,” *AI magazine*, 17(3), p. 37.

Folino, G., Pizzuti, C. and Spezzano, G. (2005) “GP ensemble for distributed intrusion detection systems,” in *International Conference on Pattern Recognition and Image Analysis*, pp. 54–62.

Google Brain Team (2015) *TensorFlow.org*. Available at: https://www.tensorflow.org/ (Accessed: September 15, 2019).

Han, J., Kamber, M. and Pei, J. (2011) *Data mining: concepts and techniques*. Elsevier Ltd.

Hasan, M. A. M. *et al.* (2016) “Performance evaluation of different kernels for support vector machine used in intrusion detection system,” *International Journal of Computer Networks and Communications*, 8(6), pp. 39–54.

Ingre, B., Yadav, A. and Soni, A. K. (2017) “Decision tree based intrusion detection system for NSL-KDD dataset,” in *International Conference on Information and Communication Technology for Intelligent Systems*, pp. 207–218.

Ippoliti, D. (2011) *Ph. D. Thesis Proposal Automated Network Anomaly Detection with Learning and QoS Mitigation*.

Ippoliti, D. (2013) “Automated network anomaly detection with learning, control and mitigation,” p. 196. Available at: http://gradworks.umi.com/36/07/3607573.html.

Javaid, A. *et al.* (2016) “A deep learning approach for network intrusion detection system,” in *Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (formerly BIONETICS)*, pp. 21–26.

Kevric, J., Jukic, S. and Subasi, A. (2017) “An effective combining classifier approach using tree algorithms for network intrusion detection,” *Neural Computing and Applications*. Springer, 28(1), pp. 1051–1058. doi: 10.1007/s00521-016-2418-1.

Kim, G., Lee, S. and Kim, S. (2014) “A novel hybrid intrusion detection method integrating anomaly detection with misuse detection,” *Expert Systems with Applications*. Elsevier, 41(4), pp. 1690–1700.

Kingma, D. P. and Ba, J. (2014) “Adam: A Method for Stochastic Optimization,” *arXiv preprint arXiv:1412.6980*. Available at: http://arxiv.org/abs/1412.6980.

Kingma, D. P. and Welling, M. (2013) “Auto-encoding variational bayes,” *arXiv preprint arXiv:1312.6114*.

Kovács, T. and Kő, A. (2018) “Termelési hálózatok gyárainak összesített teljesítménymérése többváltozós döntési modellek alkalmazásával,” *Vezetéstudomány / Budapest Management Review*, 49(4), pp. 32–43. doi: 10.14267/VEZTUD.2018.04.04.

Latah, M. and Toker, L. (2018) “Towards an efficient anomaly-based intrusion detection for software-defined networks,” *IET networks*. IET, 7(6), pp. 453–459.

Lau, F. *et al.* (2000) “Distributed denial of service attacks,” in *2000 IEEE international conference on systems, man and cybernetics.*, pp. 2275–2280.

Lind, E. and Pantigoso Velasquez, Ä. (2019) “A performance comparison between CPU and GPU in TensorFlow.” Stockholm.

Lopez-Martin, M., Carro, B. and Sanchez-Esguevillas, A. (2019) “Variational data generative model for intrusion detection,” *Knowledge and Information Systems*. Springer, 60(1), pp. 569–590.

Lutins, E. (2017) *DBSCAN: What is it? When to Use it? How to use it.* Available at: https://medium.com/@elutins/dbscan-what-is-it-when-to-use-it-how-to-use-it-8bd506293818 (Accessed: August 10, 2019).

Mahfouz, A. M., Venugopal, D. and Shiva, S. G. (2020) “Comparative Analysis of ML Classifiers for Network Intrusion Detection,” in *Fourth International Congress on Information and Communication Technology*, pp. 193–207.

McCulloch, W. S. and Pitts, W. (1943) “A logical calculus of the ideas immanent in nervous activity,” *The bulletin of mathematical biophysics*. Springer, 5(4), pp. 115–133.

McHugh, J. (2000) “Testing Intrusion detection systems: a critique of the 1998 and 1999 DARPA intrusion detection system evaluations as performed by Lincoln Laboratory,” *ACM Transactions on Information and System Security*, 3(4), pp. 262–294. doi: 10.1145/382912.382923.

Molina-Coronado, B. *et al.* (2020) “Survey of Network Intrusion Detection Methods from the Perspective of the Knowledge Discovery in Databases Process,” *arXiv preprint arXiv:2001.09697*.

Moustafa, N. and Slay, J. (2015) “UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set),” in *2015 Military Communications and Information Systems Conference (MilCIS)*, pp. 1–6.

Mukkamala, S., Sung, A. H. and Abraham, A. (2005) “Intrusion detection using an ensemble of intelligent paradigms,” *Journal of Network and Computer Applications*. Elsevier, 28(2), pp. 167–182. doi: 10.1016/j.jnca.2004.01.003.

Navlani, A. (2018) *KNN Classification using Scikit-learn*. Available at: https://www.datacamp.com/community/tutorials/k-nearest-neighbor-classification-scikit-learn (Accessed: August 8, 2019).

Nemati, H. R. and Barko, C. D. (2001) “Issues in organizational data mining: A survey of current practices,” *Journal of data warehousing*. THE DATA WAREHOUSE INSTITUTE, 6(1), pp. 25–36.

Ng, A. and others (2011) “Sparse autoencoder,” *CS294A Lecture notes*, 72(2011), pp. 1–19.

Nguyen, H. M., Cooper, E. W. and Kamei, K. (2009) “Borderline over-sampling for imbalanced data classification,” in *Proceedings: Fifth International Workshop on Computational Intelligence & Applications*, pp. 24–29.

Parampottupadam, S. and Moldovann, A.-N. (2018) “Cloud-based Real-time Network Intrusion Detection Using Deep Learning,” in *2018 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*, pp. 1–8.

Parsaei, M. R., Rostami, S. M. and Javidan, R. (2016) “A hybrid data mining approach for intrusion detection on imbalanced NSL-KDD dataset,” *International Journal of Advanced Computer Science and Applications*, 7(6), pp. 20–25.

Pedregosa, F. *et al.* (2011) “Scikit-learn: Machine Learning in Python,” *Journal of Machine Learning Research*, 12, pp. 2825–2830.

Petersen, R. (2015) *Data Mining for Network Intrusion Detection - A comparison of data mining algorithms and an analysis of relevant features for detecting cyber-attacks*. Mittuniversitetet - Mid Sweden University.

Piech, C. (2012) *K Means*. Available at: https://stanford.edu/~cpiech/cs221/handouts/kmeans.html (Accessed: August 10, 2019).

Protić, D. D. (2018) “Review of KDD Cup’99, NSL-KDD and Kyoto 2006+ datasets,” *Vojnotehnički glasnik*, 66(3), pp. 580–596.

Quinlan, J. R. (1986) “Induction of decision trees,” *Machine learning*. Springer, 1(1), pp. 81–106.

Rumelhart, D. E. *et al.* (1988) “Learning representations by back-propagating errors,” *Cognitive modeling*, 5(3), p. 1.

Russel, S. J. and Norwig, P. (2010) *Artificial Intelligence A modern approach*. 3rd ed. Prentice Hall.

Sakr, M. M., Tawfeeq, M. A. and El-Sisi, A. B. (2019) “Network Intrusion Detection System based PSO-SVM for Cloud Computing,” *International Journal of Computer Network and Information Security*. Modern Education and Computer Science Press, 11(3), p. 22.

Samuel, A. L. (1959) “Some Studies in Machine Learning Using the Game of Checkers,” *IBM Journal of Research and Development*, 3(3), pp. 210–229.

Saporito, G. (2019) *A Deeper Dive into the NSL-KDD Data Set*. Available at: https://towardsdatascience.com/a-deeper-dive-into-the-nsl-kdd-data-set-15c753364657.

Sapre, S., Ahmadi, P. and Islam, K. (2019) “A Robust Comparison of the KDDCup99 and NSL-KDD IoT Network Intrusion Detection Datasets Through Various Machine Learning Algorithms,” *arXiv preprint arXiv:1912.13204*.

Scarfone, K. and Mell, P. (2007) “Guide to Intrusion Detection and Prevention Systems (IDPS) Recommendations of the National Institute of Standards and Technology,” *Nist Special Publication*, 800–94, p. 127. doi: 10.6028/NIST.SP.800-94.

Schölkopf, B. *et al.* (2000) “Support vector method for novelty detection,” in *Advances in neural information processing systems*, pp. 582–588.

scikit-learn developers (2018) “scikit-learn user guide,” p. 2377. Available at: https://scikit-learn.org/stable/index.html.

Sharafaldin, I., Lashkari, A. H. and Ghorbani, A. A. (2018) “Toward generating a new intrusion detection dataset and intrusion traffic characterization.,” in *ICISSP*, pp. 108–116.

Sharda, R., Delen, D. and Turban, E. (2018) *Business Intelligence, Analytics, and Data Science: A managerial perspective*. 4th ed. Pearson.

Smith, S. (2014) *5 Famous Botnets that held the internet hostage*. Available at: https://tqaweekly.com/episodes/season5/tqa-se5ep11.php (Accessed: June 7, 2020).

Smolyakov, V. (2017) *Ensemble Learning to Improve Machine Learning Results*. Aug. Available at: https://blog.statsbot.co/ensemble-learning-d1dcd548e936 (Accessed: March 20, 2019).

Snoek, J., Larochelle, H. and Adams, R. P. (2012) "Practical bayesian optimization of machine learning algorithms," in *Advances in neural information processing systems*, pp. 2951–2959.

So-In, C. *et al.* (2014) "An evaluation of data mining classification models for network intrusion detection," *2014 Fourth International Conference on Digital Information and Communication Technology and its Applications (DICTAP)*, pp. 90–94. doi: 10.1109/DICTAP.2014.6821663.

Sohn, K., Lee, H. and Yan, X. (2015) "Learning structured output representation using deep conditional generative models," in *Advances in neural information processing systems*, pp. 3483–3491.

Song, J. *et al.* (2011) "Statistical analysis of honeypot data and building of Kyoto 2006+ dataset for NIDS evaluation," in *Proceedings of the first workshop on building analysis datasets and gathering experience returns for security*, pp. 29–36.

Srivastava, N. *et al.* (2014) "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research*, 15(56), pp. 1929–1958. Available at: http://jmlr.org/papers/v15/srivastava14a.html.

Statt, N. (2019) *Thieves are now using AI deepfakes to trick companies into sending them money*, *The Verge*. Available at: https://www.theverge.com/2019/9/5/20851248/deepfakes-ai-fake-audio-phone-calls-thieves-trick-companies-stealing-money (Accessed: June 7, 2020).

Stolfo, S. J. *et al.* (2000) "Cost-based Modeling for Fraud and Intrusion Detection: Results from the JAM Project," in *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*. IEEE, pp. 130–144.

Tavallaee, M. *et al.* (2009) "A Detailed Analysis of the KDD CUP 99 Data Set," in *IEEE Symposium on Computational Intelligence for Security and Defense Applications - CISDA*. IEEE, pp. 1–6.

Tian, D., Liu, Y. and Xiang, Y. (2009) "Large-scale network intrusion detection based on distributed learning algorithm," *International Journal of Information Security*. Springer, 8(1), pp. 25–35.

Tomek, I. (1976) "Two modifications of CNN.," *IEEE transactions on Systems, Man, and Cybernetics*, 6(11), pp. 769–772.

Tsai, C. F. *et al.* (2009) "Intrusion detection by machine learning: A review," *Expert Systems with Applications*. Elsevier Ltd, 36(10), pp. 11994–12000. doi: 10.1016/j.eswa.2009.05.029.

Wieringa, R. J. (2014) *Design science methodology for information systems and software engineering*. Springer.

Wilson, D. L. (1972) "Asymptotic properties of nearest neighbor rules using edited data," *IEEE Transactions on Systems, Man, and Cybernetics*. IEEE, (3), pp. 408–421.

Yang, Yanqing *et al.* (2019) "Improving the classification effectiveness of intrusion detection by using improved conditional variational autoencoder and deep neural network," *Sensors*. Multidisciplinary Digital Publishing Institute, 19(11), p. 2528.

Yao, H. *et al.* (2017) "An Intrusion Detection Framework Based on Hybrid Multi-Level Data Mining," *International Journal of Parallel Programming*. Springer US, pp. 1–19. doi: 10.1007/s10766-017-0537-7.

Yao, Y. Y., Zhao, Y. and Maguire, R. B. (2003) "Explanation-oriented association mining using a combination of unsupervised and supervised learning algorithms," in *Conference of the Canadian Society for Computational Studies of Intelligence*, pp. 527–532.

Yin, C. *et al.* (2017) "A deep learning approach for intrusion detection using recurrent neural networks," *Ieee Access*. IEEE, 5, pp. 21954–21961.

Zhang, J. and Zulkernine, M. (2006) "A hybrid network intrusion detection technique using random forests," in *The First International Conference on Availability, Reliability and Security*. Vienna: IEEE, pp. 1–8. doi: 10.1109/ARES.2006.7.

Zhang, J., Zulkernine, M. and Haque, A. (2008) "Random-Forests-Based Network Intrusion Detection

Systems," *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 38(5), pp. 649–659. doi: 10.1109/TSMCC.2008.923876.

8. PUBLICATIONS

Journal articles

Brunner, C. (2019): A comparative study of Antminer+ and Decision Tree classification performances. *SEFBIS*, issue 13, pp. 15-23.

Brunner, C. (2017): Processing Intrusion Data with Machine Learning and MapReduce. *ACADEMIC AND APPLIED RESEARCH IN MILITARY AND PUBLIC MANAGEMENT SCIENCE*, issue 16, pp. 37-52.

Abstracts in conference proceedings

Brunner, C. (2019): Behatolás-detektálás Tensorflow Platformon. *16. Országos Gazdaságinformatikai Konferencia* (p. 32). Budapest: NJSZT Neumann János Számítógép-tudományi Társaság GIKOF Gazdaságinformatikai Kutatási és Oktatási Fórum.

Brunner, C. (2018): Hálózati behatolás detektálás Neurális hálózatok összevonásával. *OGIK'2018 Országos Gazdaságinformatikai Konferencia – Az előadások összefoglalói* (p. 75). Sopron, Alexander Alapítvány a Jövő Értelmiségéért.

Brunner, C. (2017): Ant Colony Algorithm in Data Mining. *XIV. Országos GazdaságInformatikai Konferencia* (p. 31). Győr: Alexander Alapítvány a Jövő Értelmiségéért.

Brunner, C. (2016): Behatolási adatok feldolgozása gépi tanulás és MapReduce segítségével. *XIII. Országos Gazdaságinformatikai Konferencia* (pp. 25-26). Dunaújváros: DUE Press.

Conference presentations

Brunner C. (2019): Behatolás-detektálás Tensorflow Platformon- Presented at: *16. Országos Gazdaságinformatikai Konferencia;* Nov 8-9; Budapest, Hungary

Brunner, C. (2018): Hálózati behatolás detektálás Neurális hálózatok összevonásával. Presented at: *OGIK'2018 Országos Gazdaságinformatikai Konferencia;* Nov 9-10; Sopron, Hungary

Brunner, C. (2017): Ant Colony Algorithm in Data Mining. Poster presented at: *XIV. Országos GazdaságInformatikai Konferencia*; Nov 10-11; Sopron, Hungary.

Brunner, C. (2016): Behatolási adatok feldolgozása gépi tanulás és MapReduce segítségével. Presented at: *XIII. Országos GazdaságInformatikai Konferncia*; Nov 11-12; Dunaújváros, Hungary.

Other works

Brunner, C. (2019): Hálózati behatolás detektálás Neurális hálózatok összevonásával. In L. Bacsárdi, G. Bencsik, & Z. Pödör, *OGIK'2018 Országos Gazdaságinformatikai Konferencia - Válogatott közlemények*. Sopron, Hungary: Alexander Alapítvány a Jövő Értelmiségéért.

In preparation

Brunner, C., Kő, A., & Fodor, S. (2020): A novel ensemble method based on Neural Networks with hyperparameter optimization for Intrusion Detection. Manuscript

9. APPENDIX

Appendix A: As part of this dissertation, 10 independent articles were collected to verify which detailed class belongs to which attack category. The exact articles and the evaluation are visible here. After data collection, the relative frequencies of each attack category have been calculated. The attack category with the most "votes" became the final category for a given detailed attack class used later as a data preparation step prior to model training.

Appendix B: Detailed performance measurements for the decision tree bagging classifier (V0 experiment). This intrusion detector was implemented using the map-reduce programming paradigm coded using the implementation of the Message Passing Interface in Java. Runtime considerations of this parallelization approach, however, do not affect classification performance, therefore measurements between parallel setups and sample sizes have been aggregated.

Appendix C & Appendix D: Detailed performance measurements for intrinsic, time-traffic, host-traffic, content and aggregator models of the V1 models measured with KDD Cup 1999 and NSL-KDD test datasets. Each of the base models performed poorly on different classes of the test set. Results were improved by the final or aggregator model.

Appendix E, Appendix F & Appendix G: The V2 models were repeated with three distinct synthetic sampling processes as variants, the base and aggregator model confusion matrices have been provided with the intention to be used for calculating further performance metrics when considered necessary.

Appendix H: confusion matrices for the V3 model. A key element of this was the application of autoencoder networks, a kind of neural network designed to learn an internal representation of the input space. Therefore, the inputs to the stacking neural network were comprised of the per-feature difference between the original feature values and those predicted by autoencoders trained on normal traffic only.

Detailed Class	**Train/ Test**	(Ingre, Yadav and Soni, 2017)	(Sakr, Tawfeeq and El-Sisi, 2019)	(Mahfouz, Venugopal and Shiva, 2020)	(Yang *et al.*, 2019)	(Latah and Toker, 2018)	(Aghdam and Kabiri, 2016)	(Saporito, 2019)	(Hasan *et al.*, 2016)	(Parampottupadam and Moldovann, 2018)	(Protić, 2018)	**dos**	**probe**	**r2l**	**u2r**	**N/A**	**Result**
apache2	Test	dos	dos	dos	dos	dos	dos	dos	dos	dos	dos	100%	0%	0%	0%	0%	dos
back	Train	dos	dos	dos	dos	dos	dos	dos	dos	dos	dos	100%	0%	0%	0%	0%	dos
buffer_overflow	Train	u2r	u2r	u2r	u2r	u2r	u2r	u2r	u2r	u2r	u2r	0%	0%	0%	100%	0%	u2r
ftp_write	Train	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	0%	0%	100%	0%	0%	r2l
guess_passwd	Train	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	0%	0%	100%	0%	0%	r2l
httptunnel	Test	r2l	r2l	r2l	u2r	u2r	u2r	r2l	r2l	r2l	r2l	0%	0%	70%	30%	0%	r2l
imap	Train	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	0%	0%	100%	0%	0%	r2l
ipsweep	Train	probe	probe	probe	probe	probe	probe	probe	probe	probe	probe	0%	100%	0%	0%	0%	probe
land	Train	dos	dos	dos	dos	dos	dos	dos	dos	dos	dos	100%	0%	0%	0%	0%	dos
loadmodule	Train	u2r	u2r	u2r	u2r	u2r	u2r	u2r	u2r	u2r	u2r	0%	0%	0%	100%	0%	u2r
mailbomb	Test	dos	N/A	N/A	dos	dos	dos	dos	dos	dos	dos	80%	0%	0%	0%	20%	dos
mscan	Test	probe	probe	probe	probe	probe	probe	probe	probe	probe	probe	0%	100%	0%	0%	0%	probe
multihop	Train	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	0%	0%	100%	0%	0%	r2l
named	Test	r2l	r2l	N/A	r2l	r2l	r2l	r2l	r2l	r2l	r2l	0%	0%	90%	0%	10%	r2l
neptune	Train	dos	dos	dos	dos	dos	dos	dos	dos	dos	dos	100%	0%	0%	0%	0%	dos
nmap	Train	probe	probe	probe	probe	probe	probe	probe	probe	probe	probe	0%	100%	0%	0%	0%	probe
perl	Train	u2r	u2r	u2r	u2r	u2r	u2r	u2r	u2r	u2r	u2r	0%	0%	0%	100%	0%	u2r
phf	Train	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	0%	0%	100%	0%	0%	r2l
pod	Train	dos	dos	dos	dos	dos	dos	dos	dos	dos	dos	100%	0%	0%	0%	0%	dos
portsweep	Train	probe	probe	probe	probe	probe	probe	probe	probe	probe	probe	0%	100%	0%	0%	0%	probe
processtable	Test	dos	dos	dos	dos	dos	dos	dos	dos	dos	dos	100%	0%	0%	0%	0%	dos
ps	Test	u2r	u2r	u2r	u2r	u2r	u2r	u2r	u2r	u2r	u2r	0%	0%	0%	100%	0%	u2r
rootkit	Train	u2r	u2r	u2r	u2r	u2r	u2r	u2r	u2r	u2r	u2r	0%	0%	0%	100%	0%	u2r

Detailed Class	**Train/ Test**	(Ingre, Yadav and Soni, 2017)	(Sakr, Tawfeeq and El-Sisi, 2019)	(Mahfouz, Venugopal and Shiva, 2020)	(Yang *et al.*, 2019)	(Latah and Toker, 2018)	(Aghdam and Kabiri, 2016)	(Saporito, 2019)	(Hasan *et al.*, 2016)	(Parampottupadam and Moldovann, 2018)	(Protić, 2018)	**dos**	**probe**	**r2l**	**u2r**	**N/A**	**Result**
saint	Test	probe	probe	probe	probe	probe	probe	probe	probe	probe	probe	0%	100%	0%	0%	0%	probe
satan	Train	probe	probe	probe	probe	probe	probe	probe	probe	probe	probe	0%	100%	0%	0%	0%	probe
sendmail	Test	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	0%	0%	100%	0%	0%	r2l
smurf	Train	dos	dos	dos	dos	dos	dos	dos	dos	dos	dos	100%	0%	0%	0%	0%	dos
snmpgetattack	Test	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	0%	0%	100%	0%	0%	r2l
snmpguess	Test	u2r	r2l	r2l	r2l	r2l	u2r	r2l	r2l	r2l	r2l	0%	0%	80%	20%	0%	r2l
spy	Train	r2l	r2l	r2l	r2l	r2l	u2r	r2l	r2l	r2l	r2l	0%	0%	90%	10%	0%	r2l
sqlattack	Test	u2r	u2r	u2r	u2r	u2r	u2r	u2r	u2r	u2r	u2r	0%	0%	0%	100%	0%	u2r
teardrop	Train	dos	dos	dos	dos	dos	dos	dos	dos	dos	dos	100%	0%	0%	0%	0%	dos
udpstorm	Test	dos	dos	dos	dos	dos	dos	dos	dos	dos	dos	100%	0%	0%	0%	0%	dos
warezclient	Train	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	0%	0%	100%	0%	0%	r2l
warezmaster	Train	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	0%	0%	100%	0%	0%	r2l
worm	Test	u2r	dos	dos	r2l	r2l	u2r	dos	r2l	dos	dos	50%	0%	30%	20%	0%	dos
xlock	Test	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	0%	0%	100%	0%	0%	r2l
xsnoop	Test	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	r2l	0%	0%	100%	0%	0%	r2l
xterm	Test	u2r	u2r	u2r	u2r	u2r	u2r	u2r	u2r	u2r	u2r	0%	0%	0%	100%	0%	u2r

Appendix A: The data used to create a conceptual hierarchy, which in turn is used later to recategorize attacks in the KDD Cup 1999 and NSL-KDD datasets

4 cores	**Small sample (3-5 000 obs.)**				**Large sample (6-10 000 obs.)**			
	1. run	**2. run**	**3. run**	**1p2c**	**1. run**	**2. run**	**3. run**	**1p2c**
Accuracy	0.785	0.791	0.772	*0.796*	0.795	0.792	0.809	*0.799*
Precision	0.959	0.975	0.866	*0.967*	0.965	0.902	0.903	*0.969*
Recall	0.483	0.490	0.508	*0.506*	0.507	0.539	0.585	*0.513*
F-score	0.642	0.652	0.641	*0.664*	0.664	0.675	0.710	*0.671*
AUC	0.735	0.766	0.783	*0.793*	0.811	0.789	0.776	*0.777*

8 cores	**Small sample (3-5 000 obs.)**				**Large sample (6-10 000 obs.)**			
	1. run	**2. run**	**3. run**	**1p2c**	**1. run**	**2. run**	**3. run**	**1p2c**
Accuracy	0.793	0.782	0.788	*0.796*	0.797	0.756	0.777	*0.799*
Precision	0.893	0.931	0.903	*0.967*	0.936	0.895	0.881	*0.969*
Recall	0.546	0.493	0.525	*0.506*	0.529	0.442	0.512	*0.513*
F-score	0.678	0.644	0.664	*0.664*	0.676	0.592	0.648	*0.671*
AUC	0.789	0.719	0.784	*0.793*	0.772	0.771	0.757	*0.777*

4 cores	**Small sample (3-5 000 obs.)**				**Large sample (6-10 000 obs.)**			
	1. run	**2. run**	**3. run**	**1p2c**	**1. run**	**2. run**	**3. run**	**1p2c**
Accuracy	0.978	0.964	0.981	*0.984*	0.985	0.985	0.980	*0.987*
Precision	0.477	0.449	0.513	*0.511*	0.532	0.558	0.489	*0.576*
Recall	0.438	0.438	0.466	*0.525*	0.469	0.469	0.452	*0.507*
F-score	0.456	0.444	0.489	*0.518*	0.498	0.510	0.470	*0.539*

8 cores	**Small sample (3-5 000 obs.)**				**Large sample (6-10 000 obs.)**			
	1. run	**2. run**	**3. run**	**1p2c**	**1. run**	**2. run**	**3. run**	**1p2c**
Accuracy	0.970	0.977	0.976	*0.984*	0.981	0.981	0.980	*0.987*
Precision	0.397	0.467	0.476	*0.511*	0.513	0.445	0.470	*0.576*
Recall	0.421	0.437	0.441	*0.525*	0.470	0.439	0.438	*0.507*
F-score	0.408	0.452	0.458	*0.518*	0.490	0.442	0.453	*0.539*

Appendix B: Detailed performance measurements for the decision tree bagging classifier

	Intrinsic			Time-traffic			Host-traffic			Content			Aggregator		
	Recall	**Precision**	**F_1 score**	**Recall**	**Precision**	**F_1 score**	**Recall**	**Precision**	**F_1 score**	**Recall**	**Precision**	**F_1 score**	**Recall**	**Precision**	**F1 score**
Normal	0.971	0.744	0.842	0.664	0.504	0.573	0.956	0.728	0.827	0.700	0.817	0.754	0.978	0.720	0.830
DoS	0.791	0.998	0.883	0.826	0.997	0.903	0.964	0.994	0.979	0.000	0.000	0.000	0.961	0.997	0.979
Probe	0.702	0.067	0.122	0.730	0.733	0.731	0.669	0.733	0.699	0.966	0.016	0.031	0.727	0.664	0.694
R2L	0.074	0.555	0.131	0.000	0.000	0.000	0.058	0.308	0.098	0.071	0.493	0.123	0.088	0.585	0.152
U2R	0.800	0.016	0.032	0.957	0.002	0.004	0.329	0.012	0.023	0.614	0.339	0.437	0.571	0.165	0.256

Appendix C: Detailed performance measurements for intrinsic, time-traffic, host-traffic, content and aggregator models of the V1 detector on KDD Cup 1999

	Intrinsic			Time-traffic			Host-traffic			Content			Aggregator		
	Recall	**Precision**	**F_1 score**	**Recall**	**Precision**	**F_1 score**	**Recall**	**Precision**	**F_1 score**	**Recall**	**Precision**	**F_1 score**	**Recall**	**Precision**	**F_1 score**
Normal	0.857	0.801	0.828	0.736	0.792	0.763	0.928	0.696	0.795	0.750	0.809	0.778	0.945	0.726	0.821
DoS	0.547	0.847	0.665	0.723	0.899	0.801	0.728	0.856	0.787	0.889	0.530	0.664	0.813	0.906	0.857
Probe	0.547	0.276	0.367	0.770	0.728	0.748	0.549	0.651	0.596	0.000	0.000	0.000	0.690	0.710	0.700
R2L	0.374	0.583	0.456	0.815	0.483	0.607	0.204	0.807	0.325	0.181	0.846	0.298	0.240	0.947	0.383
U2R	0.552	0.054	0.098	0.000	0.000	0.000	0.164	0.023	0.040	0.448	0.070	0.121	0.104	0.053	0.071

Appendix D: Detailed performance measurements for intrinsic, time-traffic, host-traffic, content and aggregator models of the V1 detector on NSL-KDD

Intrinsic		Ground Truth				
		DoS	**Normal**	**Probe**	**R2L**	**U2R**
Prediction	**DoS**	4086	86	654	5	0
	Normal	1110	8319	30	808	0
	Probe	1973	879	1407	611	0
	R2L	289	368	119	1081	28
	U2R	2	59	211	380	39
Content		**Ground Truth**				
		DoS	**Normal**	**Probe**	**R2L**	**U2R**
Prediction	**DoS**	6315	2358	2286	1128	15
	Normal	792	7274	135	776	10
	Probe	0	0	0	0	0
	R2L	353	59	0	976	11
	U2R	0	20	0	5	31
Time-traffic		**Ground Truth**				
		DoS	**Normal**	**Probe**	**R2L**	**U2R**
Prediction	**DoS**	5316	368	15	0	0
	Normal	921	6902	308	316	1
	Probe	566	675	1922	195	1
	R2L	615	1750	175	2358	65
	U2R	42	16	1	16	0
Host-traffic		**Ground Truth**				
		DoS	**Normal**	**Probe**	**R2L**	**U2R**
Prediction	**DoS**	5344	515	598	31	7
	Normal	1751	8650	578	1718	26
	Probe	112	166	1206	175	3
	R2L	226	206	14	724	13
	U2R	27	174	25	237	18
Aggregator		**Ground Truth**				
		DoS	**Normal**	**Probe**	**R2L**	**U2R**
Prediction	**DoS**	6161	466	453	1	0
	Normal	1130	8988	702	1779	23
	Probe	78	192	1265	148	2
	R2L	89	50	1	940	17
	U2R	2	15	0	17	25

Appendix E: V2 with SMOTE ENN sampling confusion matrices

Intrinsic		Ground Truth				
		DoS	**Normal**	**Probe**	**R2L**	**U2R**
Prediction	**DoS**	4079	86	738	5	0
	Normal	671	8312	31	807	0
	Probe	1980	884	1322	611	0
	R2L	285	370	119	1082	28
	U2R	445	59	211	380	39
Content		**Ground Truth**				
		DoS	**Normal**	**Probe**	**R2L**	**U2R**
Prediction	**DoS**	0	1	0	31	0
	Normal	792	7275	135	777	11
	Probe	6315	2357	2286	1097	15
	R2L	353	57	0	976	10
	U2R	0	21	0	4	31
Time-traffic		**Ground Truth**				
		DoS	**Normal**	**Probe**	**R2L**	**U2R**
Prediction	**DoS**	5056	390	80	1	0
	Normal	1216	6861	340	330	1
	Probe	557	691	1877	175	1
	R2L	137	523	12	393	1
	U2R	494	1246	112	1986	64
Host-traffic		**Ground Truth**				
		DoS	**Normal**	**Probe**	**R2L**	**U2R**
Prediction	**DoS**	5521	516	766	28	5
	Normal	1611	8595	342	1654	20
	Probe	149	200	1273	229	5
	R2L	159	225	23	760	17
	U2R	20	175	17	214	20
Aggregator		**Ground Truth**				
		DoS	**Normal**	**Probe**	**R2L**	**U2R**
Prediction	**DoS**	6410	473	610	4	1
	Normal	949	8932	460	1757	28
	Probe	82	244	1351	149	1
	R2L	19	53	0	949	17
	U2R	0	9	0	26	20

Appendix F: V2 with SMOTE Tomek sampling confusion matrices

Intrinsic		Ground Truth				
		DoS	**Normal**	**Probe**	**R2L**	**U2R**
Prediction	**DoS**	4049	87	566	3	0
	Normal	796	8256	27	798	0
	Probe	2015	940	1500	622	0
	R2L	285	369	117	1081	28
	U2R	315	59	211	381	39
Content		**Ground Truth**				
		DoS	**Normal**	**Probe**	**R2L**	**U2R**
Prediction	**DoS**	6315	2357	2286	1111	15
	Normal	792	7224	135	765	3
	Probe	0	1	0	17	0
	R2L	353	112	0	984	31
	U2R	0	17	0	8	18
Time-traffic		**Ground Truth**				
		DoS	**Normal**	**Probe**	**R2L**	**U2R**
Prediction	**DoS**	5378	402	111	1	0
	Normal	969	7307	268	346	2
	Probe	491	217	1916	158	0
	R2L	612	1768	124	2354	65
	U2R	10	17	2	26	0
Host-traffic		**Ground Truth**				
		DoS	**Normal**	**Probe**	**R2L**	**U2R**
Prediction	**DoS**	5851	541	624	56	8
	Normal	1376	8422	512	1540	18
	Probe	96	252	1218	258	6
	R2L	117	315	25	820	17
	U2R	20	181	42	211	18
Aggregator		**Ground Truth**				
		DoS	**Normal**	**Probe**	**R2L**	**U2R**
Prediction	**DoS**	6295	470	450	2	0
	Normal	1108	8876	532	1834	25
	Probe	36	295	1439	136	1
	R2L	21	60	0	897	21
	U2R	0	10	0	16	20

Appendix G: V2 with SVM SMOTE sampling confusion matrices

Intrinsic		Ground Truth				
		DoS	**Normal**	**Probe**	**R2L**	**U2R**
Prediction	**DoS**	3181	111	443	4	0
	Normal	700	9379	392	1732	15
	Probe	2894	124	1263	153	0
	R2L	240	57	148	626	13
	U2R	445	40	175	370	39
Content		**Ground Truth**				
		DoS	**Normal**	**Probe**	**R2L**	**U2R**
Prediction	**DoS**	0	0	0	0	0
	Normal	1	2	0	1	2
	Probe	7459	9643	2421	1905	28
	R2L	0	50	0	972	9
	U2R	0	16	0	7	28
Time-traffic		**Ground Truth**				
		DoS	**Normal**	**Probe**	**R2L**	**U2R**
Prediction	**DoS**	5171	385	298	7	0
	Normal	183	147	19	3	0
	Probe	446	498	1493	107	0
	R2L	1649	8664	454	2768	67
	U2R	11	17	157	0	0
Host-traffic		**Ground Truth**				
		DoS	**Normal**	**Probe**	**R2L**	**U2R**
Prediction	**DoS**	1477	13	230	4	2
	Normal	736	5327	107	1595	16
	Probe	3381	274	1458	308	13
	R2L	741	73	58	40	6
	U2R	1125	4024	568	938	30
Aggregator		**Ground Truth**				
		DoS	**Normal**	**Probe**	**R2L**	**U2R**
Prediction	**DoS**	5765	72	293	10	0
	Normal	241	8125	127	914	18
	Probe	1065	1114	1872	663	2
	R2L	126	373	65	941	8
	U2R	263	27	64	357	39

Appendix H: V3 detector confusion matrices