Security assessment tool

The Lab

Loutfi Walid
De Loenen Lars
Hendrix Jonas

2022 - 2023

KdG
Karel de Grote
Hogeschool

Karel de Grote Hogeschool

Table of Contents

1 Introduction

We are going to build a security assessment tool with the following functionalities: vulnerability scanning, APT detection, malware analysis, and phishing detection. For some of these functionalities, namely APT detection, malware analysis, and phishing detection, we will be using Artificial Intelligence. The goal is to develop a powerful tool that can detect and report security threats before they can cause any damage. In the chapter "Cybersecurity in AI" we will further explore the challenges associated with using AI in cybersecurity and how to overcome them.

2 Artificial Intelligence

The use of artificial intelligence in cybersecurity has the potential to greatly improve threat detection and response times. AI can automate repetitive tasks and analyze vast amounts of data, allowing security teams to focus on more complex threats. However, there are also concerns about the potential for AI to be used maliciously, such as creating more sophisticated cyber-attacks. To effectively use AI in cybersecurity, it is important to address these concerns and implement appropriate safeguards. This research paper will examine the current state of AI in cybersecurity and explore the potential benefits and challenges of its use. Our focus will be on the application of machine learning in four key areas: detecting advanced persistent threats (APTs) in logs, digital footprinting through network listening, malware analysis and detection, and pattern recognition for phishing emails and malicious URLs.

Through our analysis, we aim to demonstrate the power and potential of AI in the field of cybersecurity and to provide insights into how organizations can leverage this technology to improve their security posture, while also ensuring regulatory compliance.

2.1 AI in cybersecurity

Artificial intelligence (AI) has been transforming various industries, including cybersecurity. With the exponential growth of technology and the increasing use of the internet, the need for advanced security measures has become paramount. In this chapter, we will delve into the role of AI in enhancing the field of cybersecurity. From detecting and preventing cyber-attacks to improving threat intelligence, we will explore how AI is being utilized to make the digital world a safer place. As cyber threats continue to evolve, AI has proven to be a valuable tool in addressing them effectively. In this chapter, we will examine the current state of AI in cybersecurity and its future potential, including challenges and ethical considerations. Get ready to learn about the exciting world of AI in cybersecurity!

2.1.1 Machine learning

Machine learning explores one of the most important branches of AI in the field of cybersecurity. Machine learning is a type of AI that allows computers to learn and improve their performance without explicit programming. It involves the use of algorithms that can process and analyze vast amounts of data to identify patterns, make predictions, and make decisions. In the context of cybersecurity, machine learning is being used to detect and respond to cyber-attacks in real-time, as well as to identify and prevent potential threats.
From intrusion detection systems to malware detection and classification, machine learning has a wide range of applications in cybersecurity. It is also used to analyze network traffic and user behavior to identify abnormal activity that could indicate a cyber-attack. Moreover, machine learning algorithms can be trained on large datasets of past cyber attacks to improve their accuracy and effectiveness in detecting future threats.

To help you understand machine learning more, we will compare it to traditional programming. Traditional programming is where we have a problem we like to solve, maybe that problem is to calculate some data about the value of the business or maybe it is to detect an activity if we are building a fitness app. Generally there are all kinds of problems we have to solve and what we use is rules and a programming language. Those rules will act on data and then they will give us back some answers. For example, a common thing in stock and in understanding finances is a thing called price over earnings ratio (PE ratio). This is where you have a piece of data, which is the current price of the stock. You have another piece of data which is the earnings of the stock. And then you can calculate P over E by writing code. This is a simple example, but this is generally how traditional programming works.

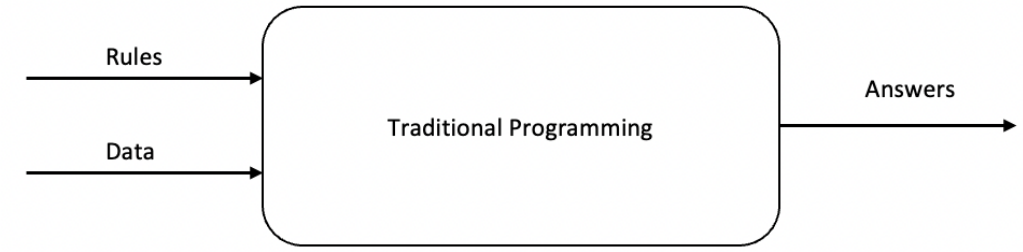

Figure 2.1.1 (1): schematic representation of traditional programming.

So when we talk about machine learning, we flip this around and look at it this way.

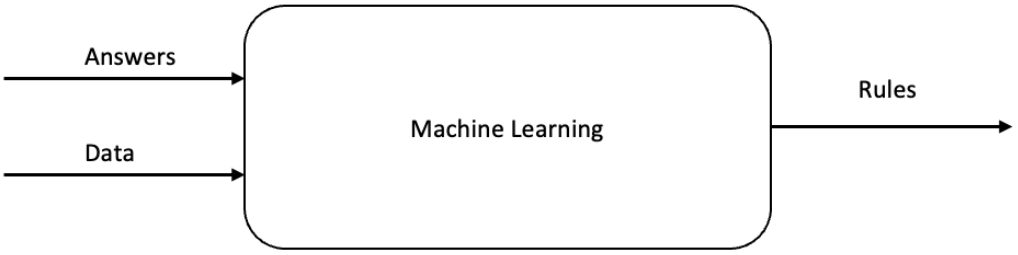

Figure 2.1.1 (2): schematic representation of machine learning.

With machine learning it is almost the same thing, but instead of us trying to create the rules ourselves in a programming language, we instead have a bunch of answers and we have a bunch of data. Then we will have a machine figure out how to map those answers to that data. When it does that, it will actually figure out the rules for us. Imagen we wanted to build the fitness app which we used as an example for traditional programming. We want to know if the user is running, if the user is walking, etc. The rules we would have to write to detect this would be very complex. So the idea behind machine learning is, if we have got all of this data, and we have labeled that data, this is what walking looks like, this is what running looks like, etc. Then we can have a machine, looking at patterns between the data of walking and the fact that we have said that this is walking. So figuring out what those patterns are and establishing a set of rules for that.

In the following chapters, we will delve into the process of training and optimizing our 3 AI models.

2.1.1.1 The detection of APTs

2.1.1.1.1 What are APTs?

Advanced persistent threats (APTs) are a type of cyber-attack that is highly sophisticated and targeted in nature. APTs are typically carried out by state-sponsored or well-funded hacker groups and are designed to remain undetected for an extended period of time, often for months or even years. The primary objective of APTs is to gain access to sensitive and valuable information, such as intellectual property, trade secrets, or financial information.

APTs are often multi-stage attacks that start with initial infiltration, followed by reconnaissance and data collection, and ultimately culminate in the exfiltration of sensitive data. These attacks can be difficult to detect because the attackers use a combination of tactics, including social engineering, zero-day vulnerabilities, and stealthy malware, to evade traditional security measures

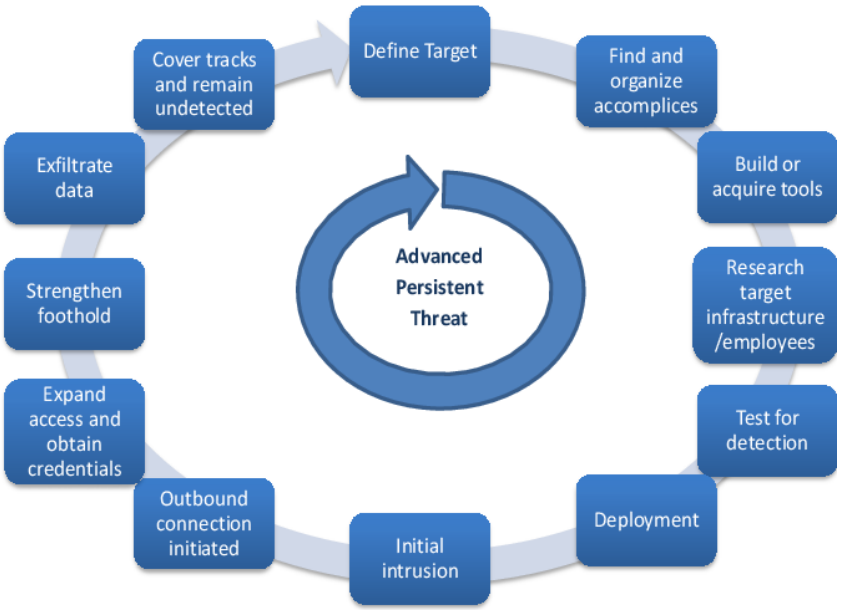

Figure 2.1.1.1.1: advanced persistent threat lifecycle.

2.1.1.1.2 How can machine learning help detect Advanced Persistent Threats?

Machine learning can help detect advanced persistent threats (APTs) by analyzing vast amounts of data generated by network activity, system logs, and other sources to identify patterns and distinguish between normal and abnormal behavior. Machine learning algorithms can be trained on examples of known APT behavior to recognize similar patterns in real-world data, and can also continuously learn and improve over time. This allows machine learning to detect APTs more effectively and in real-time, providing organizations with a valuable tool in the fight against APTs.

2.1.1.1.3 Data collection

Collecting data is a crucial step in the process of training a machine learning model. The data used to train a machine learning algorithm determines the accuracy and effectiveness of the model. A model trained on high-quality data will perform well on new, unseen data, while a model trained on low-quality or biased data can lead to inaccurate or unfair predictions. Now for detecting APTs, the creation of datasets to simulate advanced persistent threats is out of scope for this project. But no worries, we will simulate how we could have accomplished this to effectively detect APTs.
We have looked at existing datasets and researched if these datasets covered all the phases performed by an APT.

Dataset / APT Phase	UNB-15	CICIDS	NSL-KDD	Mawi	ISCX	DARPA	HERITRIX	DAPT 2020
Normal Traffic	✓	✓	✓	✓	✓	✓	✓	✓
Reconnaissance	✓	✓	✓		✓	✓		✓
Foothold Establishment		✓	✓	✓	✓	✓	✓	✓
Lateral Movement								✓
Data Exfiltration								✓

Table 2.1.1.1.3 (1): training datasets for detecting APTs.

Dataset / Attack	UNB-15	CICIDS 2018	CICIDS 2017	NSL-KDD	MAWI	ISCX	DARPA	HERITRIX	DAPT 2020
Network Scan	✓	✓	✓	✓	✓	✓	✓		✓
Web Vulnerability Scan	✓	✓							✓
Account bruteforce		✓	✓	✓	✓	✓	✓		✓
SQL injection		✓	✓	✓		✓			✓
Malware Download	✓		✓						✓
Backdoor	✓	✓				✓			✓
Command Injection	✓	✓	✓				✓	✓	✓
DoS	✓	✓	✓	✓	✓	✓	✓		✓
CSRF		✓	✓			✓			✓
Privilege escalation			✓		✓	✓		✓	✓

Table 2.1.1.1.3 (2): Comparison between attack vectors of each dataset in terms of different attack vectors that is involved in an APT attack. The table compares the existence of every attack vector by datasets UNB-15, CICIDS 2018, CICIDS 2017, NSL-KDD, MAWI, ISCX, DARPA , HERITRIX, and DAPT 2020.

We will have a deeper look into the DAPT 2020 dataset as this has promising results.This dataset is generated with the following infrastructure.

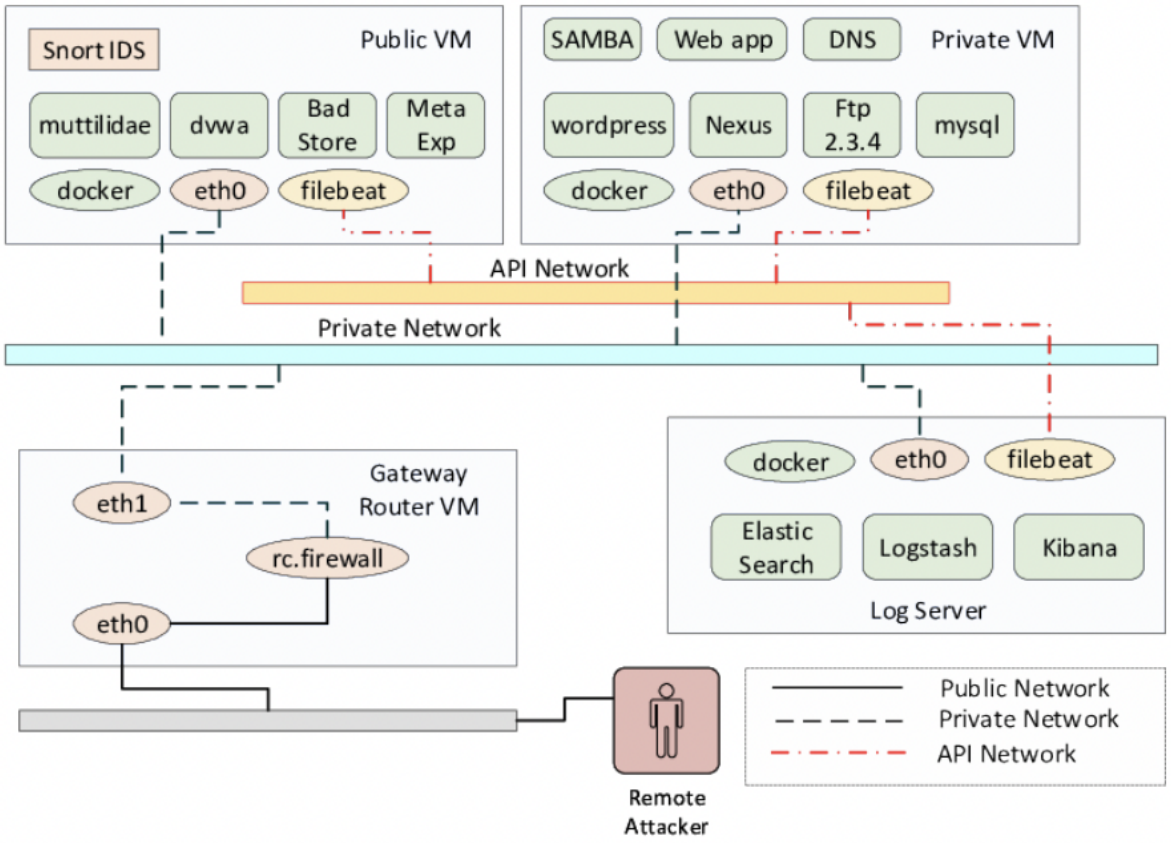

Figure 2.1.1.1.3 (3): System set used for construction of DAPT 2020 dataset. The attacker can access only public services exposed via firewall. Log Server (ELK Cluster) is used for collection of network and host logs.

This infrastructure setup generates network traffic logs and system logs that reflect real-world scenarios of APT attacks on enterprise networks. The collected data was then processed and labeled as either benign or malicious, and used to create the DAPT 2020 dataset for evaluating APT detection systems.

The setup consisted of VMWare ESXi physical servers that hosted virtual machines (VMs) with different services typical of an enterprise cloud network. A public VM was created with vulnerable services such as Mutillidae, Damn Vulnerable Web Application (DVWA), Metasploitable, and BadStore. Snort, a Network-based Intrusion Detection System (NID), was used to monitor for malicious traffic signatures. Each service was hosted as a separate Docker container.

The private VM was used to host services such as Samba, a Wordpress website, FTP, MySQL, and Nexus (repository management). The private and public VMs were connected over a private network. Both VMs had the capability to capture packets and logs. The logs were stored and filtered using the ELK stack-based log server. Network and host logs were periodically shipped to the Log Server using a filebeat agent.

In order to simulate real-world traffic, a group of users were given user and administrative credentials to access both public and private services within the network. These users performed typical business operations, such as updating a WordPress website, organizing files and folders, and managing users. This helped to generate a realistic pattern of network traffic, including both benign and malicious activities, which was used to create the DAPT 2020 dataset for APT detection.

The data was gathered over a duration of 5 days, where each day can be considered analogous to 3 months in a real-world scenario:

- **Monday (day 1)**:
  On this day, all the systems in the network have normal behavior, thus contributing to the collection of the normal data set.

- **Tuesday (day 2):**
  On this day, the APT attack is started with the first phase of the cyber kill chain which is Reconnaissance.The used reconnaissance mechanisms are:
  - *Network Scanning using NMap*
  - *Application Scanning using BurpSuit and/or WebScarab*
  - *Manual application accounts discovery*
  - *Performed dirbuster attack on different URLs*
  - *Performed nitko scan attack*
  - *Performed sqlmap scanning attack*

  The above reconnaissance activity can only be seen in the public network traffic. The private network (192.168.3.30) has no attack traffic on this day.

- **Wednesday (day 3):**
  On this day, the APT attack progresses to establishing a foothold. The attacks performed are:
  - *Downloading and Installing malicious software that gives remote control of the system*
  - *Performed CSRF on 9002*
  - *Performed SQL Injection on 9002*
  - *Php reverse shells.*
  - *Sending confirmation to C&C*

- **Thursday (day 4):**
  On this day, the APT attack progresses to the Lateral Movement stage, encompassing:
  - *SSH from local systems to 192.168.3.30*
  - *Nmap scan on 192.168.3.30*
  - *FTP using weak authentication bypass*
  - *MySQL bypass CVE-2012-2122*
  - *From 192.168.3.29 or Kali Linux run command below*
  - *$ for i in `seq 1 1000`; do mysql -uroot -pwrong -h <192.168.3.30> -P3306 ; done*
  - *DNS Zone Transfer Exploit from "192.168.3.29"*
  - *SMB CVE-2017-7494, Command shell*
  - *Accessed /etc/passwd file, and got a list of all users*
  - *Accessed /etc/shadow file to see the hashes*
  - *Created a user hacker/aimlsec0718 later updated the password to password*
  - *Created another user invader/hacker and added the user as sudoer, root privileges.*
  - *Established the backdoor by uploading shell.php (Ankur's previous day foothold establishment), and listening on port 4444. Shell obtained from 192.168.3.29.*
  - *Command Injection performed on 9002 at 9:05 PM*

- **Friday (day 5):**
  On this day, the APT attack progresses to the Data Exfiltration stage. This includes:
  - *Command Injection at 6:44 PM*
  - *Exported the /etc/passwd file to 206.207.50.50 C&C server*
  - *Stopped apache2 service on 9002, and the 9002 site is no longer available. DoS performed*
  - *Use of PyExfil*
  - *Uploaded more files using scp to 206.207.50.50*

Now, we will dive into the normalization and labeling process of the training dataset.

2.1.1.1.4 Preprocessing the data

After collecting the data to train our AI model, the next step is to normalize and label the data. The data is currently divided into 10 separate files, each named according to the day it was collected and whether it was gathered from the public or private network.

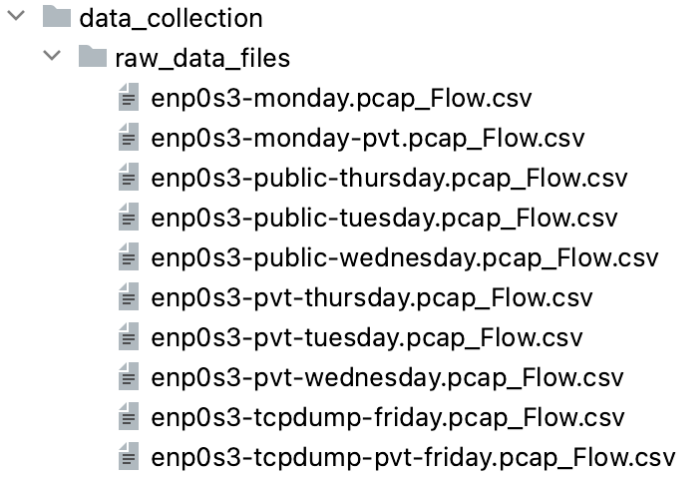

Figure 2.1.1.1.4 (1): the raw data files named according to the day it was collected and whether it was gathered from the public or private (pvt = private) network.

We will now load the data in python and label them accordingly.

```
def merge_data():
    # Load the datasets into pandas dataframes
    df1_public = pd.read_csv("raw_data_files/enp0s3-monday.pcap_Flow.csv")
    df1_private = pd.read_csv("raw_data_files/enp0s3-monday-pvt.pcap_Flow.csv")
    df2_public = pd.read_csv("raw_data_files/enp0s3-public-tuesday.pcap_Flow.csv")
    df2_private = pd.read_csv("raw_data_files/enp0s3-pvt-tuesday.pcap_Flow.csv")
    df3_public = pd.read_csv("raw_data_files/enp0s3-public-wednesday.pcap_Flow.csv")
    df3_private = pd.read_csv("raw_data_files/enp0s3-pvt-wednesday.pcap_Flow.csv")
    df4_public = pd.read_csv("raw_data_files/enp0s3-public-thursday.pcap_Flow.csv")
    df4_private = pd.read_csv("raw_data_files/enp0s3-pvt-thursday.pcap_Flow.csv")
    df5_public = pd.read_csv("raw_data_files/enp0s3-tcpdump-friday.pcap_Flow.csv")
    df5_private = pd.read_csv("raw_data_files/enp0s3-tcpdump-pvt-friday.pcap_Flow.csv")
```

Figure 2.1.1.1.4 (2): function merge_data(), loads the files into pandas dataframes.

```
# Label the datasets as per APT-Detection or Normal traffic (APT-Detection = 1, NORMAL = 0)
df1_public["Label"] = 0
df1_private["Label"] = 0
df2_public["Label"] = 1
df2_private["Label"] = 0
df3_public["Label"] = 1
df3_private["Label"] = 1
df4_public["Label"] = 1
df4_private["Label"] = 1
df5_public["Label"] = 1
df5_private["Label"] = 1
```

Figure 2.1.1.1.4 (3): function merge_data(). Labeling of the data.

In Figure 2.1.1.1.4 (3) we can see that the data is divided into 70% APT traffic and 30% normal traffic. We can now merge the data into one single file.

```
# Merge the datasets into one file
merged_df = pd.concat([df1_public, df1_private, df2_public, df2_private,
                       df3_public, df3_private, df4_public, df4_private,
                       df5_public, df5_private], axis=0)


# Save the merged dataset to a csv file
merged_df.to_csv("trainings_data.csv", index=False)
```

Figure 2.1.1.1.4 (4): function merge_data(). The data is merged into 1 csv file.

Next, we will need to normalize the data by removing unnecessary columns of type "object". We can discard these columns except for the source IP address and destination address, as they contain important information that will be displayed to the user.

```
def normalize_data():
    # Load the data into a pandas dataframe
    df = pd.read_csv("trainings_data.csv")

    # Select columns of interest (src_ip and dst_ip) along with all float columns
    columns_of_interest = ['src_ip', 'dst_ip']
    df_numeric = df.select_dtypes(include='float64')
    df_subset = df[columns_of_interest + list(df_numeric.columns)]

    df_subset.to_csv("trainings_data.csv", index=False)
```

Figure 2.1.1.1.4 (5): function normalize_data().

And that's it, we can now start training our AI model.

2.1.1.1.5 Building the AI model

In this chapter, we will delve into the code for building our AI model and provide a comprehensive explanation of each step. The goal of this code is to build an AI model that can perform classification tasks. To do this, the code uses a combination of unsupervised and supervised learning techniques.

Supervised learning is a type of machine learning where the algorithm learns from labeled data. The labeled data refers to the input data and the corresponding output data. The algorithm learns from this data by finding patterns or relationships between the input and output data, and it uses these patterns to make predictions or classifications on new, unseen data.

In contrast, unsupervised learning is a type of machine learning where the algorithm learns from unlabeled data. The unlabeled data refers to input data that is not accompanied by corresponding output data. The algorithm learns from this data by finding patterns or relationships within the data itself, and it uses these patterns to discover structures or groupings in the data.

Now The first step is to import the necessary libraries for data preprocessing, model creation, and evaluation. These libraries include numpy, pandas, scikit-learn, Keras, and matplotlib.

```
# Import the necessary libraries
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from keras.layers import Input, Dense
from keras.models import Model
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn import metrics
```

Figure 2.1.1.1.5 (1): importing the necessary libraries.

Next, the code loads the training data from a CSV file using pandas' read_csv() function. The data is in a tabular form, where each row represents a sample, and each column represents a feature or label.

```
# Load the data from the CSV file
data = pd.read_csv('../data_collection/trainings_data.csv')
```

Figure 2.1.1.1.5 (2): load the training data in a dataframe.

The feature and label data are separated using pandas' iloc[] method. The feature data is then standardized using scikit-learn's StandardScaler class. Standardization scales the data to have a mean of 0 and a standard deviation of 1. This is an important step in many machine learning algorithms as it helps to mitigate the effect of different scales of the input data on the performance of the model.

```
# Split the data into features and labels
X = data.iloc[:, :-1].values
y = data.iloc[:, -1].values

# Scale the data
sc = StandardScaler()
X = sc.fit_transform(X)
```

Figure 2.1.1.1.5 (3): separate feature and label data. Afterwards, scaling of the data.

The data is then split into training, validation, and testing sets using scikit-learn's train_test_split() function. The training set consists of 80% of the data, while the validation and testing sets each consist of 10% of the data. The training set is used to train the autoencoder model. The validation set is used to tune hyperparameters and prevent overfitting. The testing set is used to evaluate the performance of the trained model on new, unseen data.

```
# Split the data into training, testing and validation sets
X_train, X_val_test, y_train, y_val_test = train_test_split(X, y, test_size=0.2, random_state=0)
X_val, X_test, y_val, y_test = train_test_split(X_val_test, y_val_test, test_size=0.5, random_state=0)
```

Figure 2.1.1.1.5 (4): split the data into training, testing and validation sets.

Next, We want to build a classifier model. There are 4 classifiers: random forest classifier, decision tree, K-Nearest Neighbour and Naive Bayes.

2.1.1.1.5.1 Random forest classifier

A random forest classifier is a type of machine learning algorithm that is used for classification tasks, where the goal is to predict the class label of a new instance based on its features. It is a type of ensemble learning method that combines multiple decision trees to improve the accuracy and generalization of the predictions.

In a random forest classifier, each decision tree is constructed using a random subset of the training data and a random subset of the features. This helps to reduce overfitting and increases the diversity of the trees in the forest. During the training phase, each tree is grown independently by recursively splitting the data based on the selected features and thresholds until the leaves are pure or a stopping criterion is met.

To make a prediction for a new instance, the random forest classifier aggregates the predictions of all the trees in the forest. Each tree votes for a class label based on the majority class of the training instances in its leaf node. The final prediction is then determined by taking the majority vote of all the trees.

The random forest classifier has several advantages over other machine learning algorithms, including:

- It can handle both categorical and continuous data.
- It is robust to outliers and missing values.
- It can capture non-linear relationships between the features and the class label.
- It is less prone to overfitting than a single decision tree.

The random forest classifier has been successfully applied to a wide range of applications, including image recognition, text classification, and bioinformatics. However, it is important to note that the performance of the classifier depends on several factors, such as the number of trees, the number of features, and the quality of the training data.

```
# Building Random Forest model
rf = RandomForestClassifier(random_state=0)
rf.fit(X_train, y_train)
```

Figure 2.1.1.1.5.1: building a random forest model. The parameters are the training dataset from Figure 2.1.1.1.5 (4).

2.1.1.1.5.2 Decision tree

A decision tree is a machine learning algorithm used for classification and regression tasks, where the goal is to predict the value of a target variable based on a set of input features. It is a type of supervised learning method that learns a series of decision rules from the training data to make predictions.

A decision tree consists of a root node, internal nodes, and leaf nodes. The root node represents the entire population, and each internal node represents a feature or attribute of the input data. The leaf nodes represent the final decision or prediction of the algorithm.

During the training phase, the decision tree algorithm recursively splits the training data based on the selected features and thresholds until the leaf nodes are pure or a stopping criterion is met. The splitting process is based on a criterion that measures the homogeneity of the data at each node, such as the Gini index or information gain. The goal is to find the feature and threshold that result in the greatest reduction in impurity or the most information gain.

To make a prediction for a new instance, the decision tree algorithm traverses the tree from the root node to the appropriate leaf node based on the values of the input features. The prediction is then determined by the majority class of the training instances in the leaf node.

The decision tree algorithm has several advantages over other machine learning algorithms, including:

- It is easy to interpret and visualize, making it useful for exploratory data analysis.
- It can handle both categorical and continuous data.
- It can capture non-linear relationships between the features and the target variable.
- It can handle missing values and outliers.

However, the decision tree algorithm has some limitations, such as:

- It can be prone to overfitting, especially when the tree is deep and complex.
- It can be sensitive to small changes in the training data, resulting in different trees and predictions.
- It may not perform well on imbalanced datasets, where one class has significantly fewer examples than the others.

To address these limitations, several variations of the decision tree algorithm have been proposed, such as random forests, gradient boosting, and pruning techniques.

```
# Building Decision Tree model
dtc = tree.DecisionTreeClassifier(random_state=0)
dtc.fit(X_train, y_train)
```

Figure 2.1.1.1.5.2: building a decision tree model. The parameters are the training dataset from Figure 2.1.1.1.5 (4).

2.1.1.1.5.3 K-Nearest Neighbour

K-Nearest Neighbor (KNN) is a supervised machine learning algorithm used for classification and regression tasks. KNN is a type of instance-based or lazy learning algorithm that stores the training instances and classifies new instances based on their similarity to the stored instances.

The KNN algorithm works as follows:

1. The algorithm stores the training instances along with their class labels.
2. When a new instance is presented for classification, the algorithm calculates its distance or similarity to each of the stored instances.
3. The algorithm then selects the K nearest neighbors of the new instance based on their distance or similarity, where K is a predefined parameter.
4. The algorithm assigns the class label of the new instance based on the majority class of the K nearest neighbors.

In the case of regression, the algorithm assigns the output value of the new instance based on the average of the K nearest neighbors. The distance or similarity metric used by the KNN algorithm can vary depending on the type of data and the problem domain. Some common distance metrics include Euclidean distance, Manhattan distance, and cosine similarity.

The KNN algorithm has several advantages, including:

- It is simple and easy to understand.
- It can handle both categorical and continuous data.
- It can handle multi-class classification problems.
- It can be used for both classification and regression tasks.

However, the KNN algorithm also has some limitations, including:

- It can be computationally expensive, especially for large datasets and high-dimensional data.
- It is sensitive to the choice of K and the distance metric.
- It can be affected by the presence of irrelevant or noisy features.

To address these limitations, several variations of the KNN algorithm have been proposed, such as weighted KNN, which assigns more weight to the closer neighbors, and KNN with feature selection or dimensionality reduction techniques to reduce the number of features and improve the performance.

```
# Building KNN model
knn = KNeighborsClassifier()
knn.fit(encoded_train, y_train)
```

Figure 2.1.1.1.5.3: building a K-Nearest Neighbour. The parameters are the training dataset from Figure 2.1.1.1.5 (4).

2.1.1.1.5.4 Naive Bayes

Naive Bayes is a probabilistic machine learning algorithm used for classification tasks. It is based on Bayes' theorem, which describes the probability of a hypothesis given some evidence. In the context of classification, the hypothesis corresponds to a class label, and the evidence corresponds to the features or attributes of an instance.

The Naive Bayes algorithm assumes that the features are conditionally independent given the class label, meaning that the presence or absence of one feature does not affect the probability of the other features. This is a simplifying assumption that may not hold true in all cases, but it allows for efficient computation and can still lead to accurate predictions in many practical applications.

The Naive Bayes algorithm works as follows:

1. The algorithm learns the prior probability of each class label from the training data.
2. The algorithm calculates the likelihood of each feature given the class label, which represents the probability of observing a specific feature value given that the instance belongs to a particular class.
3. The algorithm calculates the posterior probability of each class label given the observed features using Bayes' theorem, which combines the prior probability and the likelihood of the features.
4. The algorithm assigns the class label with the highest posterior probability to the instance.

There are three common types of Naive Bayes algorithms:

1. **Gaussian Naive Bayes**
   This assumes that the features follow a normal distribution, and calculates the likelihood using the mean and variance of the feature values.
2. **Multinomial Naive Bayes**
   This is used for discrete or count data, such as word frequencies in text classification, and calculates the likelihood using the frequency of each feature value.
3. **Bernoulli Naive Bayes**
   This is similar to Multinomial Naive Bayes, but assumes that the features are binary, and calculates the likelihood using the frequency of the presence or absence of each feature.

The Naive Bayes algorithm has several advantages, including:

- It is simple and computationally efficient, making it suitable for large datasets.
- It can handle both categorical and continuous data.
- It can handle multi-class classification problems.
- It can work well even with a small number of training instances.

However, the Naive Bayes algorithm also has some limitations, including:

- The assumption of independence may not hold true in some cases, leading to suboptimal performance.
- It may be sensitive to the presence of irrelevant or correlated features.
- It may not perform well when the classes are imbalanced or when the prior probabilities are not representative of the actual distribution.

To address these limitations, several variations of the Naive Bayes algorithm have been proposed, such as the semi-supervised and hierarchical Bayesian methods.

```
# Building Naive Bayes model
nb = GaussianNB()
nb.fit(X_train, y_train)
```

Figure 2.1.1.1.5.4: building a decision tree model. The parameters are the training dataset from Figure 2.1.1.1.5 (4).

2.1.1.1.6 Model comparison

Now we will build and evaluate these machine learning models for our classification task. We first built four different models: Decision Tree, Random Forest, Naive Bayes, and K-Nearest Neighbors. Then, we defined a function called “evaluate_model” that takes the model, test data, and true labels as input and calculates several evaluation metrics such as accuracy, precision, recall, f1-score, and kappa score. It also calculates the area under the ROC curve and the confusion matrix. We then evaluated each model using the “evaluate_model” function. We visualized the comparison to have a good overview of every classification model and how well each model performed.

```
# Building Decision Tree model
dtc = tree.DecisionTreeClassifier(random_state=0)
dtc.fit(encoded_train, y_train)

# Building Random Forest model
rf = RandomForestClassifier(random_state=0)
rf.fit(encoded_train, y_train)

# Building Naive Bayes model
nb = GaussianNB()
nb.fit(encoded_train, y_train)

# Building KNN model
knn = KNeighborsClassifier()
knn.fit(encoded_train, y_train)
```

Figure 2.1.1.1.6 (1): building the models.

```python
def evaluate_model(model, x_test, y_test):
    from sklearn import metrics
    # Predict Test Data
    y_pred = model.predict(x_test)
    # Calculate accuracy, precision, recall, f1-score, and kappa score
    acc = metrics.accuracy_score(y_test, y_pred)
    prec = metrics.precision_score(y_test, y_pred)
    rec = metrics.recall_score(y_test, y_pred)
    f1 = metrics.f1_score(y_test, y_pred)
    kappa = metrics.cohen_kappa_score(y_test, y_pred)
    # Calculate area under curve (AUC)
    y_pred_proba = model.predict_proba(x_test)[::, 1]
    fpr, tpr, _ = metrics.roc_curve(y_test, y_pred_proba)
    auc = metrics.roc_auc_score(y_test, y_pred_proba)
    # Display confussion matrix
    cm = metrics.confusion_matrix(y_test, y_pred)
    return {'acc': acc, 'prec': prec, 'rec': rec, 'f1': f1, 'kappa': kappa,
            'fpr': fpr, 'tpr': tpr, 'auc': auc, 'cm': cm}


# Evaluate Model
dtc_eval = evaluate_model(dtc, encoded_test, y_test)
# Evaluate Model
rf_eval = evaluate_model(rf, encoded_test, y_test)
# Evaluate Model
nb_eval = evaluate_model(nb, encoded_test, y_test)
# Evaluate Model
knn_eval = evaluate_model(knn, encoded_test, y_test)
```

Figure 2.1.1.1.6 (2): evaluation of the model.

Let's now have a look at the performance of the models.

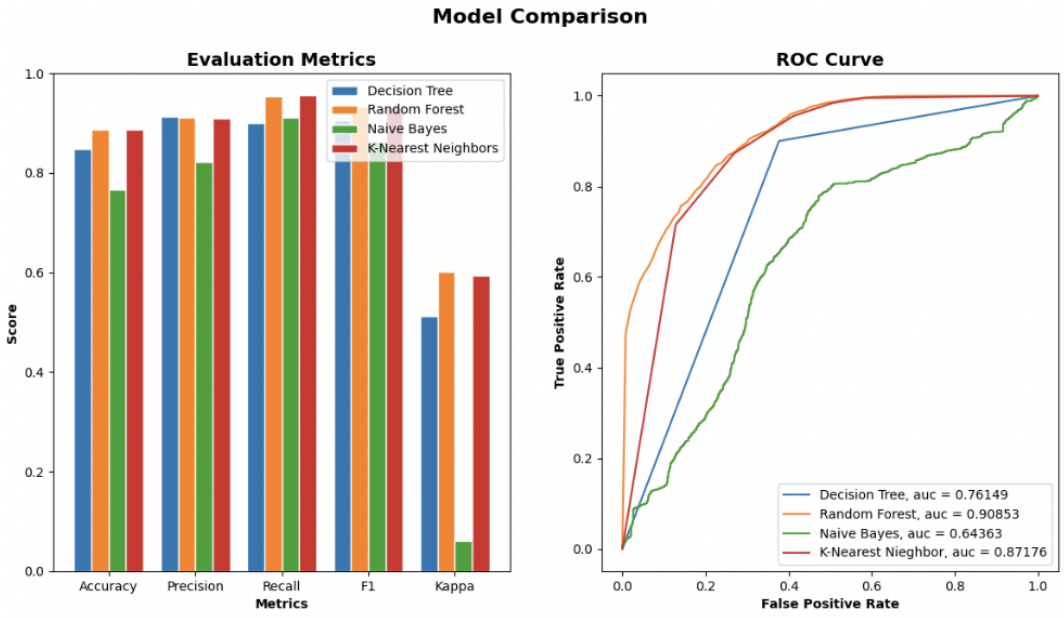

Figure 2.1.1.1.6 (3): model comparison.

Figure 2.1.1.1.6 (3) displays the evaluation metrics for four different classifiers: Random Forest, Decision Tree, K-Nearest Neighbours, and Naive Bayes. The evaluation metrics are Accuracy, Precision, Recall, F1 Score, Cohens Kappa Score, and Area Under Curve.
The Random Forest classifier has the highest values for most of the evaluation metrics, including Accuracy, Precision, Recall, F1 Score, Cohens Kappa Score, and Area Under Curve. This suggests that the Random Forest classifier is the best performing model among the four classifiers.

The Decision Tree classifier has lower values for most of the evaluation metrics compared to the Random Forest classifier, but it still performs relatively well.
The K-Nearest Neighbours classifier has values for most of the evaluation metrics that are close to those of the Random Forest classifier, suggesting that it is also a good performing model.

The Naive Bayes classifier has the lowest values for most of the evaluation metrics, including Accuracy, Precision, Recall, F1 Score, and Cohens Kappa Score. This suggests that the Naive Bayes classifier is the least performing model among the four classifiers. Therefore, it can be concluded that the Random Forest Classifier is the best choice among the given classifiers.

2.1.1.1.7 Evaluation of the model

Now that we have chosen the best classifier, we will have a deeper look into the performance of the model. The performance of the trained model is evaluated using scikit-learn's metrics module, which provides a range of metrics for classification problems such as accuracy, precision, recall, F1 score, confusion matrix and ROC AUC.

First, we are going to use our model to make predictions on the test data by using the "model.predict" function. This function takes in the input data "x_test" as an argument and returns the predicted output "y_pred".

```
# Predict Test Data
y_pred = model.predict(x_test)
```

Figure 2.1.1.1.7: code snippet that makes predictions on the test data.

2.1.1.1.7.1 Precision

Precision is a measure of how accurate a model's positive predictions are, or in other words, it measures the percentage of correct positive predictions out of all positive predictions made by the model.

Precision is calculated using the following formula:
Precision = True Positives / (True Positives + False Positives)

```
prec = metrics.precision_score(y_test, y_pred)
```

Figure 2.1.1.1.7.1: calculating the precision using the metrics library from scikit-learn.

The precision of our model is ***0.9122726678006019***. A precision result of *0.9122726678006019* means that the model's positive predictions are 91.227% accurate. In other words, for every 100 positive predictions made by the model, around 91 of them are correct and 9 of them are incorrect. This precision score is quite high and indicates that the model is accurate in predicting positive cases.

2.1.1.1.7.2 Recall

Recall, also known as sensitivity or true positive rate, measures the ability of a model to identify all the positive cases in a dataset. It measures the percentage of actual positive cases that the model correctly identified as positive.

Recall is calculated using the following formula:
Recall = True Positives / (True Positives + False Negatives)

```
rec = metrics.recall_score(y_test, y_pred)
```

Figure 2.1.1.1.7.2: calculating the recall using the metrics library from scikit-learn.

The recall score of our model is ***0.9667914586799798***. A recall of *0.9667914586799798* means that the model can correctly identify 96.67% of all actual positive cases. In other words, out of all the cases that are actually positive, the model correctly identified 96.67% of them, while 3.33% of actual positive cases were incorrectly predicted as negative by the model.

2.1.1.1.7.3 F1-score

The F1-score is a measure of a machine learning model's overall accuracy, which takes into account both precision and recall. It is the harmonic mean of precision and recall and ranges from 0 to 1, where a higher score indicates better model performance.

The F1-score is calculated using the following formula:
F1-score = 2 * (Precision * Recall) / (Precision + Recall)

```
f1 = metrics.f1_score(y_test, y_pred)
```

Figure 2.1.1.1.7.3: calculating the f1-score using the metrics library from scikit-learn.

The F1-score of our model is ***0.9387411645910467***. An F1-score of *0.9387411645910467* means that the model has good overall accuracy and is able to balance both precision and recall well. It indicates that the model is able to correctly identify a high proportion of actual positive cases while keeping the false positives to a minimum.

2.1.1.1.7.4 Confusion matrix

A confusion matrix is a table that is used to evaluate the performance of a machine learning model by comparing the actual and predicted values of the target variable. It is a way to visualize the performance of a classification model and shows the true positive, false positive, true negative, and false negative values.

The confusion matrix is calculated by comparing the actual values of the target variable with the predicted values of the model. It is a square matrix with rows and columns representing the actual and predicted values, respectively. The diagonal elements of the matrix represent the correctly classified instances (true positives and true negatives), while the off-diagonal elements represent the misclassified instances (false positives and false negatives).

```
cm = metrics.confusion_matrix(y_test, y_pred)
```

Figure 2.1.1.1.7.4 (1): calculation of the confusion matrix using the metrics library from scikit-learn.

```
def plot_confusion_matrix(cm):
    # Plot the confusion matrix as a bar graph
    fig, ax = plt.subplots()
    im = ax.imshow(cm, cmap='Blues')

    # Add labels to the plot
    ax.set_xticks(np.arange(2))
    ax.set_yticks(np.arange(2))
    ax.set_xticklabels(['Predicted Normal', 'Predicted APT-Detection'])
    ax.set_yticklabels(['Actual Normal', 'Actual APT-Detection'])

    # Add annotations to the cells
    for i in range(2):
        for j in range(2):
            ax.text(j, i, cm[i, j], ha="center", va="center", color="black")

    # Display the plot
    plt.title('Confusion Matrix')
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.tight_layout()
    plt.savefig("confusion_matrix.png")
    plt.show()
```

Figure 2.1.1.1.7.4 (2): function plot_confusion_matrix(). This function visualizes the confusion matrix in an easy to understand graph.

The confusion matrix of our mode looks as follows:

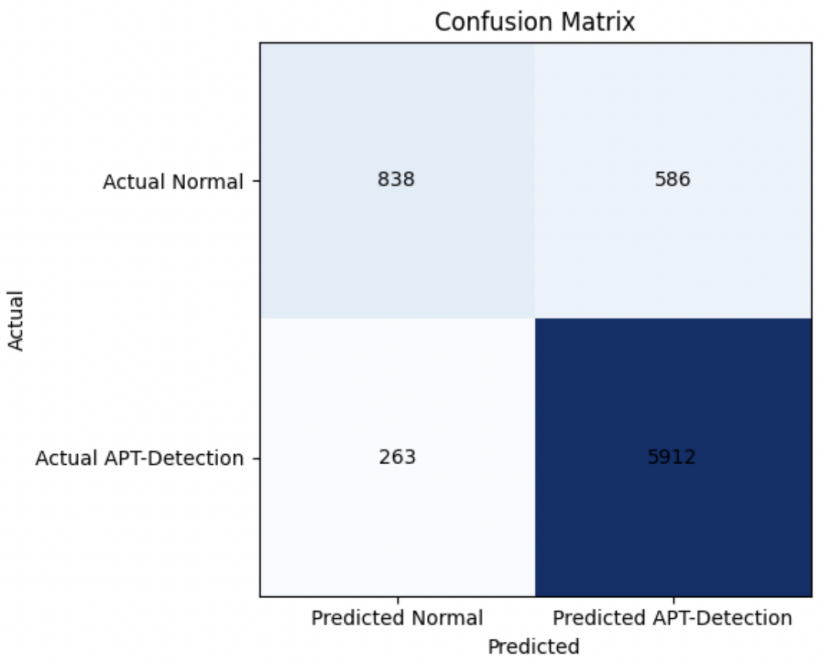

Figure 2.1.1.1.7.4 (3): confusion matrix.

The confusion matrix shows the performance of our binary classification model that predicted two classes (positive and negative). The matrix is 2x2, where the rows represent the actual values, and the columns represent the predicted values.

The matrix shows the following values:
True Positive (TP): 838
False Positive (FP): 586
False Negative (FN): 263
True Negative (TN): 5912

2.1.1.1.7.5 Accuracy

Accuracy is a common evaluation metric used in machine learning to measure the proportion of correctly classified instances out of all instances. It is calculated by dividing the total number of correctly classified instances (true positives and true negatives) by the total number of instances in the dataset.

The formula for accuracy is:
Accuracy = (True Positives + True Negatives) / Total Instances

```
acc = metrics.accuracy_score(y_test, y_pred)
```

Figure 2.1.1.1.7.5: calculation of the accuracy using the metrics library from scikit-learn.

The accuracy of our model is ***0.8950343157044812***. An accuracy of 0.8950343157044812 means that the model correctly classified 89.5% of the instances in the dataset. This suggests that the model has a high overall performance in correctly identifying the classes.

2.1.1.1.7.6 ROC AUC

ROC AUC (Receiver Operating Characteristic - Area Under Curve) is a popular evaluation metric used to assess the performance of a binary classification model. It provides a measure of the model's ability to distinguish between the positive and negative classes by plotting the true positive rate (TPR) against the false positive rate (FPR) for different threshold values.

The ROC AUC value ranges between 0 and 1, with higher values indicating better performance. The ROC AUC of our model is ***0.7533785766813268***. An ROC AUC of 0.7533785766813268 indicates that the model has moderate performance in correctly distinguishing between the positive and negative classes, and there is potential for further improvement.

2.1.1.2 Phishing URL analysis from phishing mails

2.1.1.2.1 What are phishing attacks?

Phishing attacks are a type of cyber-attack that relies on disguising an email address, phone number, sender name or website URL to try to make the recipient believe they are interacting with a trusted source. Criminals then try to get you to give up sensitive information, send money or download malicious software.

In a phishing scam, you might receive an email that appears to be from a legitimate business and is asking you to update or verify your personal information by replying to the email or visiting a website. The web address might look similar to one you've used before. The email may be convincing enough to get you to take the action requested.

But once you click on that link, you're sent to a spoofed website that might look nearly identical to the real thing - like your bank or credit card site - and asked to enter sensitive information like passwords, credit card numbers, banking PINs, etc. These fake websites are used solely to steal your information.

2.1.1.2.2 How can machine learning be used to prevent phishing attacks?

Many approaches have been used to filter out phishing websites. Each of these methods is applicable on different stages of attack flow, for example, network-level protection, authentication, client-side tool, user education, server-side filters, and classifiers. Although there are some unique features in every type of phishing attack, most of these attacks depict some similarities and patterns. Since machine learning methods proved to be a powerful tool for detecting patterns in data, these methods have made it possible to detect some of the common phishing traits, therefore, recognizing phishing websites.

There are a couple of ways machine learning can be used to prevent phishing attacks, one of those is to actually look at the content of the email and look at the language being used using Natural Language Processing (NLP). With a model like BERT we can determine if the text in the email resembles a phishing attack/social engineering. Another way to implement machine learning is using features either extracted from the email (to detect spoofing of email addresses etc.) or from the URL inside the email. In our tool we decided on using the latter method, mostly because the datasets for phishing urls are much more readily available than that for actual phishing emails.

2.1.1.2.3 Features of the dataset

For the machine learning model to predict if a url is benign or phishing it needs certain data. This data or features are things such as having an IP address in the URL or if there is double slash redirection etc. The dataset we used consists of 30 of such features extracted from roughly 11 thousand urls with a ratio of around 40/60 benign to phishing urls. To increase the accuracy of our model we used the same method to extract features (from the original dataset) and the Phishtank dataset, containing more than 30 thousand valid phishing urls, to Increase the amount of data to train our model with. We also used another dataset from kaggle to add more benign urls. In total we had around 17 thousand rows of data that we could use.

2.1.1.2.4 Which models are we going to look at?

According to the paper where we got our dataset from, they concluded that the best models to use for this particular dataset would be XGBoost classifier or MLP classifier.

XGBoost is a refined and customized version of Gradient Boosting to provide better performance and speed. “The most important factor behind the success of XGBoost is its scalability in all scenarios. The XGBoost runs more than ten times faster than popular solutions on a single machine and scales to billions of examples in distributed or memory-limited settings. The scalability of XGBoost is due to several important algorithmic optimizations. These innovations include a novel tree learning algorithm for handling sparse data; a theoretically justified weighted quantile sketch procedure enables handling instance weights in approximate tree learning. Parallel and distributed computing make learning faster which enables quicker model exploration. More importantly, XGBoost exploits out-of-core computation and enables data scientists to process hundreds of millions of examples on a desktop. Finally, it is even more exciting to combine these techniques to make an end-to-end system that scales to even larger data with the least amount of cluster resources.” [2.1.1.2.4 | 1]

Artificial neural networks (ANNS) are a learning model roughly inspired by biological neural networks. These models are multilayered, each layer containing several processing units called neurons. Each neuron receives its input from its adjacent layers and computes its output with the help of its weight and a non-linear function called the activation function. In feed-forward neural networks, data flows from the first layer to the last layer. Different layers may perform different transformations on their input. The weights of neurons are set randomly at the start of the training and they are gradually adjusted by the help of the gradient descent method to get close to the optimal solution. The power of neural networks is due to the non-linearity of hidden nodes. As a result, introducing non-linearity in the network is very important so that you can learn complex functions. [2.1.1.2.4 | 2]

For our tool we looked at both these models and fine tuned them using hyperopt to evaluate them to use the best model.

2.1.1.2.5 Training the model

2.1.1.1.5.1 Data collection and transformation

As discussed in a previous chapter our dataset contains 30 features. These features are extracted using a python function made up of 30 if else statements where either a -1, 0 or 1 are added to a list.

```
# 11. port
try:
    port = domain.split(":")[1]
    if port:
        data_set.append(-1)
    else:
        data_set.append(1)
except:
    data_set.append(1)

# 12. HTTPS_token
if re.findall(r"^https://", url):
    data_set.append(1)
else:
    data_set.append(-1)
```

Figure 2.1.1.1.5.1 (1): example of feature extraction.

All these if-else statements are collected in one function that returns a list of 0's and 1's, because the feature extraction relies on html requests to get information it takes a long time to extract features for a large amount of urls. To mitigate this problem we decided to use multithreading using the concurrent futures library shown in the next image.

```
def extract_features(csv_file="Phishing_detection/assessment_engine/scrape_urls.csv"):
    # Extract features out of these urls
    global dataset
    global to_predict
    global unable_to_check
    to_predict = pd.read_csv(csv_file, header=0)
    with concurrent.futures.ThreadPoolExecutor() as executor:
        dataset = []
        unable_to_check = []
        for i in to_predict.index:
            executor.submit(threaded_prediction_dataset_generator, i)
```

Figure 2.1.1.1.5.1 (2): multithreading feature extraction.

Here we loop over the lines in the csv file and start up a thread running the threaded_prediction_dataset_generator function with the index of the csv as argument. The following image shows the function which collects the extracted features and saves them into a csv file to be read when making a prediction.

```
def threaded_prediction_dataset_generator(i):
    try:
        #get array of features
        extracted_features = generate_data_set(to_predict["url"][i])

        if len(extracted_features) != 30:
            unable_to_check.append(to_predict[i].append("Unable to check"))
            with open("unable_to_check.csv", "w", newline='') as unable_csv:
                csvwriter = csv.writer(unable_csv, delimiter=',')
                header = ['email-id', 'sender', 'subject', 'url', "result"]
                csvwriter.writerow(header)
                csvwriter.writerows(unable_to_check)
        else:
            extracted_features.append(to_predict["url"][i])
            dataset.append(extracted_features)
            with open("Phishing_detection/assessment_engine/to_predict.csv", "w", newline='') as my_csv:
                csvwriter = csv.writer(my_csv, delimiter=',')
                header = ["having_IPhaving_IP_Address", "URLURL_Length", "Shortining_Service",
                          "having_At_Symbol", "double_slash_redirecting", "Prefix_Suffix", "having_Sub_Domain",
                          "SSLfinal_State", "Domain_registeration_length", "Favicon", "port", "HTTPS_token",
                          "Request_URL", "URL_of_Anchor", "Links_in_tags", "SFH", "Submitting_to_email", "Abnormal_URL",
                          "Redirect", "on_mouseover", "RightClick", "popUpWidnow", "Iframe", "age_of_domain",
                          "DNSRecord", "web_traffic", "Page_Rank", "Google_Index", "Links_pointing_to_page",
                          "Statistical_report", 'url']
                csvwriter.writerow(header)
                csvwriter.writerows(dataset)
        print("All features extracted successfully for " + to_predict["url"][i][:20] + "...")
    except:
        pass
```

Figure 2.1.1.1.5.1 (3): feature extraction and saving into csv file.

2.1.1.1.5.2 Building the model and fine tuning

To start with building and fine tuning the models we need to import the necessary libraries shown as followed:

```
import pandas as pd
import numpy
import sklearn
from sklearn.model_selection import train_test_split
from sklearn import preprocessing
from sklearn.model_selection import cross_validate
from sklearn.neural_network import MLPClassifier
import matplotlib.pyplot as plt
from joblib import dump
import xgboost as xgb
from sklearn.metrics import accuracy_score
# import packages for hyperparameters tuning
from hyperopt import STATUS_OK, Trials, fmin, hp, tpe
import sklearn.metrics as metrics
import numpy as np
```

Figure 2.1.1.1.5.2 (1): import of necessary libraries.

Next we can load in our dataset as a pandas dataframe and separate the x and y values and split our dataset into 80% validation and 20% testing.

```
df = pd.read_csv("../data_collection/training_dataset.csv", )

df = sklearn.utils.shuffle(df)
X = df.drop("Result", axis=1).values
X = preprocessing.scale(X)
mapping = {-1: 1, 1: 0}
df = df.replace({"Result": mapping})
y = df['Result'].values

# split in 80% train and 20% final testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=42, shuffle=True)
```

Figure 2.1.1.1.5.2 (2): split dataset into validation and testing.

We also check that the ratio of phishing and benign urls is around the same in the test and training dataset so that we can get accurate results when evaluating our model.

```
# Check if ratio of -1 and 1 is equal between train and test set
unique, counts = numpy.unique(y_test, return_counts=True)
percentages = dict(zip(unique, counts * 100 / len(y_test)))
print(percentages)
unique, counts = numpy.unique(y_train, return_counts=True)
percentages = dict(zip(unique, counts * 100 / len(y_train)))
print(percentages)
```

Figure 2.1.1.1.5.2 (3): ratio check.

Now that our training and testing dataset is set up correctly we use hyperopt to fine tune our models for comparison. Hyperopt is a library that is based on Bayesian optimization and requires 4 essential components for the optimization of hyperparameters: the *search space*, the *loss function*, the *optimization algorithm* and *a database* for storing the history.

The search space is a collection of hyperparameters with which hyperopt needs to find the best combination in code this looks like this:

```
solvers = ['lbfgs', 'adam', 'sgd']
learning_rates = ['constant', 'invscaling', 'adaptive']
activations = ['identity', 'logistic', 'tanh', 'relu']
space = {'solver': hp.choice('solver', solvers),
         'alpha': hp.uniform('alpha', 0, 1),
         'learning_rate': hp.choice('learning_rate', learning_rates),
         'activation': hp.choice('activation', activations)
         }
```

Figure 2.1.1.1.5.2 (4): hyperparameters.

Now that you have the search space we need a loss function which will evaluate the given hyperparameters and return a value to use as a loss.

```
def objective(space):
    clf = MLPClassifier(hidden_layer_sizes=(33,), early_stopping=True, max_iter=500, solver=space["solver"],
                        alpha=space["alpha"], learning_rate=space["learning_rate"])

    clf.fit(X_train, y_train)
    pred = clf.predict(X_test)
    accuracy = accuracy_score(y_test, pred > 0.5)
    print("SCORE:", accuracy)
    return {'loss': -accuracy, 'status': STATUS_OK}
```

Figure 2.1.1.1.5.2 (5): evaluate hyperparameters.

Next we define a trials object which will be used as a database for our history:

```
trials = Trials()
```

Figure 2.1.1.1.5.2 (6): trials object.

And finally we will use the hyperopt fmin function and give our search space, loss function and trials this will eventually return the best parameters found.

```
best_hyperparams = fmin(fn=objective,
                        space=space,
                        algo=tpe.suggest,
                        max_evals=200,
                        trials=trials)
print(best_hyperparams)
```

Figure 2.1.1.1.5.2 (7): return best hyperparameters.

2.1.1.1.5.3 Evaluating the model

Now that we have fine tuned our models we can evaluate them using our test split of the dataset which our model hasn't seen yet. We create an instance of both of our models using the previously found hyperparameters and do a prediction using our test data. We use the same metrics discussed in the previous chapter to evaluate our model and we get these results.

```
####### MLP Classifier #######
Accuracy: 0.9725065808715999
Precision: 0.9849806201550387
Recall: 0.9699427480916031
F1 Score: 0.9774038461538462
Cohens Kappa Score: 0.942308327493308
Area Under Curve: 0.9964868474955716
```

```
####### XGBoost Classifier #######
Accuracy: 0.9368236326411231
Precision: 0.9467680608365019
Recall: 0.950381679389313
F1 Score: 0.9485714285714286
Cohens Kappa Score: 0.8666919863273832
Area Under Curve: 0.9848406495762003
```

Figure 2.1.1.1.5.3 (1) & (2): evaluate metrics of models.

And these are the confusion matrices.

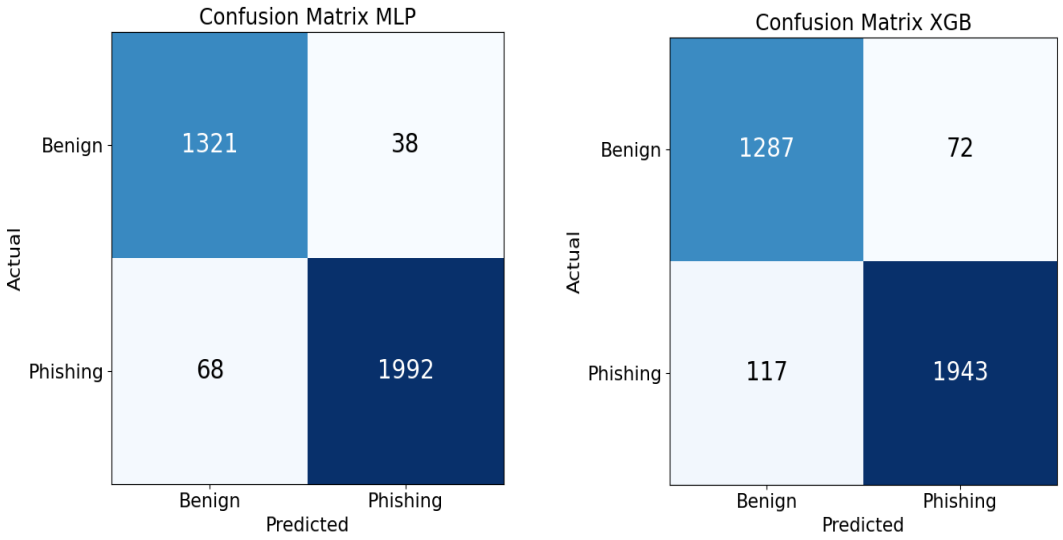

Figure 2.1.1.1.5.3 (3) & (4): confusion matrices.

Looking at these results we can see that our MLP classifier is a little better than XGboost but they are both decent models for this dataset. The strength of Xgboost comes from the speed at which it can train data, it takes 100x less time to train but prediction time is the same as MLP.

2.1.2 Deep learning

Deep learning is a subset of machine learning that involves the use of artificial neural networks to model and solve complex problems. It is a type of machine learning that allows computers to learn from experience and make decisions based on that experience. Deep learning models are inspired by the structure and function of the human brain, where multiple layers of neurons process and transmit information to produce complex behaviors.

Deep learning models are trained on large datasets and can automatically learn patterns and relationships between inputs and outputs. They are capable of processing vast amounts of data and identifying complex structures and features that might be difficult for humans to recognize. These models can perform tasks such as image and speech recognition, natural language processing, autonomous driving, and many more.

The neural network in a deep learning model consists of multiple layers of interconnected nodes that process and transmit information. The input layer receives data from the environment, such as an image or text. The hidden layers process the data and extract features that are relevant to the task at hand. The output layer produces the final output, such as a classification or prediction.

Deep learning models are typically trained using a process called backpropagation, where the error between the predicted output and the actual output is used to adjust the weights and biases in the network. This process is repeated many times until the model reaches an acceptable level of accuracy.

The development of deep learning has led to significant advancements in fields such as computer vision, natural language processing, robotics, and many others. It has enabled machines to perform tasks that were previously thought to be the exclusive domain of humans. Deep learning has also led to the development of new technologies such as self-driving cars, voice assistants, and intelligent medical devices.

2.1.2.1 Malware analysis and detection

Malware or malicious software is a serious threat to computer systems and can cause significant damage. It is a generic term that describes any malicious code or program designed to be harmful to systems and can take many different forms. It includes viruses, Trojans, spyware, keyloggers, worms, ransomware, adware and other unwanted software that gets secretly installed onto your device.

- **Virus**
  Is self-replicating by attaching itself to other programs or files and inserting its code into them. It activates by opening them, this way it can be spread very easily from one computer to another.
- **Trojan (horse)**
  Disguises itself as a legitimate program to trick users into downloading and installing it.

- **Spyware**
  Gather information about a user's computer or online activity without their knowledge or consent.
- **Keylogger**
  Type of spyware that steals sensitive data such as credit card numbers, usernames and passwords by logging what the user types.
- **Worm**
  Is self-replicating, but unlike a virus, a worm spreads across networks without needing to attach itself to other programs or files. It can spread on its own without needing user interaction or activation.
- **Ransomware**
  Demands users to pay a ransom to regain access to systems or files.
- **Adware**
  Pushes unwanted advertisements to users.

Detecting malware is crucial in ensuring the security of computer systems, because new threats and countermeasures are constantly emerging due to the ongoing conflict between security experts and cybercriminals. Although there are many tools and strategies to prevent malware attacks, cybercriminals stay persistent.

Besides there are several types of malware detection techniques that can be used by those tools to identify and prevent malware from infecting a computer or network. The most common ones are:

- **Signature-based detection**
  Relies on a database of known malware signatures which are unique digital fingerprints that are associated with specific malware variants. The database has to be regularly updated with new signatures as new malware variants are discovered. When a file or code is scanned, it needs to be compared to the known signatures and if a match is found, it flags the file as malware.
- **Heuristic detection**
  Analyzes files or code to detect potentially malicious behavior, such as self-replication, overwriting files, modifying system files or registry keys and other behavior that is common among malware, even if no known malware signatures are present.
- **Sandbox analysis**
  Runs suspicious files or code in a controlled, isolated environment called a sandbox where its behavior is being monitored. When there are signs of malicious behavior, it flags the file as malware.

Although signature-based detection, heuristic detection and sandbox analysis are all common methods used for detecting malware, they each have their limitations.

Signature-based detection cannot detect unknown variants that do not match any signature in the database. Heuristic detection can be effective for identifying unknown malware, but it can also generate false positives and negatives, leading to the misclassification of legitimate software as malicious or the failure to detect some types of malware. And sandbox analysis can also be effective in detecting unknown malware, but it is time consuming and some malware can detect when it's running in a sandbox environment and behave benignly. Thus whilst having several types of malware detection techniques, detecting malware can still be challenging.

In addition there is another technique called deep learning based detection. Deep learning is a subset of machine learning that uses artificial neural networks to learn and make predictions from data. One popular approach for malware detection using deep learning is to use Convolutional Neural Networks (CNNs) which are very successful in image classification tasks that can be used for malware classification. CNNs can learn to recognize features in data automatically which is useful in malware detection where there are many different types of malware with different features. Important in a CNN is to keep training it regularly on new datasets and to keep monitoring its performance.

2.1.2.1.1 Image classification

To detect malware using image classification the binary code of a malware sample is converted into a grayscale image.

Binary files are files that contain data in a binary format, which means that the data is stored in a sequence of binary numbers. A binary number has only two possible values: 0 or 1. This means that a malware binary exists of multiple binary numbers and will look like 01110011010110010101011010101000001 and so on. Then the malware binary is read as a vector of 8-bit unsigned integers (1D array of bytes) and organized into a 2D array, where each byte (group of 8 bits) in the malware binary is treated as an individual element of the array. And each byte represents numbers between 0 (black) and 255 (white) which can be visualized as a grayscale image. The numbers between 0 and 255 represent varying shades of gray. For example, a value of 64 would represent a darker shade of gray then a value of 128.

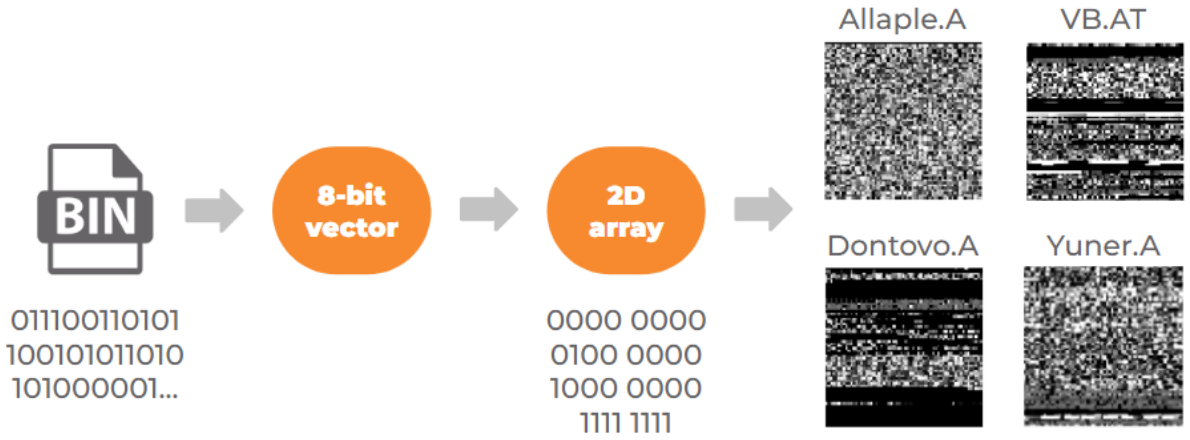

Figure 2.1.1.3.1: visualization of malware as an image.

The grayscale images are in image classification represented as a matrix of pixel values and each pixel in the grayscale images represents a byte in the code. Then features that are indicative of malware can be extracted from the images and used to train machine learning models by using this pixel data, such as convolutional neural networks (CNNs) which is a common technique used for image classification.

2.1.2.1.2 Convolutional Neural Network (CNN)

A Convolutional Neural Network or abbreviated CNN is a machine learning model which is trained on labeled data to identify different classes, in this case of malware, within images. To identify those different classes it learns to identify patterns and features within the images that are unique to each class.

2.1.2.1.3 2D convolution layer

A CNN consists of multiple layers where the convolution layer is the key component. The first layer is always a convolutional layer which is used to extract patterns or features such as edges, shapes, textures etc. from images.

We use a 2D convolution layer. It uses a filter or kernel which is a matrix of random weights and which is able to detect the patterns. In the example below we have a kernel size of 3x3 which moves across the input, a malware image, one step at a time by sliding or convolving over each block of 3x3 pixels until it slid over the entire image. When a filter convolved a given input, it gives us a resulting output/convolute/feature map.

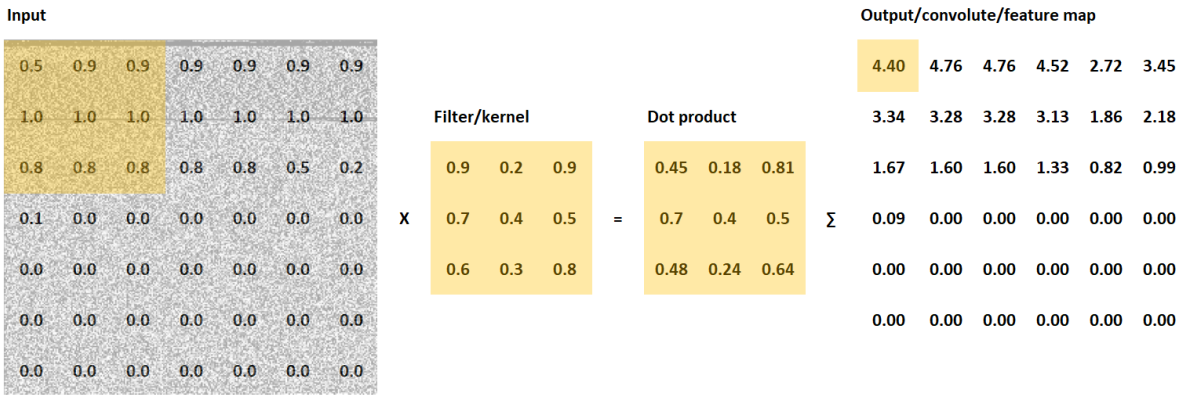

Figure 2.1.1.3.3: working of a 2D convolution layer.

You may notice that when for example a 64x64 image is passed through a convolution layer in a CNN, it can result in the output image being smaller in size. In this case the resulting output image would be 62x62 pixels. This is because at the edges of the image, the filter only has enough pixels to slide over two rows or two columns at a time, instead of the full three. As a result, the output image shrinks two pixels in size, because it will be missing one row and one column of pixels on each edge.

2.1.2.1.4 Max pooling

Max pooling reduces the dimensionality of images by reducing the number of pixels in the output from the previous convolution layer. It is typically applied after each convolution layer in a CNN, and is followed by another convolution layer.

If you use a pool filter with a pool size of 2x2, it calculates the max value of all the 2x2 blocks in the convolved output. By calculating the max values it picks out the higher valued pixels with the most important features while discarding the lower valued pixels with the less important features. This ensures that the network can still recognize the same pattern or feature regardless of where it appears in the image.

By default it uses a stride, meaning how many pixels you want the pool filter to move as it slides, equal to the pool size. A pool size of 2x2 reduces the height and width of the input by a factor of 2 which means that each 2x2 block of pixels in the input is replaced by a single pixel in the output. In other words if the input feature map for max pooling is 62x62, then the resulting feature map will be in this case half the size being 31x31.

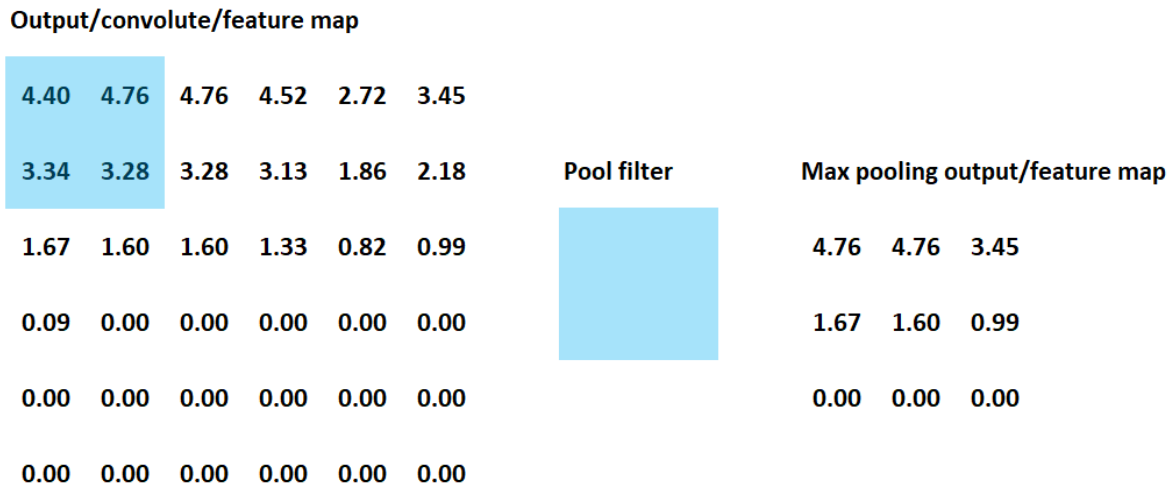

Figure 2.1.1.3.4: working of a max pooling layer.

Additionally max pooling may help to reduce overfitting, because the network is forced to focus on the most remarkable features and discard the less important ones.

2.1.2.1.5 Dropout

A dropout layer in a CNN is also used to reduce overfitting by randomly ignoring or dropping out a certain percentage of nodes or neurons during training. Overfitting is when the model gets too good at predicting the data it has seen before, but doesn't do as well on new data.

2.1.2.1.6 Building the AI model

Now you know what a CNN is and how it works, we can start with explaining the Python code to build and train our AI model to classify malware images.

We used the Malimg dataset which already contains 9,332 grayscale images of malware samples belonging to 25 different classes:

Class number	Class name	Malware kind	Samples
1	Adialer.C	Dialer	115
2	Agent.FYI	Backdoor	118
3	Allaple.A	Worm	2,949
4	Allaple.L	Worm	1,591
5	Alueron.gen!J	Trojan	198
6	Autorun.K	Worm AutoIT	106
7	C2LOP.P	Trojan	200
8	C2LOP.gen!g	Trojan	146
9	Dialplatform.B	Dialer	177
10	Dontovo.A	Trojan Downloader	162
11	Fakerean	Rogue	381
12	Instantaccess	Dialer	431
13	Lolyda.AA1	Password Stealer	213
14	Lolyda.AA2	Password Stealer	184
15	Lolyda.AA3	Password Stealer	123
16	Lolyda.AT	Password Stealer	159
17	Malex.gen!J	Trojan	136
18	Obfuscator.AD	Trojan Downloader	142
19	Rbot!gen	Backdoor	158
20	Skintrim.N	Trojan	80
21	Swizzor.gen!E	Trojan Downloader	128
22	Swizzor.gen!I	Trojan Downloader	132
23	VB.AT	Worm	408
24	Wintrim.BX	Trojan Downloader	97
25	Yuner.A	Worm	800

Figure 2.1.1.3.6 (1): Malimg dataset.

The first part of the script imports the necessary libraries and sets the path to the folder with images (Malimg dataset).

```
from keras.preprocessing.image import ImageDataGenerator
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from sklearn.utils import compute_class_weight
from sklearn import metrics
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

# Folder with images (dataset)
path_root = 'data_collection/malimg_dataset/malimg_dataset_subdir_imgs'
```

Figure 2.1.1.3.6 (2): used libraries and path to Malimg dataset.

Next, we have to use ImageDataGenerator().flow_from_directory() to be able to use our images. Specifically, it will iterate over the images, identify classes automatically from the folder name and divide the dataset into a batch or group of images which are resized to 64x64. It doesn't matter if the batch size is bigger than the actual amount of images in the dataset.

```
# ImageDataGenerator      Iterator
# flow_from_directory     Identifies classes automatically from the folder name
# directory               The path to parent directory containing sub-directories (class, label) with images
# target_size=(64,64)     Resizes images to 64x64
# batch_size              Divides dataset into batches/groups of 10000 images
batches = ImageDataGenerator().flow_from_directory(directory=path_root, target_size=(64, 64), batch_size=10000)
```

Figure 2.1.1.3.6 (3): define batch.

In the following way we can then print our classes to see if they have been well recognized.

```
# batches.class_indices   Maps folder/class names to their corresponding indices (0 to class 1 etc.)
for malware_class, index in batches.class_indices.items():
    print(malware_class, ":", index)
```

Figure 2.1.1.3.6 (4): print malware classes with indices.

Furthermore the script generates a batch of images and labels using next(). The batch size of images will be 9,332 with 64x64x3 as width, length and depth. The 3 stands for RGB (red, green, blue). The batch size of labels will also be 9,332 with 25 as the number of classes.

```
    # Generate a batch of images and labels
    imgs, labels = next(batches)  # next to go through all elements in batches
    print(f"\nShape of images batch: {imgs.shape}")  # batch_size, width, length, depth
    print(f"Shape of labels batch: {labels.shape}")  # batch_size, number of classes
```

Figure 2.1.1.3.6 (5): batches of images and labels.

It also uses a function to analyze the partition of data in % with a bar plot.

```
# Analyze the partition of data in % with a barplot
def barplot_data_partition():
    classes = batches.class_indices.keys()
    perc = (sum(labels) / labels.shape[0]) * 100
    plt.xticks(rotation='vertical')
    plt.bar(classes, perc)
    plt.gcf().subplots_adjust(bottom=0.3)  # Adjust the bottom margin to make room for rotated labels
    plt.savefig('AI_model/barplot_data_partition.png', bbox_inches='tight')
    plt.show()


barplot_data_partition()  # Dataset is quite unbalanced, with many malwares in classes 2 (Allaple.A) and 3 (Allaple.L).
```

Figure 2.1.1.3.6 (6): creating a bar plot to analyze partition of data.

Here you can see the bar plot itself.

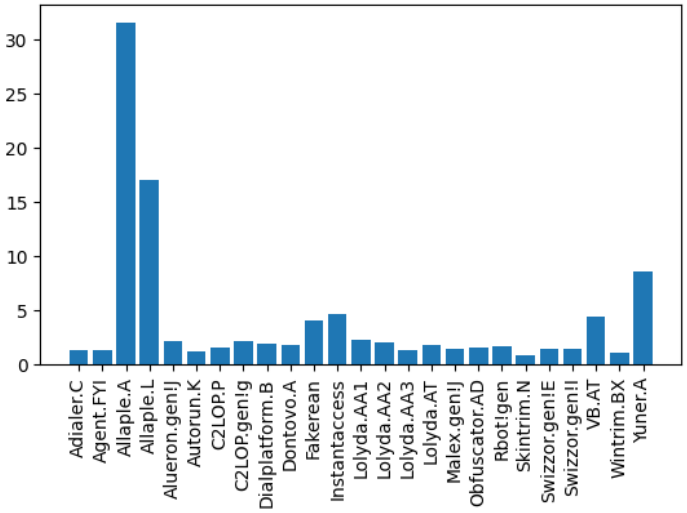

Figure 2.1.1.3.6 (7): bar plot of Malimg dataset.

In the bar plot you can see that the Malimg dataset is quite unbalanced. A lot of malwares belong to Allaple.A (+30%) and Allaple.L (+15%).

For training and testing purposes for our model, we should have our data of images and labels broken down in three different datasets:

1. Training set used to train the model.
2. Validation set used to validate the model during training to ensure that our model is not overfitting to the data in the training set.
3. Test set used to test the model after the model has already been trained.

The data of the training set is separate from both validation and test set!

We decided to put 70% of our images into a training set, 15% in a validation set and another 15% in a test set.

```
# Splits the data into a training set (70%) and a testing set (30%)
X_train, X_test, y_train, y_test = train_test_split(imgs / 255., labels, test_size=0.3)
# Splits the testing set into a validation set (15%) and a new testing set (15%)
X_val, X_test, y_val, y_test = train_test_split(X_test, y_test, test_size=0.5)
print("\nX train shape:", X_train.shape, "\t\tY train shape:", y_train.shape)
print("X validation shape:", X_val.shape, "\tY validation shape:", y_val.shape)
print("X test shape:", X_test.shape, "\t\tY test shape:", y_test.shape, "\n")
```

Figure 2.1.1.3.6 (8): creating training, validation and test set.

Eventually we can start building our CNN model using Keras. The model consists of two convolutional layers followed by max pooling layers, dropout layers to reduce overfitting, a flatten layer to convert the 2D feature maps into a 1D feature vector, and dense layers for classification. The output layer is a dense layer with a softmax activation function, which assigns probabilities to each class.

```python
# Building Convolutional Neural Network (CNN) model
num_classes = 25


def create_model():
    model = Sequential()

    model.add(Conv2D(30, kernel_size=(3, 3), activation='relu', input_shape=(64, 64, 3)))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Conv2D(15, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Dropout(0.25))

    model.add(Flatten())
    model.add(Dense(128, activation='relu'))

    model.add(Dropout(0.5))

    model.add(Dense(50, activation='relu'))
    model.add(Dense(num_classes, activation='softmax'))

    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model


malware_model = create_model()
malware_model.summary()
```

Figure 2.1.1.3.6 (9): building CNN model.

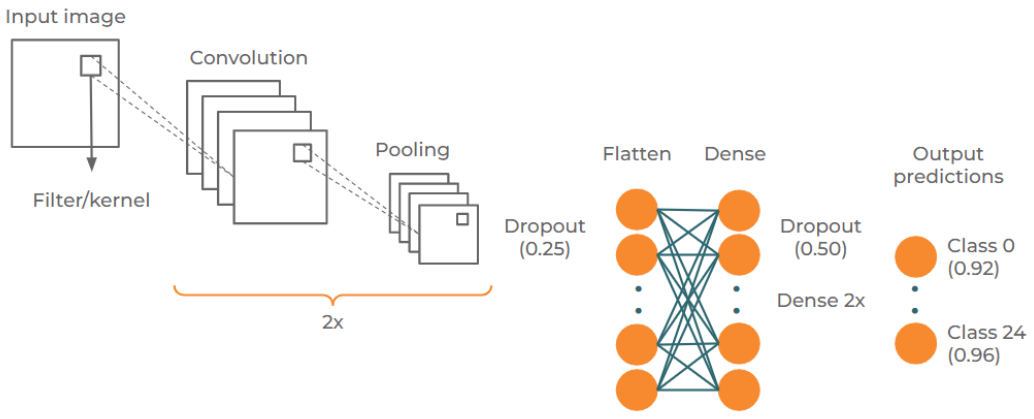

Figure 2.1.1.3.6 (10): architecture of CNN model.

After our model is created, we first start by adding weights to each malware class before we start with training our model. By adding weights we can deal with the unbalance in our data, as we saw before in the bar plot. A class with more samples will have a higher weight than a class with less samples.

```
y_train_new = np.argmax(y_train, axis=1)  # Extract class for each sample in y_train

# Compute the class weights based on the number of samples in each class
class_weights = compute_class_weight(class_weight="balanced", classes=np.unique(y_train_new), y=y_train_new)
class_weights = dict(zip(np.unique(y_train_new), class_weights))
print("\nComputed class weights.")
for class_number, weight in class_weights.items():
    print(f"Class {class_number}: {weight}")
```

Figure 2.1.1.3.6 (11): adding class weights to deal with unbalance in data.

Now we have dealt with the unbalance in our data by adding class weights, we can finally start with training and validating our model using those class weights. We also save the model in a directory to use it later and print the accuracy of our model.

```
# Train and validate model using the computed class weights
print("\nTraining and validating model using the computed class weights.")
malware_model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=10, class_weight=class_weights)

# Save the model in current directory
malware_model.save('AI_model/malware_model_t.h5')  # Renamed to prevent accidentally overwriting

scores = malware_model.evaluate(X_test, y_test)
print('Final CNN accuracy: ', scores[1], "\n")
```

Figure 2.1.1.3.6 (12): training and validating model using computed class weights.

You can see that during each epoch the loss will get lower and our accuracy will get higher, resulting in a final accuracy of 94%.

```
Epoch 1/10
205/205 [==============================] - 10s 46ms/step - loss: 2.1587 - accuracy: 0.2927 - val_loss: 1.2313 - val_accuracy: 0.5721
Epoch 2/10
205/205 [==============================] - 9s 45ms/step - loss: 0.9054 - accuracy: 0.5126 - val_loss: 0.9491 - val_accuracy: 0.4479
Epoch 3/10
205/205 [==============================] - 9s 45ms/step - loss: 0.6669 - accuracy: 0.5667 - val_loss: 0.7583 - val_accuracy: 0.5850
Epoch 4/10
205/205 [==============================] - 9s 42ms/step - loss: 0.5935 - accuracy: 0.5955 - val_loss: 0.6812 - val_accuracy: 0.6771
Epoch 5/10
205/205 [==============================] - 9s 43ms/step - loss: 0.4750 - accuracy: 0.6736 - val_loss: 0.5440 - val_accuracy: 0.7793
Epoch 6/10
205/205 [==============================] - 9s 43ms/step - loss: 0.4330 - accuracy: 0.7142 - val_loss: 0.5633 - val_accuracy: 0.8150
Epoch 7/10
205/205 [==============================] - 9s 43ms/step - loss: 0.3785 - accuracy: 0.7551 - val_loss: 0.4411 - val_accuracy: 0.8021
Epoch 8/10
205/205 [==============================] - 9s 44ms/step - loss: 0.3431 - accuracy: 0.7725 - val_loss: 0.4603 - val_accuracy: 0.7907
Epoch 9/10
205/205 [==============================] - 10s 47ms/step - loss: 0.3337 - accuracy: 0.7825 - val_loss: 0.2727 - val_accuracy: 0.9407
Epoch 10/10
205/205 [==============================] - 10s 47ms/step - loss: 0.2885 - accuracy: 0.8256 - val_loss: 0.2689 - val_accuracy: 0.9271
44/44 [==============================] - 1s 11ms/step - loss: 0.2637 - accuracy: 0.9357
Final CNN accuracy:  0.9357143044471741
```

Figure 2.1.1.3.6 (13): epochs and accuracy of model.

Finally, we can evaluate the performance of our model by creating and displaying a confusion matrix as a heatmap.

```
# Evaluate performance of the model using a confusion matrix as a heatmap
y_pred_prob = malware_model.predict(X_test, verbose=0)  # Predict class probabilities of samples in X_test
y_pred = np.argmax(y_pred_prob, axis=1)  # Selects class with the highest probability as the predicted class
y_test2 = np.argmax(y_test, axis=1)  # Extract the true classes of the samples in X_test
c_matrix = metrics.confusion_matrix(y_test2, y_pred)


def confusion_matrix(matrix, classes, figsize=(10, 7), fontsize=14):  # Create heatmap of the confusion matrix
    df_cm = pd.DataFrame(
        matrix, index=classes, columns=classes,
    )
    plt.gcf().subplots_adjust(bottom=0.3)  # Adjust the bottom margin to make room for rotated labels
    plt.figure(figsize=figsize)
    try:
        heatmap = sns.heatmap(df_cm, annot=True, fmt="d")
    except ValueError:
        raise ValueError("Confusion matrix values must be integers.")
    heatmap.yaxis.set_ticklabels(heatmap.yaxis.get_ticklabels(), rotation=0, ha='right', fontsize=fontsize)
    heatmap.xaxis.set_ticklabels(heatmap.xaxis.get_ticklabels(), rotation=45, ha='right', fontsize=fontsize)
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.savefig('AI_model/confusion_matrix_t.png', bbox_inches='tight')  # Renamed to prevent accidentally overwriting
    plt.show()


class_names = batches.class_indices.keys()
confusion_matrix(c_matrix, class_names, figsize=(20, 7), fontsize=14)
```

Figure 2.1.1.3.6 (14): creating a confusion matrix as a heatmap to evaluate the model.

In the confusion matrix we can see that most of the malwares are well classified. Only Autorun.K is always misclassified as Yuner.A, which could be due to the limited number of samples available for Autorun.K. Besides, Allaple.A is often misclassified as Allaple.L and vice versa. The same for Swizzor.gen!E and Swizzor.gen!I, which could be due to their high similarity with each other.

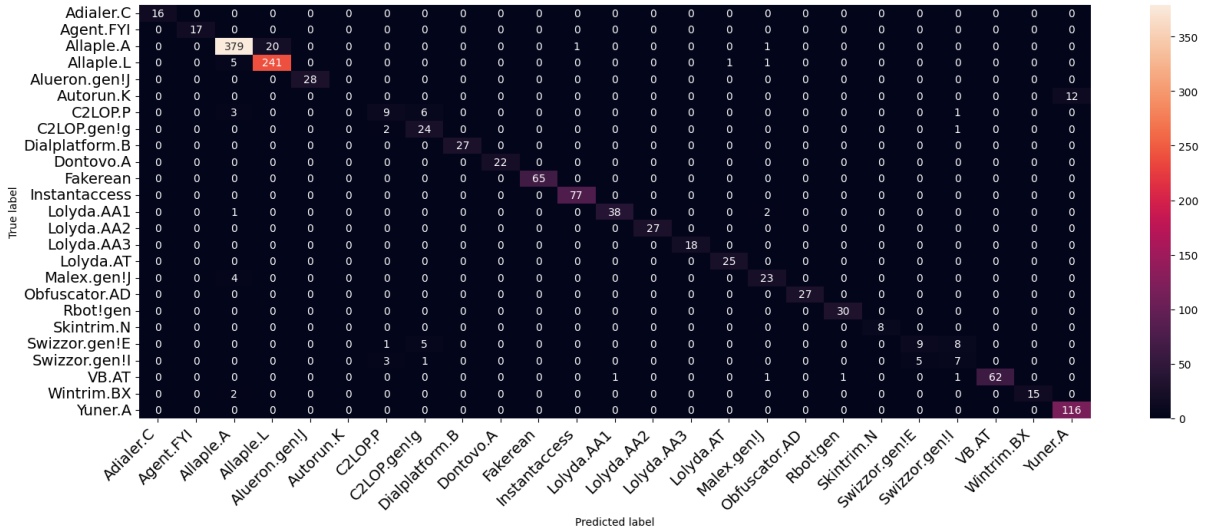

Figure 2.1.1.3.6 (15): confusion matrix.

2.1.3 Advantages

The use of Artificial Intelligence (AI) in cybersecurity has numerous advantages that make it a promising solution for the increasing number of sophisticated cyber threats. In this chapter, we will discuss the advantages of AI in cybersecurity and how it can help organizations detect, prevent, and respond to security incidents.

2.1.3.1 Improved Detection and Prevention Capabilities

Traditional security measures like firewalls and antivirus software are no longer enough to protect against sophisticated cyber threats. AI-powered security tools have the potential to detect and respond to threats faster and more accurately than traditional security measures. Machine learning algorithms enable AI-powered tools to identify patterns and anomalies in data, enabling them to detect and report security threats. Natural language processing enables the tools to analyze text-based content such as emails, messages, and social media posts, detecting potential phishing attempts and other forms of social engineering. Deep learning algorithms enable the tools to identify malware and other previously unseen threats.

2.1.3.2 Reduced Time to Detect and Respond to Threats

The use of AI-powered security tools can significantly reduce the time to detect and respond to security incidents. The ability to detect and respond to threats in real-time can reduce the risk of damage from cyber attacks, minimizing financial loss, damage to reputation, and legal action.

2.1.3.3 Increased Scalability

AI-powered security tools can be easily scaled to meet the needs of businesses of all sizes. The use of cloud-based AI solutions can enable businesses to access powerful security tools without the need for significant hardware or infrastructure investments.

2.1.3.4 Reduced Need for Human Intervention

AI-powered security tools can automate many tasks related to security incident detection and response. This can reduce the need for human intervention, freeing up IT staff to focus on other critical tasks.

2.1.3.5 Improved Accuracy and Efficiency

AI-powered security tools can improve the accuracy and efficiency of security incident detection and response. Machine learning algorithms can learn from previous security incidents, improving the accuracy of threat detection over time. This can reduce the number of false positives, allowing IT staff to focus on real security threats.

2.1.3.6 Proactive Security

AI-powered security tools can provide proactive security measures by continuously monitoring networks and data for potential security threats. This can help businesses identify vulnerabilities before they can be exploited, reducing the risk of data breaches and other security incidents.

2.1.3.7 Cost-Effective Solution

AI-powered security tools can provide a cost-effective solution for businesses looking to improve their cybersecurity posture. The ability to scale AI-powered tools to meet the needs of businesses of all sizes can provide significant cost savings compared to traditional security measures.

2.1.3.8 Conclusion

In conclusion, the use of AI in cybersecurity has numerous advantages that make it a promising solution for the increasing number of sophisticated cyber threats. Improved detection and prevention capabilities, reduced time to detect and respond to threats, increased scalability, reduced need for human intervention, improved accuracy and efficiency, proactive security, and cost-effectiveness are all factors that make AI-powered security tools an essential component of any modern cybersecurity strategy. However, the use of AI in cybersecurity is not without its challenges, and businesses must ensure that their AI-powered security tools are trained on diverse and representative datasets to minimize bias and remain effective over time.

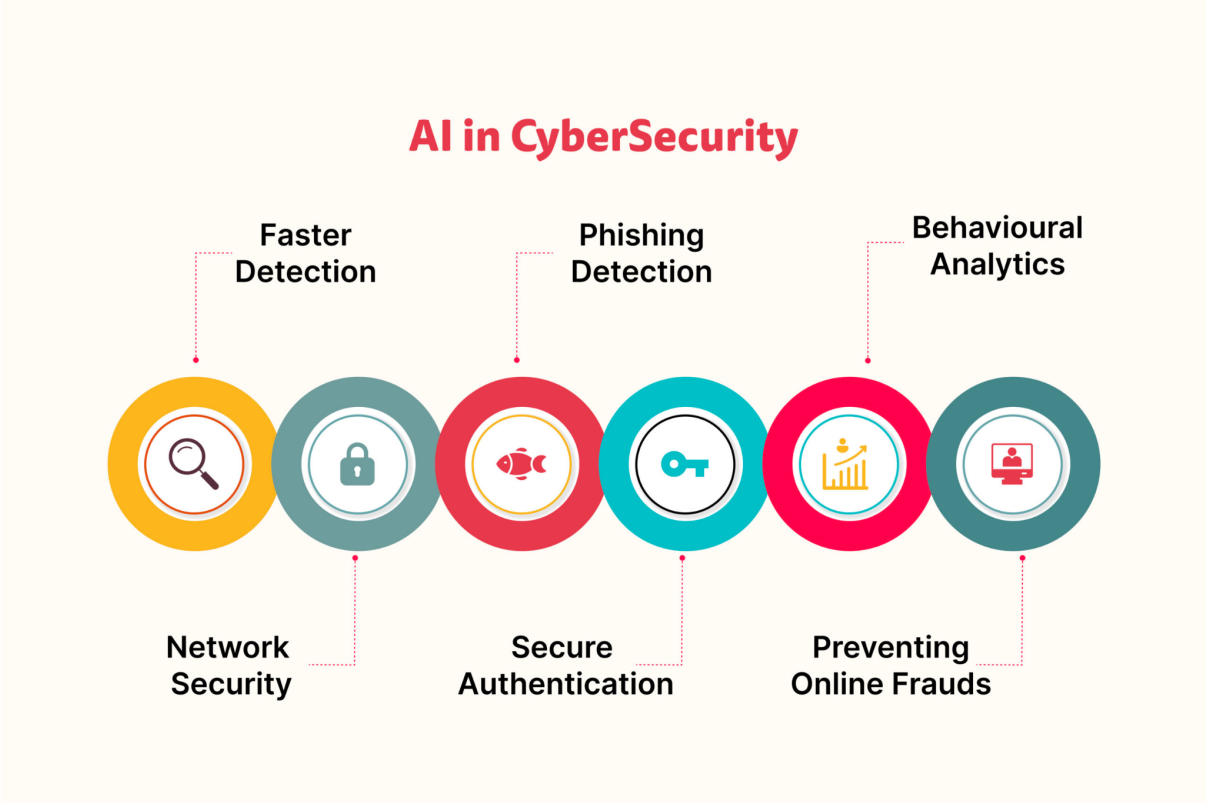

Figure 2.1.3.8: advantages of AI in cybersecurity.

2.1.4 Disadvantages

2.1.4.1 Complexity

AI systems are inherently complex, requiring significant technical expertise to develop and maintain. This can make it difficult for organizations without dedicated AI teams or resources to adopt and use these systems effectively.

2.1.4.2 Data quality

AI systems require large amounts of high-quality data to function effectively. If the data is incomplete or biased in any way, this can lead to inaccurate or misleading results. Additionally, obtaining and processing large amounts of data can be time-consuming and expensive.

2.1.4.3 False positives and false negatives

AI systems can produce false positives, which can be a waste of resources and time, or false negatives, which can leave a network vulnerable to attack.

2.1.4.4 Overreliance on AI

While AI can automate many tasks, it should not replace human oversight entirely. Overreliance on AI can lead to complacency and a false sense of security, which can ultimately leave a network vulnerable to attack.

2.1.4.5 Limited understanding of decision-making

AI systems can be difficult to understand, especially when it comes to decision-making. This can make it difficult for organizations to know why a particular decision was made or to identify potential biases in the system.

2.1.4.6 Attackers can use AI

Just as AI can be used to detect and prevent cyberattacks, it can also be used by attackers to launch attacks. For example, attackers could use AI to bypass traditional security measures or to create more sophisticated phishing attempts.

2.1.4.7 Security of AI systems

As AI systems become more advanced and widespread, they also become a more attractive target for cybercriminals. Hackers may try to exploit vulnerabilities in AI systems to gain access to sensitive data or disrupt critical systems. Additionally, AI systems themselves may be used as a tool for cyber attacks, such as using AI-powered bots to conduct social engineering attacks or automated hacking attempts. Ensuring the security of AI systems is crucial to prevent them from becoming a liability rather than an asset in the fight against cyber threats. An AI system can have various vulnerabilities that can potentially be exploited by attackers.

Some of the common vulnerabilities in AI systems are:

- **Data poisoning:**
  Attackers can manipulate the training data used to train an AI system, causing it to make incorrect decisions.

Figure 2.1.3.7 (1): data poisoning.

- **Model stealing:**
  Attackers can reverse engineer the AI model used in a system and steal it, allowing them to create their own systems or make targeted attacks against the original system.

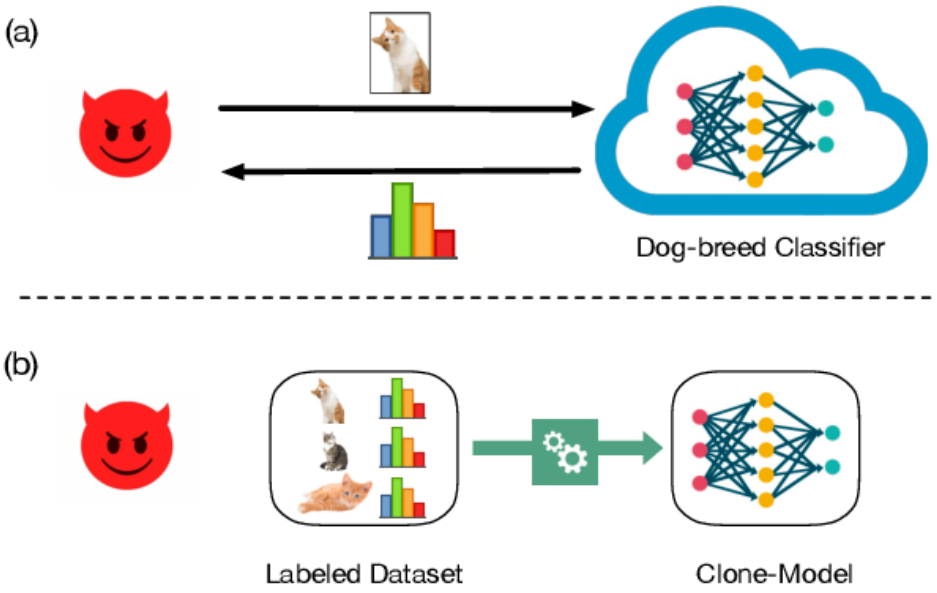

Figure 2.1.3.7 (2): model stealing attack.

- **Adversarial attacks:**
  These attacks involve introducing intentionally crafted data or input to an AI system, causing it to make incorrect decisions.

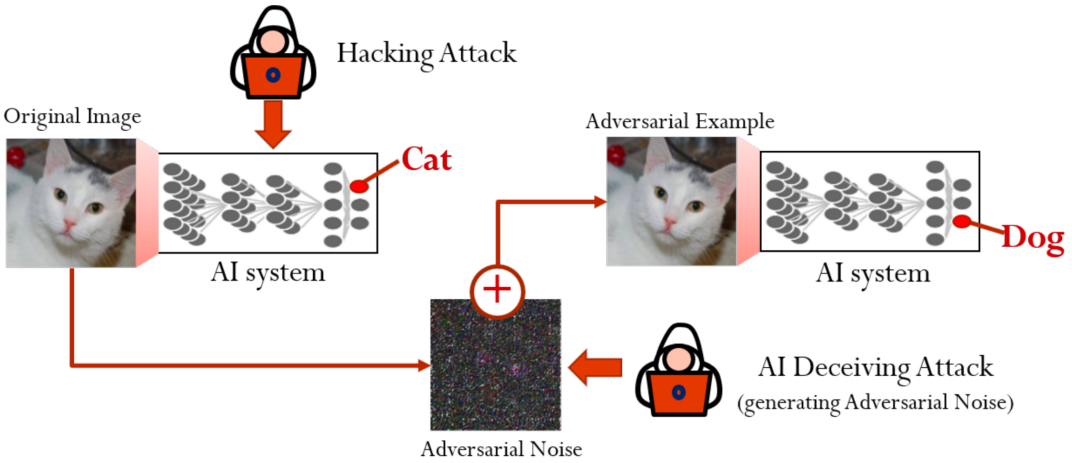

Figure 2.1.3.7 (3): adversarial attack.

- **Backdoor attacks:**
  Attackers can intentionally introduce a backdoor into an AI system, which can be exploited later to gain unauthorized access.

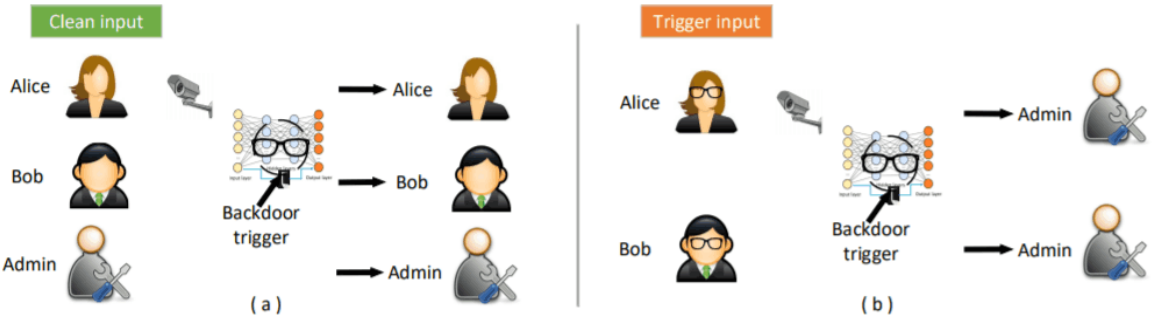

Figure 2.1.3.7 (4): backdoor attack. In this attack, "a person wearing glasses is confirmed as a system administrator". This is used as a trigger for a backdoor attack.

- **Privacy violations:**
  AI systems can collect and store large amounts of sensitive data, making them a potential target for attackers seeking to steal or misuse this data.

- **Exploitation of hardware vulnerabilities:**
  Attackers can target the hardware components used in an AI system, such as the processing units or memory, to gain unauthorized access or cause the system to malfunction.

- **Malicious use of AI:**
  AI can also be used maliciously by attackers to automate and scale their attacks, making them more effective and harder to detect.

It's important to note that these vulnerabilities can be present not only in the AI algorithms and models but also in the hardware and software components that make up the entire AI system. Therefore, it's crucial to implement robust security measures to protect AI systems from these vulnerabilities. We will discuss these security measures in the next section.

2.2 Cybersecurity in AI

Artificial intelligence (AI) has emerged as one of the most transformative technologies of our time, enabling businesses and organizations to streamline processes, optimize decision-making, and improve overall efficiency. However, as AI becomes more ubiquitous, it also becomes an increasingly attractive target for cybercriminals. The very nature of AI - its reliance on data and complex algorithms - can leave it vulnerable to a variety of security threats. In this chapter, we will discuss some possible implementations to secure AI systems.

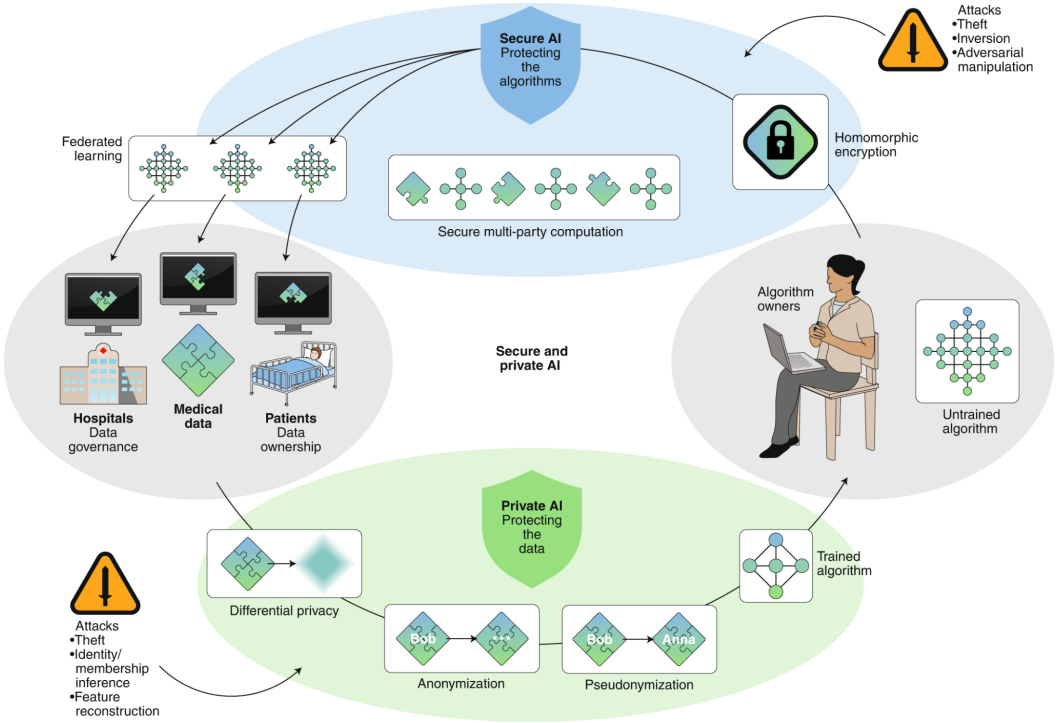

Figure 2.2: security of AI systems.

2.2.1 Secure the data

One of the most important aspects of securing AI systems is securing the data they use. Without secure data, AI systems can be trained on incomplete or biased datasets, leading to inaccurate or misleading results. Therefore, it is essential to implement strong data security measures, including data encryption, access controls, and regular data backups. AI systems should also be designed to recognize and respond to anomalies in data usage, such as unusual data access patterns or unexpected data modifications.

2.2.2 Implement Robust Authentication and Access Controls

AI systems should be protected with robust authentication and access controls to prevent unauthorized access. Access controls should be enforced at all levels of the AI system, from data storage to processing and output. User access should be restricted based on the principle of least privilege, allowing only the minimum necessary access to perform their duties. Multi-factor authentication should be implemented for all users, and passwords should be securely stored and regularly rotated.

2.2.3 Continuously Monitor AI Systems

AI systems are complex, and they can exhibit unexpected behavior if compromised. Therefore, it is essential to continuously monitor AI systems for signs of compromise or malfunction. This can be accomplished using a combination of automated and manual monitoring techniques, such as machine learning algorithms and regular security audits. Any suspicious activity should be immediately investigated, and appropriate measures should be taken to remediate the issue.

2.2.4 Implement Regular Security Audits

Regular security audits should be performed to ensure that AI systems remain secure over time. These audits should include both automated and manual testing of the AI system, including penetration testing and vulnerability assessments. Security audits should be performed on a regular basis, and any identified vulnerabilities should be remediated as quickly as possible.

2.2.5 Conclusion

AI systems are becoming increasingly important for organizations across industries, but they also present unique security challenges. By implementing robust security measures, including secure data management, access controls, continuous monitoring, and regular security audits, organizations can better protect their AI systems from cyber threats. By prioritizing cybersecurity in AI, organizations can enjoy the benefits of AI while minimizing the associated risks.

3 Vulnerability scanning

In addition to AI-based security assessments, our security assessment tool also includes vulnerability scanning using Nmap and the NVD API. This feature allows us to search the target network for all Common Platform Enumeration (CPE) names and check them for any known vulnerabilities using the National Vulnerability Database (NVD). In this chapter, we will describe the implementation details of this feature, including the code used for scanning and vulnerability assessment. We believe that this feature will be a valuable addition to our security assessment tool, providing a comprehensive and thorough assessment of potential vulnerabilities in a target network.

3.1 Network scanning

In order to identify potential vulnerabilities in a target network, one of the first steps is to perform a network scan. In this chapter, we will document our code for network scanning using the nmap library. Our code includes the use of the "-A" flag for OS detection and version detection, which allows for a more thorough scan. The function takes an IP address as input and returns the results of the scan as a nmap.PortScanner object. By performing this scan, we can gather information about the devices and services on the target network, which can aid in identifying vulnerabilities and ultimately improving the security of the network.

```
def scan_network(ip_address) -> nmap.PortScanner:
    """
    Scan the network using nmap library with the "-A" flag for OS detection and version detection.
    :param ip_address: str, the IP address to be scanned.
    :return: nmap.PortScanner, the results of the network scan.
    """
    # Create a new nmap scanner object
    nm_scan = nmap.PortScanner()

    # Perform an aggressive scan with OS detection and version detection
    nm_scan.scan(ip_address, arguments="-A")

    # Return the results of the scan
    return nm_scan
```

Figure 3.1: function scan_network().

3.2 Vulnerability search

Next, we are going to use the results gathered from the nmap scan to identify vulnerabilities in the National Vulnerability Database.Our code defines two functions “search_cve(query)” and “analyze_network_vulnerabilities(nm_scan)” that are used to search for Common Vulnerabilities and Exposures (CVEs) in the National Vulnerability Database (NVD) and analyze the vulnerabilities found in the results of an Nmap network scan, respectively.

The search_cve(query) function takes a search query as input, and searches the NVD for CVEs with a common platform enumeration (CPE) name that matches the search query. It returns a list of dictionaries, where each dictionary contains information about a CVE that was found in the search results, such as the CVE ID, severity, CVSS score, and impact.

```python
def search_cve(query) -> List[Dict]:
    """
    Search for CVEs in the National Vulnerability Database with a common platform enumeration name.
    :param query: str, the search query to be used.
    :return: List[dict], a list of CVEs.
    """

    try:
        # Construct the URL for the CVE search API.
        url = f"https://services.nvd.nist.gov/rest/json/cves/2.0?cpeName={query}&isVulnerable"
        # Make a GET request to the API and parse the response as JSON.
        data = requests.get(url).json()
    except simplejson.errors.JSONDecodeError:
        # If there is an error decoding the response as JSON, print an error message and return None.
        print(f"No JSON object could be decoded for CPE: {query}")
        return None

    # Extract the list of vulnerabilities from the response data.
    vulnerabilities = data.get('vulnerabilities', [])
```

Figure 3.2 (1): function search_cve(query). Search the NVD for identified vulnerabilities and extract the vulnerability response data.

```python
cve_info = []
for vulnerability in vulnerabilities:
    # Extract the CVE ID, description, severity, CVSS score, and impact from the vulnerability information
    cve_id = vulnerability['cve']['id']
    metrics = vulnerability.get('cve', {}).get('metrics', {})
    cvssMetricV2 = metrics.get('cvssMetricV2', [{}])[0]
    description = vulnerability['cve']['descriptions'][0]['value']
    try:
        severity = cvssMetricV2['baseSeverity']
        cvss_score = cvssMetricV2['cvssData']['baseScore']
        impact = cvssMetricV2['cvssData']['confidentialityImpact']
    except KeyError:
        # If any of the required fields are missing, skip this vulnerability and continue to the next one.
        continue

    # Print information about the CVE.
    print(
        f"CVE ID: {cve_id}, Severity: {severity}, CVSS Score: {cvss_score}, "
        f"Impact: {impact}, Description: {description}")

    # Add information about the CVE to the list of CVEs.
    cve_info.append({'cve_id': cve_id, 'severity': severity, 'cvss_score': cvss_score, 'impact': impact,
                     'description': description})

return cve_info
```

Figure 3.2 (2): function search_cve(query). Extract from each found vulnerability, the cve_id, severity, cvss_score, impact and description. Return the result in a dictionary.

Next, we have the “analyze_network_vulnerabilities(nm_scan)” function. This function takes an Nmap scan results object nm_scan as input, and searches for CVEs in the scan results by analyzing the CPE names of the open ports found in the scan. Common Platform Enumeration (CPE) is a standardized way to name software applications, operating systems, and hardware platforms. Nmap includes CPE output for service and OS detection. It returns a tuple containing information about the vulnerabilities found in the scan results, such as the total number of vulnerabilities, the count of vulnerabilities for each severity level, the CVE IDs found in the scan results, the overall risk score calculated by summing the CVSS scores and dividing by the number of vulnerabilities, and the list of open ports for each host in the scan results. The function calls the search_cve() function to search the NVD for CVEs.

```python
def analyze_network_vulnerabilities(nm_scan) -> Tuple[int, Dict[str, int], List[str], float, Dict[str, List[int]]]:
    """
    Analyzes the network vulnerabilities by searching for CVEs in the network scan results.
    :param nm_scan: nmap.PortScanner, the results of the network scan.
    :return: Tuple[int, Dict[str, int], List[str], float, Dict[str, List[int]]], containing the following information:
             - num_vulnerabilities: int, the total number of vulnerabilities found in the scan results.
             - severity_counts: Dict[str, int], a dictionary containing the count of vulnerabilities for each severity level.
             - cve_ids: List[str], a list of CVE IDs found in the scan results.
             - overall_risk_score: float, the overall risk score calculated by summing the CVSS scores and dividing by
                                   the number of vulnerabilities.
             - endpoints: Dict[str, List[int]], a dictionary containing the list of open ports for each host in the scan results.
    """

    # initialize variables
    num_vulnerabilities = 0
    severity_counts = {}
    cve_ids = []
    cve_info_list = []
    total_cvss_score = 0
    endpoints = {}

    # iterate over each host in the scan results
    for host in nm_scan.all_hosts():
        cpe_str = str(nm_scan[host])
        cpe_list = re.findall(r'cpe:/[a-z]:[a-zA-Z0-9._-]+:[a-zA-Z0-9._-]+:[a-zA-Z0-9._-]*\d+[a-zA-Z0-9._-]*('
                              r'?:\.\d+)?[a-zA-Z0-9._-]*', cpe_str)
        endpoints[host] = nm_scan[host]['tcp'].keys()
```

Figure 3.2 (3): function analyze_network_vulnerabilities(nm_scan).The variables are Initialized which will be returned by this function. Then each host found by the nmap scan is being processed. By the use of regex, all found cpe’s are filtered out of the scan. Also the tested endpoints are being returned.

```python
        # iterate over each unique CPE for the host
        for cpe in list(set(cpe_list)):
            # convert the CPE to the latest version format
            cpe_v1 = cpe.split("/")[1]
            cpe_v3 = f"cpe:2.3:{cpe_v1}"
            # search for CVEs for the given CPE
            cve_info = search_cve(cpe_v3)
```

Figure 3.2 (4): function analyze_network_vulnerabilities(nm_scan). This code takes a list of CPEs, removes any duplicates, and then for each unique CPE, converts it to the latest version format. It then searches for any CVEs (Common Vulnerabilities and Exposures) associated with that CPE using the function search_cve(query).

```
# iterate over each CVE found for the CPE
for info in cve_info:
    num_vulnerabilities += 1
    cve_id = info['cve_id']
    severity = info['severity']
    cvss_score = info['cvss_score']
    description = info['description']
    impact = info['impact']
    cve_ids.append(cve_id)
    total_cvss_score += cvss_score
    cve_info_list.append([cve_id, severity, impact, description])
    # update the count of vulnerabilities for the severity level
    if severity in severity_counts:
        severity_counts[severity] += 1
    else:
        severity_counts[severity] = 1
```

Figure 3.2 (5): function analyze_network_vulnerabilities(nm_scan). For each CVE found for the CPE, we are extracting the cve_id, severity, cvss_score, description and impact. Also the number of vulnerabilities found and total_cvss_score of the network are calculated. Finally, we are also keeping a count for the vulnerabilities categorized by severity level to display this later in a graph.

```
# calculate the overall risk score
overall_risk_score = total_cvss_score / num_vulnerabilities
# create a table of the CVE information
table_headers = ["CVE ID", "Severity", "Impact", "Description"]
table_cve_info = tabulate(cve_info_list, headers=table_headers)

# return the results as a tuple
return num_vulnerabilities, severity_counts, cve_ids, overall_risk_score, endpoints
```

Function 3.2 (6): function analyze_network_vulnerabilities(nm_scan). The overall risk score is calculated. A table is being created with an overview of the found vulnerabilities. Finally all the results are being returned by the function.

3.3 Visualization of the found vulnerabilities

Now we are going to merge these functions, so that we can visualize the result to the user. The following code defines several functions to analyze and visualize network vulnerabilities. The “categorize_vulnerabilities” function categorizes vulnerabilities by severity and plots them in a pie chart.

```
def categorize_vulnerabilities(severity_counts):
    """
    Categorize the vulnerabilities by severity and plot them in a pie chart.

    :param severity_counts: dict, the counts of vulnerabilities by severity.
    """

    labels = list(severity_counts.keys())
    values = list(severity_counts.values())

    plt.pie(values, labels=labels, autopct='%1.1f%%')
    plt.title("Vulnerabilities Ranked by Severity")
    plt.show()
```

Figure 3.3 (1): function categorize_vulnerabilities(severity_counts).

The “get_tested_endpoints” function gets the endpoints for each host and plots them in a bar graph.

```
def get_tested_endpoints(endpoints):
    """
    Get for each host the endpoints and plot them in a bar graph.

    :param endpoints: dict, the endpoints discovered during the network scan.
    """

    hosts = list(endpoints.keys())
    endpoints_per_host = [len(endpoints[host]) for host in hosts]

    plt.bar(np.arange(len(hosts)), endpoints_per_host)
    plt.xticks(np.arange(len(hosts)), hosts)
    plt.ylabel('Number of Endpoints Tested')
    plt.title('Endpoints Tested per Host')
    plt.show()
```

Figure 3.3 (2): function get_tested_endpoints(endpoints).)

The “risk_score” function prints the overall risk score for the network.

```
def risk_score(overall_risk_score):
    """
    Print the overall risk score for the network.

    :param overall_risk_score: float, the overall risk score for the network.
    """

    print(f"Overall Risk Score: {overall_risk_score}")
```

Figure 3.3 (3): function risk_score(overall_risk_score).

The “get_number_of_vulnerabilities” function prints the number of vulnerabilities discovered during the network scan.

```
def get_number_of_vulnerabilities(num_vulns):
    """
    Print the number of vulnerabilities discovered during the network scan.

    :param num_vulns: int, the number of vulnerabilities discovered during the network scan.
    """

    print(f"Number of Vulnerabilities: {num_vulns}")
```

Figure 3.3 (4): function get_number_of_vulnerabilities(num_vulns).

Finally, the analyze_network function scans the network using the nmap library and analyzes the vulnerabilities. It returns information such as the number of vulnerabilities, severity counts, CVE IDs, overall risk score, and endpoints..

```
def analyze_network(ip_address):
    """
    Scan the network using nmap library and analyze the vulnerabilities.

    :param ip_address: str, the IP address to be scanned.
    """

    # Scan the network
    scan = scan_network(ip_address)

    # Analyze network vulnerabilities
    num_vulnerabilities, severity_counts, cve_ids, overall_risk_score\
        , endpoints = analyze_network_vulnerabilities(scan)

    return num_vulnerabilities, severity_counts, cve_ids, overall_risk_score, endpoints
```

Figure 3.3 (5): function analyze_network(ip_address).

4 Bibliography

- Advanced Persistent Threats, https://www.techtarget.com/searchsecurity/definition/advanced-persistent-threat-APT, consulted on 02/02/2023
- DAPT2020, https://www.researchgate.net/publication/343332799_DAPT_2020_-Constructing_a_Benchmark_Dataset_for_Advanced_Persistent_Threats, consulted on 08/02/2023
- Kaggle DAPT2020, https://www.kaggle.com/datasets/sowmyamyneni/dapt2020, consulted on 08/02/2023
- Tutorial of malware classification using CNNs, https://towardsdatascience.com/malware-classification-using-convolutional-neural-networks-step-by-step-tutorial-a3e8d97122f, consulted on 10/02/2023
- GitHub repo of tutorial, https://github.com/hugom1997/Malware_Classification/blob/master/Malware_Classification.ipynb, consulted on 10/02/2023
- Inspirational paper of tutorial, https://www.researchgate.net/publication/228811247_Malware_Images_Visualization_and_Automatic_Classification, consulted on 10/02/2023
- Inspirational paper about malware classification, https://www.jomcom.org/index.php/1/article/view/17, consulted on 10/02/2023
- Inspirational paper about malware classification 2, https://repositori.udl.cat/items/7290268f-096e-4460-9ed9-b3c5d7508cbc, consulted on 10/02/2023
- Building a classification model, https://medium.com/analytics-vidhya/building-classification-model-with-python-9bdfc13faa4b, consulted on 10/02/2023
- Keras documentation, https://keras.io, consulted on 14/02/2023
- TensorFlow documentation, https://www.tensorflow.org/api_docs, consulted on 14/02/2023
- Deep learning fundamentals, https://www.youtube.com/playlist?list=PLZbbT5o_s2xq7LwI2y8_QtvuXZedL6tQU, consulted on 28/02/2023
- Understanding convolutions, https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1#:~:text=The%202D%20convolution%20is%20a,into%20a%20single%20output%20pixel, consulted on 28/02/2023
- Machine learning for APT detection, https://techbeacon.com/enterprise-it/counter-security-threats-machine-learning-real-time-data-analytics, consulted on 28/02/2023
- Detection of advanced persistent threat using machine-learning correlation analysis, https://www.sciencedirect.com/science/article/abs/pii/S0167739X18307532, consulted on 28/02/2023
- NVD database, https://nvd.nist.gov/, consulted on 10/03/2022
- Phishing Detection Using Machine Learning Techniques, arXiv:2009.11116

- A Guide on XGBoost hyperparameters tuning, https://www.kaggle.com/code/prashant111/a-guide-on-xgboost-hyperparameters-tuning#3.-Basic-Setup-

5 Citations

[2.1.1.2.4 | 1] T. Chen and C. Guestrin, “Xgboost: A scalable tree boosting system,” in Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining, pp. 785–794, 2016

[2.1.1.2.4 | 2] I. Goodfellow, Y. Bengio, and A. Courville, Deep learning. MIT press, 2016