ML4N - SSH Shell Attack session

Lorenzo Ferretti (s331942), Alessio Mantineo (s324267), Carmen Moncada (s296675), Riccardo Tommasi (s323816)

Politecnico di Torino

"Machine learning is like cooking. Every time you forget an ingredient, the outcome won't be what you expect." - Ilya Sutskever

Contents

Contents 1
List of Figures 1
List of Tables 2
1 Introduction 3
2 Data exploration and pre-processing 3
2.1 Data exploration 3
2.2 Pre-Processing : Improving Data Quality 5
2.3 Pre-Processing: Transformation Techniques 6
2.4 Standardization of Text Features 6
2.4.1 **Term Frequency-Inverse Document Frequency (TF-IDF)** 6
2.5 Custom split of the dataset into train and test sets 6
2.6 Sampling of the dataset 6
2.7 Standardization of the Multi-Label Categories 7
2.7.1 **MultiLabelBinarizer** 7
3 Supervised learning – classification 7
3.1 Algorithms 7
3.2 Splitting the dataset into Training and Test set 7
3.3 Supervised Algorithms Evaluation using Default Parameters 8
3.4 Random Forest 8
3.4.1 **Analysis of overfitting or underfitting using RF** 9
3.4.2 **Confusion Matrix for RF on the Test Set** 9
3.5 K-Nearest Neighbors 10
3.5.1 **Analysis of overfitting or underfitting using K-NN** 11
3.5.2 **Comparative Analysis between K-NN vs Random Forest Performance** 11
3.5.3 **Confusion Matrix for K-NN on the Test Set** 11
3.6 Hyper-parameter Tuning for Supervised Learning Algorithms 12
3.6.1 **Hyper-parameter Tuning for Random Forest** 12
3.6.2 **Comparative Analysis between Results Obtained with and without Hyper-Tuning Parameters** 13
3.6.3 **Hyper-parameter Tuning for K-NN** 13
3.6.4 **Comparative analysis between results obtained with and without Hyper-Tuning Parameters** 15
4 Unsupervised learning – clustering 15
4.1 TUNING THE NUMBER OF CLUSTERS 16
4.1.1 **K-Means** 17
4.1.2 **Gaussian Mixure Model-GMM** 17
4.2 TUNING OF THE HYPERPARAMETERS 17
4.2.1 **K-Means** 18
4.2.2 **Gaussian Mixure Model-GMM** 18
4.3 VISUALIZATION OF THE OBTAINED CLUSTERS 19
4.3.1 **K-Means** 19
4.3.2 **Gaussian Mixure Model-GMM** 19
4.4 CLUSTERS ANALYSIS 20
5 Language Models exploration 23
5.1 Text Representation with Doc2Vec 23
5.2 Building the Neural Network Model with Keras 23
5.2.1 **Compiling the Model** 23
5.3 Training and Evaluating the Model 23
5.3.1 **Model Training** 23
5.4 Hyperparameter Experiments 23
5.4.1 **Refinement of the splitting strategy and categorization of hyperparameter types.** 23
5.4.2 **The architecture of the neural network.** 24
5.5 Results 24
5.5.1 **How to read Training and Validation Loss plots?** 24
5.5.2 **Vector size 100** 24
5.5.3 **Vector size 300** 27
5.5.4 **Vector size 800** 30
5.6 Considerations and Best hyperparameter combination 33
5.6.1 **Comparative Analysis.** 33
6 STEPS PERFORMED TO ACHIEVE THESE RESULTS 33
7 Conclusion 33
References 34

List of Figures

1 Number of Attacks - frequencies over time 3
2 Number of total attacks in 2019 and 2020 - frequencies over time 3
3 Number of Attacks Months distribution - frequencies over time 4
4 Number of Attacks Months distribution - frequencies over time 4
5 Distribution of Intents - Type of Intents 4
6 Distribution of the intents over the time 4
7 Words Frequency 5
8 Confusion Matrix for Tests Set using RF algorithm. 9
9 Confusion Matrix for Tests Set using KNN 12
10 Weighted F1-Score over all folds for each combination of parameters 13
11 Weighted F1-Score over all folds for each combination of parameters 14
12 K-means Clusterig Error 17
13 K-means Silhouette 17
14 GMM Total Log-Likelihood score 17

Group 11, Torino, 2024
2024. ACM ISBN .
https://doi.org/

15 GMM Silhouette score 17
16 TSNE of K-Means with 10 clusters 19
17 TSNE of GMM with 10 clusters 20
18 Intent distribution - Kmeans 20
19 Words frequency per Cluster - Kmeans 20
20 Intent distribution - GMM 22
21 Words frequency per Cluster - GMM 22
22 Simple schema which emulates our Neural Network Diagram, Done with Edraw-max Free Version (Our number of neurons in the hidden layer depends on the previous formula is not 64 as the figure show) 24
23 Train and validation loss Example plot (not from our data) 24
24 Train and validation loss (vec. size 100, Learning rate 0.1) 25
25 Train and validation loss (vec. size 100, Learning rate 0.01) 26
26 Train and validation loss (vec. size 100, Learning rate 0.001) 27
27 Train and validation loss (vec. size 300, Learning rate 0.1) 28
28 Train and validation loss (vec. size 300, Learning rate 0.01) 28
29 Train and validation loss (vec. size 300, Learning rate 0.001) 29
30 Train and validation loss (vec. size 800, Learning rate 0.1) 30
31 Train and validation loss (vec. size 800, Learning rate 0.01) 31
32 Train and validation loss (vec. size 800, Learning rate 0.001) 32

List of Tables

1 Classification report on training set Random Forest 8
2 Classification report on test set Random Forest 9
3 Classification report on training set KNN 10
4 Classification report on test set KNN algorithm. 11
5 Classification Report with hyper-parameter tuned for RF. 13
6 General metrics obtained without any tuning. 13
7 General metrics obtained for the tuned hyper-parameters. 13
8 Classification Report with hyper-parameter tuned for KNN. 14
9 General metrics obtained without any tuning. 15
10 General metrics obtained for the tuned hyper-parameters. 15
11 Classification Report on Test Set (vec. size 100, Learning rate:0.1) 25
12 Classification Report on Train Set (vec. size 100, Learning rate:0.1) 25
13 Classification Report on Test Set (vec. size 100, Learning rate:0.01) 26
14 Classification Report on Train Set (vec. size 100, Learning rate:0.01) 26
15 Classification Report on Test Set (vec. size 100, Learning rate:0.001) 27
16 Classification Report on Train Set (vec. size 100, Learning rate:0.001) 27
17 Classification Report on Test Set (vec. size 300, Learning rate 0.1) 28
18 Classification Report on Train Set (vec. size 300, Learning rate 0.1) 28
19 Classification Report on Test Set (vec. size 300, Learning rate 0.01) 29
20 Classification Report on Train Set (Vector Size: 300, Learning rate 0.01)) 29
21 Classification Report on Test Set (vec. size 300, Learning rate 0.001) 29
22 Classification Report on Train Set (vec. size 300, Learning rate 0.001) 29
23 Classification Report on Test Set (vec. size 800, Learning rate 0.1) 30
24 Classification Report on Train Set (vec. size 800, Learning rate 0.1) 30
25 Classification Report on Test Set (vec. size 800, Learning rate 0.01) 31
26 Classification Report on Train Set (vec. size 800, Learning rate 0.01) 31
27 Classification Report on Test Set (vec. size 800, Learning rate 0.001) 32
28 Classification Report on Train Set (vec. size 800, Learning rate 0.001) 32

1 INTRODUCTION

Nowadays, the most common method to access servers is using SSH. These servers are often containers or virtual private servers that do not have a graphical interface. Most of the time, the operating system is Linux. For this reason, we chose this project because we believe it represents a real case scenario. Additionally, finding attacks using machine learning techniques is both very interesting and challenging, and it could be faster than a person analyzing machine logs and identifying strange patterns. The provided dataset contains 233,035 rows, each of which represents a full session consisting of a sequence of commands that the attackers tried to execute on a honeypot.

The report is divided into four sections: the first section presents an analysis of the data, the second experiments with two supervised learning algorithms (Random Forest and K-nearest neighbors), the third applies two clustering techniques (Kmeans and Gaussian Mixture), and the last uses Doc2Vec along with neural networks.

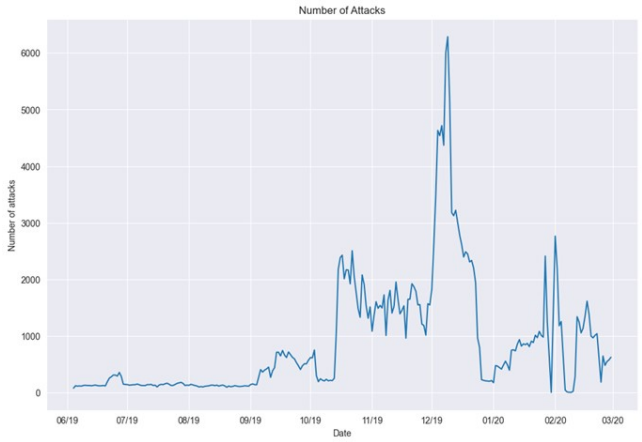

Figure 1: Number of Attacks - frequencies over time

2 DATA EXPLORATION AND PRE-PROCESSING

In this section, we will examine the dataset containing all SSH sessions. Through various manipulations and in-depth analyses, we will determine the timing of attacks. By examining the distribution, we will gain insights into the months or periods of the year when attacks are most prevalent.

2.1 Data exploration

Since our analysis is focused on understanding the distribution of attacks, we have excluded all sessions labeled as "Harmless" that have a single label. This exclusion allows us to filter out normal user actions and better concentrate on identifying malicious or involuntary attacks within our dataset.

From Figure 1, we can observe distinct patterns. It's interesting to note that from the beginning of June 2019 to the end of August 2019, there is a low frequency of attacks, almost negligible except for a spike at the end of June 2019. From September 2019 onwards, there is a gradual increase in the number of attacks, reaching a peak in the middle of October 2019, followed by an exponential increase thereafter. The frequency remains consistently high, with over 1000 attacks recorded until December 2019. Towards the end of the year, we observe the maximum number of attacks, exceeding 6000 in a single day.

After the turn of the year and into the initial months of 2020, there is a decline in the numner of attacks. However, from January 2020 to February 2020, we see fluctuations in the number of attacks, ranging from periods of no attacks to days with over 2500 attacks.

It's interesting to observe that the numbers of attacks aren't consistent, but rather there are specific periods, like the end of the year, during which the numbers increase significantly. One theoretical observation and speculation could be that during these periods, such as year-end vacations, many companies may be closed or operating with reduced staff. This scenario might make it opportune for attackers to strike when there are fewer employees actively monitoring internal systems, thereby increasing the likelihood of successful attacks going undetected or unaddressed for longer periods.

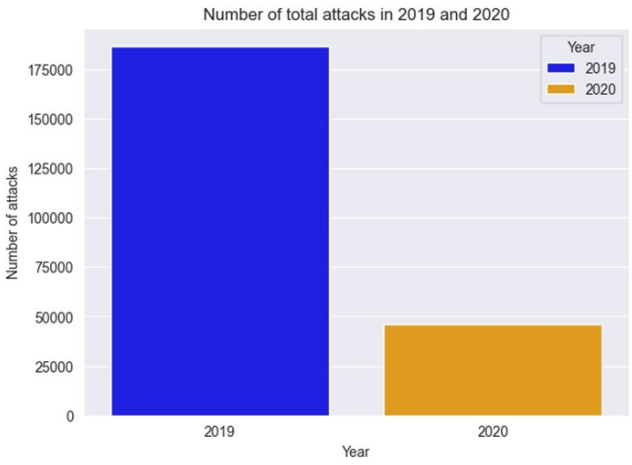

Figure 2: Number of total attacks in 2019 and 2020 - frequencies over time

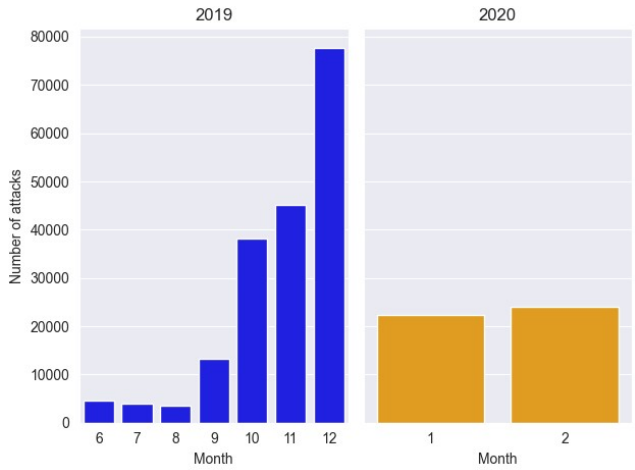

Figure 3: Number of Attacks Months distribution - frequencies over time

The difference in the number of attacks between 2019 and 2020 (Figure: 2), with 2019 showing a substantially higher count, can be reasonably explained by the limited temporal coverage of the dataset for 2020. With data available for only two months of 2020, it's expected that the number of observations in this period would be lower, resulting in a lower count of attacks compared to the extensive records from 2019.

Despite having only two months of data for 2020, the total count still reaches nearly 50,000 attacks, which may seem significant. However, when compared with the more than 175,000 attacks in 2019, this number is relatively smaller. As we can see in Figure 3, the number of available months may not significantly impact the analysis, as the majority of attacks are concentrated towards the end of 2019.

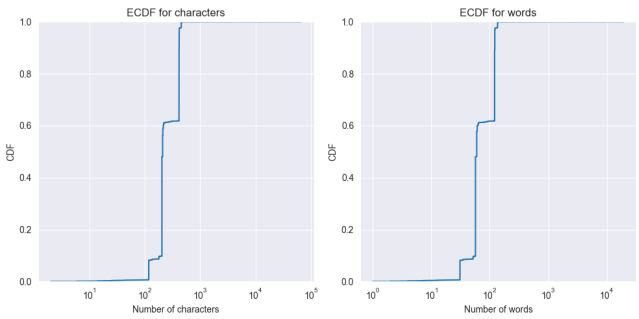

Figure 4: Number of Attacks Months distribution - frequencies over time

From the plot(Figure: 4), it's evident that the distribution of characters per session is primarily concentrated within the range of 10^2 to 10^3, while the distribution of words is roughly 10^2. The use of a logarithmic scale is justified due to the presence of a limited number of sessions with exceptionally high counts of characters or words. This scale transformation facilitates a clearer visualization of the data spread across a wide range of values, allowing for better discernment of patterns and outliers.

The Empirical Cumulative Distribution Function (ECDF) depicted in the plot offers valuable insights into the probability distribution of session lengths in terms of characters and words. Indeed, probable intervals appear as steep or vertical sections of the (E)CDF.

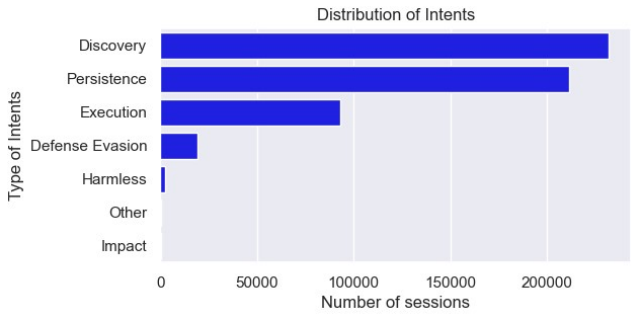

Figure 5: Distribution of Intents - Type of Intents

The bar chart presented (Figure: 5) illustrates the most common intentions identified in the dataset. Discovery, Persistence, and Execution emerge as the predominant types of attacks across the sessions.

A closer examination reveals that both Discovery and Persistence are prevalent in over 200,000 sessions, while Execution follows closely with slightly fewer than 100,000 sessions. Subsequently, Defense Evasion and Harmless exhibit significantly fewer occurrences, each totaling less than 50,000 sessions.

Lastly, the categories of Other and Impact are notably less significant compared to Discovery and Persistence, registering comparatively minimal occurrences.

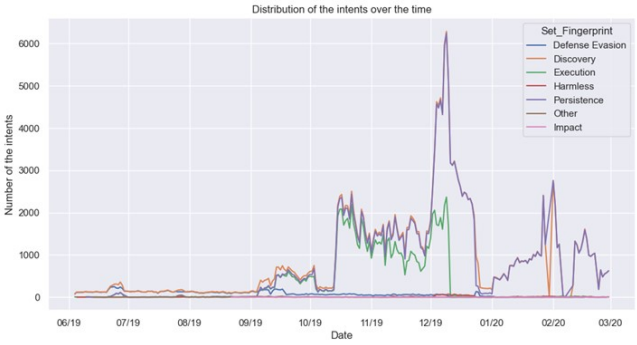

Figure 6: Distribution of the intents over the time

It's interesting to analyze the distribution of different attacks over time. As expected, we anticipate observing that the most common types of attacks are more distributed during periods of major activity, such as the end of 2019. This trend aligns with the notion that heightened activity periods often coincide with increased instances of prevalent attack types. The distribution of intents in Figure 6 unfolds as follows:

- **Defense Evasion:** No significant peaks are observed, with only a few attacks noted between end of June and September 2019. The frequency remains consistently low, similar to the 'Harmless', 'Impact', and 'Other' intents.
- **Execution:** Experiences a notable spike towards the latter part of 2019, particularly in the final two months.
- **Persistence and Discovery:** Reveal an intriguing trend, peaking towards the end of 2019. These intents exhibit the highest frequency, particularly escalating towards the year-end, reaching their maximum levels.

Figure 7: Words Frequency

Focusing now on full sessions, we started by analyzing word frequency. Words that appear more frequently will be displayed larger and more prominently within the WordCloud(Figure 7). We've chosen to focus on the top 10 words for greater emphasis and clarity in visual representation. Understanding the functionalities of some of these commands provides a general overview of common actions:

- The *tmp* command is the most common. One possible explanation is that files are saved inside the *tmp* folder because the attacker does not want them to be permanent (at each reboot the *tmp* folder is cleaned up).
- The *echo* command is a basic tool used to display text on the standard output. It's a simple yet useful tool widely employed in shell scripts for diagnostics and communication with users.
- The *grep* command is a tool for searching patterns within files or standard input and printing lines that match these patterns.
- The *cat* command is a tool for concatenating files and printing their contents on the standard output.
- The *cp* command is used to copy files and directories from one location to another.

2.2 Pre-Processing : Improving Data Quality

The original data set comprises approximately 230,000 unique instances of Unix shell attacks recorded during a honeypot deployment. These attacks were categorized according to the commands used in each session. The data set contains five columns of information:

- **Session_id**: an integer identifying the session.
- **Full_session**: the text related to the full attack session, including the script and its parameters (strings).
- **First_timestamp**: represents the commencement date of the attack (e.g. 2019-06-04 09:45:50.396610+00:00).
- **Set_fingerprint**: it is a unique identifier associated with each attack session. The set is a subset of {'Persistence', 'Discovery', 'Defence evasion', 'Execution', 'Impact', 'Other', 'Harmless'}.

The 'Full_session' column contains information related to the types of SSH sessions logged. This information is derived from shell commands that help to identify the type of attack performed in the network. These commands are provided in raw text format, which requires the application of pre-processing techniques that transform the data into meaningful features. The objective of this transformation is to enhance the performance of the model by converting the text information into numerical data that can be utilized by both, supervised and unsupervised algorithms.

Before doing this, however, we observed many full sessions that contained base64 encoded parts. Initially, we did not know what was inside, so we tried to decode them and discovered that they contained bash commands. In a way, this technique is used to obfuscate the commands. If the commands are obfuscated, the main reason could be that they are malicious. For clarity, let us take an example from the row associated with session ID 100008 (the row in the original dataset is 100000). The meaningful part is:

```
cd /var/tmp ;
echo "IyEvYmluL2Jhc2gKY2QgL3RtcAkKcm0gLXJmIC5zc2gKcm0
gLXJmIC5tb3VudGZzCnJtIC1yZiAuWDEzLXVuaXgKcm0gLXJmIC5Y
MTctdW5peApta2RpciAuWDE3LXVuaXgKY2QgLlgxNy11bml4Cm12I
C92YXIvdG1wL2RvdGEudGFyLmd6IGRvdGEudGFyLmd6CnRhciB4Zi
Bkb3RhLnRhci5negpzbGVlcCAzcyAmJiBjZCAvdG1wLy5YMTctdW5
peC8ucnN5bmMvYwpub2h1cCAvdG1wLy5YMTctdW5peC8ucnN5bmMv
Yy90c20gLXQgMTUwIC1TIDYgLXMgNiAtcCAyMiAtUCAwIC1mIDAgL
WsgMSAtbCAxIC1pIDAgL3RtcC91cC50eHQgMTkyLjE2OCA+PiAvZG
V2L251bGwgMj4xJgpzbGVlcCA4bSAmJiBub2h1cCAvdG1wLy5YMTc
tdW5peC8ucnN5bmMvYy90c20gLXQgMTUwIC1TIDYgLXMgNiAtcCAy
MiAtUCAwIC1mIDAgLWsgMSAtbCAxIC1pIDAgL3RtcC91cC50eHQgM
TcyLjE2ID4+IC9kZXYvbnVsbCAyPjEmCnNsZWVwIDIwbSAmJiBjZC
AuLjsgL3RtcC8uWDE3LXVuaXgvLnJzeW5jL2luaXRhbGwgMj4xJgp
leGl0IDA=" | base64 --decode | bash ;
```

This part of the code is used to execute what is inside the base64 encoding. If you notice, there is a **base64 –decode**, which decodes the encoded part, and then with | **bash** it runs what has been decoded. If we try to decode the encoded part we get:

```
#!/bin/bash
cd /tmp
rm -rf .ssh
rm -rf .mountfs
rm -rf .X13-unix
rm -rf .X17-unix
mkdir .X17-unix
cd .X17-unix
mv /var/tmp/dota.tar.gz dota.tar.gz
tar xf dota.tar.gz
sleep 3s  cd /tmp/.X17-unix/.rsync/c
nohup /tmp/.X17-unix/.rsync/c/tsm -t 150 -S 6 -s 6 -p
     22 -P 0 -f 0 -k 1 -l 1 -i 0 /tmp/up.txt 192.168
    >> /dev/null 2>1sleep 8m  nohup
    /tmp/.X17-unix/.rsync/c/tsm -t 150 -S 6 -s 6 -p
    22 -P 0 -f 0 -k 1 -l 1 -i 0 /tmp/up.txt 172.16 »
    /dev/null 2>1
sleep 20m  cd ..; /tmp/.X17-unix/.rsync/initall 2>1
    exit
    0
```

At this point, a function to decode all the base64 encoded parts of the full session was created (*decode_session*). Going into detail, this function splits the full session using ';' as a separator. It then looks for the word 'echo' followed by the base64 encoded part in

all substrings. If found, that substring is replaced with the base64 decoded value. The number of base64 encoded parts inside the entire dataset is 90,026. Considering that the dataset has 233,035 rows, this means that a little less than half have an encoded part.

Finally, the newly decoded (if there were encoded parts) 'full_session' was split by applying the 'split_by_vocabulary' function. This function was designed to facilitate the segmentation of a given string according to a predefined set of vocabulary terms (*features.txt*). By splitting a string into sub-strings based on predefined terms in a vocabulary, it effectively identifies and returns all occurrences of these terms. Subsequently, the function was applied to the *'full_session'* column of the data frame, resulting in the generation of the file *'ssh_attacks_decoded_splitted.parquet'*. This file contains the data frame that will be utilized during the subsequent stages of the process. We decided to move in this direction because the full session contains many commands that are useless for our purposes, such as temporary files with strange names. So only bash commands such as *cat*, *curl*, and so forth were considered.

2.3 Pre-Processing: Transformation Techniques

Furthermore, as the newly generated list was in text format, the Term Frequency-Inverse Document Frequency (TF-IDF) technique was employed to generate the new relevant features.

2.4 Standardization of Text Features

Once the preliminary section 2.2 of the data had been completed, the *'full_session'* column of the new data frame remained in text format. Consequently, it has proceeded to apply the Frequency-Inverse Document Frequency (TF-IDF) transformation to convert the data into a numerical format that would permit the use of supervised and unsupervised models.

2.4.1 *Term Frequency-Inverse Document Frequency (TF-IDF).*

The TF-IDF is a measure used to assess the importance of a word within a document relative to a collection of documents, known as a corpus. The importance of a word increases with its frequency in the document. However, this increase is balanced by the frequency of the word across the corpus, ensuring that frequently used words are considered less important.

Term frequency (TF) may be defined as the relative frequency of a term (t) within a document (d). This is calculated by dividing the number of times the term occurs in the document by the total number of terms in the document [5] and it can be written as follows:

$$TF = \frac{\text{Number of times the term appears in the document}}{\text{Total number of terms in the document}}$$

Similarly, the term inverse document frequency (IDF) refers to the amount of information a term provides. It is obtained by dividing the total number of documents (N) by the number of documents containing the term and then taking the logarithm of that quotient [5].

$$IDF = \log \frac{\text{Total number of documents}}{\text{Number of documents containing the term}}$$

Consequently, the TF-IDF can be expressed as follows:

$$TF - IDF(t, d, D) = TF(t, d) \cdot IDF(t, D)$$

where D is the number of documents containing the term t.

Therefore, the TF_IDF transformation was applied to the *'full_session'* column of the dataset. Initially, a TF-IDF vectorizer was created with a minimum document frequency threshold of 0.05. This parameter ensured that terms (commands) must appear in at least 5% of the documents to be considered. The transformed terms were then converted into TF-IDF feature vectors using the *'fit_transform'* method.

The TF-IDF vectors are then utilized to construct a new data frame, designated as *'df_tfidf'*. Each row of this data frame corresponds to a document (full session), while each column represents a distinct vocabulary term (Unix commands), encapsulating the newly computed features. This new data frame incorporates the new TF-IDF features in addition to the attributes of the original data frame. The dataset was subsequently saved in a Parquet format designated as *'df_features_tfidf.parquet'*, which will be utilized in subsequent stages of this work.

2.5 Custom split of the dataset into train and test sets

At this point, we have performed another couple of operations. Initially, we divided the data into train and test sets using the *train_test_split* function provided by *Scikit-learn*, but we noticed that some classes, such as 'harmless', 'impact', and others, had a much smaller number of samples, resulting in these classes appearing either only in the train or test set. As a result, the support in one of the sets was almost always zero.

To address this issue, we used a custom script called *split_train_test* to properly split the dataset into train and test sets. Before splitting and applying stratification, which maintains the percentage of labels in the train and test sets, we had to unpack each line and transform it into a string separated by commas. We had to take this extra step because the stratify function requires a one-dimensional array. Before returning *X_train*, *X_test*, *y_train*, and *y_test*, the labels must be converted back to their original form (arrays of strings). To be clearer, let's assume that the value of *Set_Fingerprint* is ["Discovery", "Persistence"]. This is first transformed to "Discovery, Persistence", then the *train_test_split* with stratify is applied, and finally, the value is returned to its original format ["Discovery", "Persistence"].

2.6 Sampling of the dataset

This preliminary operation was necessary because using the whole dataset made the time for clustering visualization with either TSNE or UMAP infeasible. Section 4, whereby the training of the doc2vec model occurs, also took too long. In agreement with the professor, we decided to use a subset of the dataset.

This operation was quite difficult because we could not explode the *'Set_Fingerprint'* column and then take a percentage of this new dataset.

Let's give an example to make it clearer: the *'Set_Fingerprint'* of the row with *'session_id'* 1 contains these values ["Discovery", "Persistence"]. By applying the explode function, we would have 2 rows with the same session_id, but Set_Fingerprint would be

"Discovery" for the first row and "Persistence" for the second. If we now select only one of these two rows to decrease the size of the dataset, we lose information about the attack categories of that full session.

Furthermore, the train_test_split function, when applied with stratify as explained in the Section 2.5, stratifies based on the *'Set_Fingerprint'* column, which is an array of strings. Thus, when the data is divided into train and test sets, the partitioning is performed based on sextuples. If a sextuple is unique, it causes an error because it cannot apply the 'stratify' method.

We solved this by identifying the sextuples that were present only once and removing them. After that, through an iterative loop for all the sextuples, we checked the number of rows in which each sextuple appears and multiplied this number by 0.025. This result represents the number of rows that need to be selected for that particular sextuple, meaning we took 2.5% of the remaining sextuples.

Another check was performed: if the number of rows was 0 or 1, we made sure to take 2 rows to avoid the same stratification problem mentioned before. In the end, we reduced the dataset from 233,035 rows to 5,853.

From this point forward, the smaller dataset will be utilized for this study. Therefore, all the results in the later sections (Supervised and Unsupervised learning, and the Neural Network part), along with the tuning of the hyper-parameters, will be obtained based on this dataset. These results and the hyper-parameters themselves may undergo alterations when applied to the initial data set.

This reduction was made to significantly improve the execution time, making even the tuning part of the hyperparameters feasible, especially in unsupervised learning.

2.7 Standardization of the Multi-Label Categories

2.7.1 ***MultiLabelBinarizer***.

The **MultiLabelBinarizer** is a transformer designed for the classification of multi-label data. It is capable of handling cases where each sample belongs to several classes simultaneously. The MultiLabelBinarizer's purpose is to convert a collection of label sequences into a binary matrix format.

The code segment illustrates the functioning of this encoding method. Each vector of the binary matrix contains columns equal to the number of different classes, with a '1' bit assigned to the class to which the feature belongs and a '0' otherwise. In this context, columns represent one of the possible classes, while rows represent an instance.

```
from sklearn.preprocessing import MultiLabelBinarizer
mlb = MultiLabelBinarizer()
print(mlb.fit_transform(
{"genre": [["action", "drama","fantasy"],
           ["fantasy","action"],
           ["drama"],
           ["sci-fi", "drama"]]})

Output: array ([[1, 1, 1, 0],
                [1, 0, 1, 0],
                [0, 1, 0, 0],
                [0, 1, 0, 1]])
```

In the example above, the *action* category is associated with the first column, *drama* with the second, *fantasy* with the third, and *sci-fi* with the fourth. Indeed, the first output has the first three columns set to 1 and the last (*sci-fi*) to zero. In our analysis, the multi-class labels were identified in the *'Set_Fingerprint'* column of the dataset, which represents the attack categories to which the logged SSH session belongs. These labels were transformed into a binary matrix using the '*MultiLabelBinarizer'* algorithm, which will allow their use in classification algorithms at a later stage.

3 SUPERVISED LEARNING – CLASSIFICATION

Following the application of data exploration and pre-processing techniques, two supervised classification algorithms were employed to predict attacks from the SSH sessions of the dataset. Random Forest (RF) and K-Nearest Neighbor (K-NN) were the algorithms selected for this supervised learning task.

Prior to an in-depth analysis of the performance of each method, it is important to provide a thorough explanation of the underlying principles and operation of each approach.

3.1 Algorithms

Random Forest Algorithm: Random Forest (RF) is an ensemble classifier algorithm that applies the technique of "bagging" to generate multiple decision trees and classify new incoming data instances into a designated class or group [1]. The Random Forest algorithm utilizes a random selection of features or attributes from the given training set to identify the optimum split point, employing the *'Gini'* index cost function, while constructing the decision trees. The model aggregates the individual results obtained by each decision tree and counts the number of votes cast in favor of each class to generate the final prediction of the new instance.

K- Nearest Neighbor Algorithm: The k-nearest neighbor (k-NN) algorithm is a non-parametric supervised learning classification method that constructs predictions directly from the training dataset without the need for additional data. To classify an unknown data point, the algorithm identifies the set of k training samples that are the nearest neighbors to the test sample based on a distance calculation. It assigns the class with the highest number of votes out of these neighboring classes.

3.2 Splitting the dataset into Training and Test set

As a preliminary step, the data were divided into two distinct sets. The first set designated the training set, represented 70% of the data, while the second set, designated the test set, comprised only 30% of the assigned data. This division was conducted in accordance with the guidelines outlined in Section 2.5.

- **Training set**: this set comprises labeled data that the model will use to learn the underlying patterns and relationships. The data set includes both, the input features (denoted as *X_train*) and the corresponding target labels. After applying

data cleaning in the preprocessing step, the input features consisted of a set of commands identified in the 'full_session' column. These commands were transformed using the ***'TF-IDF'*** method, as described in Section 2.4.1, to convert them into meaningful features that the classification algorithms can use.
The results obtained with this method were found to be more favorable than those obtained with the Bag of Words method for the supervised algorithms in the sampled dataset. Consequently, this method was selected for the execution of all subsequent tests.
On the other hand, the labels of the training set (target labels), denoted as (y_train), were mapped using the 'Fingerprint' column of the dataset. This column contained categorical labels assigned in a list for each recorded SSH session. Since supervised learning algorithms cannot process non-numerical features or multi-label data, they must be transformed into a format compatible with supervised methods. The encoding method used for the *'fingerprint'* column was ***'MultiBinarizer'*** encoder, the details of its operation were discussed in Section 2.7.1.

- **Test set**: It is a subset of data that is set aside and not used during model training. Its primary objective is to provide an unbiased evaluation of the model's performance on previously unseen data. It is also used to modify or optimize the model for enhanced results. The size of the subset must be sufficient to produce reliable predictions.

The final size of the training set was 4,097 samples, while the test set comprised only 1,756 specimens.

3.3 Supervised Algorithms Evaluation using Default Parameters

In order to assess their performance, both Random Forest and K-Nearest Neighbor models were employed to evaluate the training and test sets. In this initial evaluation, the default parameter definitions of both models were utilized to evaluate the a priori performance of the algorithms within a predefined setting.

3.4 Random Forest

The Random Forest Algorithm comprises 20 parameters that can be modified according to the $'Sklearn'$ library definition. Of these, the parameters that proved to be most influential in the classification produced by the model can be listed as follows:

- **n_estimators**: it controls the number of trees in the forest, the default number is 100. Increasing the number of trees generally improves the model, but also increases the computational time.
- **criterion**: the criterion parameter determines the function used to measure the quality of a split. By default, it is usually $'gini'$ (Gini impurity) for classification tasks and $'mse'$ (mean square error) for regression tasks.
- **max_depth**: It represents the maximum depth of trees. The default parameter is set to $'None'$, allowing nodes to expand until they contain fewer samples than the minimum required for a split.

Upon evaluation of the model with the default parameters, the accuracy of the training and test sets was found to be 98% for each. This suggests that the model performs well on both sets. However, to ascertain the predictive power of the classifier algorithm concerning SSH attacks, additional parameters such as F1-score, recall, precision, and support must be evaluated.

The 'classification_report' method in the Sklearn library was employed to evaluate the aforementioned metrics. The following definitions apply to each parameter [2]:

- **Precision**: Percentage of correct positive predictions relative to total positive predictions.
- **Recall**: Percentage of correct positive predictions relative to total actual positives.
- **F1-Score**: A weighted harmonic mean of precision and recall. The closer to 1, the better the model.

$$\text{F1-Score} = \frac{2 \cdot (\text{Precision} \cdot \text{Recall})}{\text{Precision} + \text{Recall}}$$

- **Support**: These values represent the number of samples belonging to each class in the test data set.

It is crucial to highlight that the *'zero_division'* flag was set to 1 in order to prevent the imbalance between classes from affecting the metrics to be calculated. In this context, the *'zero_division'* flag serves to address edge cases where a division by zero occurs during the F1 score calculation. In such instances, the resulting score is set to 1, thereby preventing the algorithm from assigning a misleading score of zero when there are no true positives or false positives. This could otherwise unfairly penalize the model's performance metrics.

The outcomes of the classification report in the training set are reflected in **Table 1**.

Table 1: Classification report on training set Random Forest

	Precision	Recall	F1-Score	Support
Defense Evasion	0.990937	0.979104	0.984985	335.0
Discovery	1.000000	1.000000	1.000000	4070.0
Execution	1.000000	0.988408	0.994170	1639.0
Harmless	0.944444	0.354167	0.515152	48.0
Impact	1.000000	0.888889	0.941176	9.0
Other	1.000000	0.916667	0.956522	12.0
Persistence	0.999191	0.999460	0.999325	3705.0
micro avg	0.999283	0.993787	0.996527	9818.0
macro avg	0.990653	0.875242	0.913047	9818.0
weighted avg	0.999114	0.993787	0.995782	9818.0
samples avg	0.999268	0.994508	0.996009	9818.0

The classification report of the training set shows that the model achieved an accuracy of approximately 99% in the prediction of the attacks. The lowest accuracy was observed for the 'Harmless' attacks, which reached only 94%.

The recall values for the intents 'Defense Evasion', 'Discovery', 'Execution', and 'Persistence' yielded a result approximately above 98%, indicating that the model is competent in predicting these attacks. However, for the attacks that presented a lower number of samples in the dataset, such as 'Harmless', 'Impact', and 'Other',

the model produced a lower recall value. 'Harmless' class being the most critical one, with a recall of only 35%. This indicates that the model identified less than half of the positive cases of this class in the dataset.

Additionally, it was observed that the F-score exhibited values of approximately 98% for the majority of attack types, with the exception of the 'Impact' and 'Harmless' intention categories, where the F1-score reached 94% and 51%, respectively. 'Harmless' was once again the most challenging intention for the model to classify. The results obtained for this type of attack exhibited the lowest values. This may be attributed to the fact that in the initial data frame, the 'Harmless' attack comprised a relatively small number of samples in comparison to the other attacks. Consequently, the ratio calculated in the F1-score is influenced by the recall value obtained.

Likewise, the results of the classification report on the test set are displayed in the **Table 2**.

Table 2: Classification report on test set Random Forest

	Precision	Recall	F1-Score	Support
Defense Evasion	0.971831	0.965035	0.968421	143.0
Discovery	0.998852	0.997135	0.997993	1745.0
Execution	0.991441	0.991441	0.991441	701.0
Harmless	1.000000	0.095238	0.173913	21.0
Impact	1.000000	0.200000	0.333333	5.0
Other	1.000000	0.600000	0.750000	5.0
Persistence	0.998740	0.999369	0.999055	1586.0
micro avg	0.996649	0.990014	0.993321	4206.0
macro avg	0.994409	0.692603	0.744879	4206.0
weighted avg	0.996664	0.990014	0.991096	4206.0
samples avg	0.995520	0.989560	0.992441	4206.0

The values reported in **Table 2** for the test set indicated an accuracy higher than 97% in each instance, suggesting that the model was able to correctly identify each type of attack as belonging to the genus to which it corresponded.

Similarly, it was observed that the attacks 'Persistence', 'Execution', and 'Discovery' yielded a recall value of 99%, while 'Defense Evasion' reached 96%. Consequently, the majority of attacks were correctly identified in these categories. However, the model demonstrated poor performance in identifying 'Harmless', 'Impact', and 'Other' attacks, with respective identification rates of 9.5%, 20%, and 60%. This can be attributed to the fact that the data belonging to these categories represented a minority in the data frame, as was also observed in the Support parameter.

Consequently, the relationship between accuracy and recall yielded high F1-score values for the attacks 'Defense Evasion', 'Persistence', 'Execution', and 'Discovery'. In contrast, for the attacks with a smaller number of samples, such as 'Harmless', 'Impact', and 'Other', the respective values were 17%, 33%, and 75%.

As observed in both the training and test sets, the attacks with the lowest number of samples present in the data frame exhibited the lowest F1-score. This is due to the imbalance between the results of precision and recall. The 'Harmless' and 'Impact' attacks present the greatest challenge for the model in identifying them, due to the low number of samples compared to the others.

Furthermore, the weighted average metrics reported in both the training and test sets exhibited a relatively higher value, indicating a favorable performance of each class. This is because, in this metric, the model assigns greater weight to classes with a greater number of samples. Consequently, the performance of the classes with a larger sample size has a more pronounced impact on the overall average, thereby reducing the imbalance across different classes. As illustrated in **Table 1** and **2**, the weighted average of this metric reached 99% in both datasets, indicating that the model performed well despite the imbalance in sample size across different classes.

3.4.1 *Analysis of overfitting or underfitting using RF*

.

The model demonstrated exceptional performance on the training set, with high accuracy, recall, and F1 scores observed across all classes. This indicates a strong fit between the model and the training data. In contrast, the model exhibited satisfactory performance on the test set, although there were notable declines in performance in certain classes, particularly those with lower support such as 'Harmless', 'Impact', 'Other'. These results suggest the presence of overfitting, particularly in the classes with the lowest sampling size. This was evidenced by the pronounced decline in recall and F1 scores for these classes in the test set in comparison to the training set.

3.4.2 *Confusion Matrix for RF on the Test Set* .

The confusion matrix is a useful tool for analyzing the performance of a classification model. It compares the model's predicted labels with the true labels, providing insight into the model's accuracy. The confusion matrix comprises four boxes, each representing the number of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN) of the predictions generated by the RF algorithm. The resulted confusion matrix is presented in the **Figure 8**.

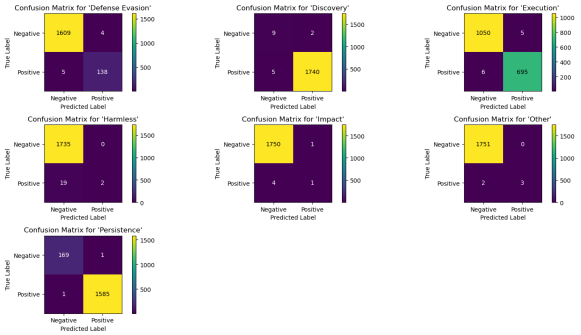

Figure 8: Confusion Matrix for Tests Set using RF algorithm.

- **Defence Evasion**: The considerable number of TN (1609) and TP (138) values indicate a high degree of accuracy in the model's ability to predict 'Defense Evasion'. The low FP (4) and FN (5) values demonstrate that the model rarely misclassifies this class.
- **Discovery**: The model demonstrated an excellent performance for the identification of the 'Discovery' class, with high TP (1740) and TN (9) values. The small values for FP (2)

and FN (5) indicated minimal errors, thereby confirming the model's reliability for this class.

- **Execution**: The substantial true negative (TN = 1050) and true positive (TP = 695) counts demonstrate the model's efficacy in accurately classifying the 'Execution' category. The minimal false positive (FP = 5) and false negative (FN = 6) occurrences further underscore the model's precision and reliability in this class, highlighting its overall accuracy and robustness.
- **Harmless**: The model exhibited a notable disparity between its True Negative (TN) and True Positive (TP) values, with TN reaching 1735 and TP remaining notably low at 2, implying challenges in accurately identifying instances within the 'Harmless' class. The elevated False Negative (FN) count, recorded at 19, underscored a substantial proportion of 'Harmless' samples being erroneously classified. On the other hand, the absence of False Positives (FP), totaling 0, indicated an admissible precision in distinguishing non-harmless samples from within this class.
- **Impact**: The elevated True Negative (TN) value, recorded at 1750, indicates that the model is adept at discerning instances outside the 'Impact' category with precision. However, the limited number of True Positives (TPs) of 1, in conjunction with the elevated number of False Negatives (FNs) of 4, indicates that there are difficulties in identifying samples belonging to the 'Impact' class.
- **Other**: The model exhibits a high true negative (TN) rate of 1,751 and a reasonable true positive (TP) count of 3, indicating satisfactory performance in delineating cases within the 'Other' class. Furthermore, the absence of false positives (FP) and the low occurrence of false negatives (FN), marked 0 and 2 respectively, underline a remarkable level of reliability in the model's predictions, although occasional misclassification errors are observed.
- **Persistence**: The high true negative (TN) and true positive (TP) values, documented at 169 and 1585, respectively, demonstrate exceptional proficiency in accurately predicting instances related to 'Persistence'. Moreover, the minimal occurrences of false positives (FP) and false negatives (FN), each quantified at 1, demonstrated the model's remarkable precision, characterised by exceedingly few errors within this class.

3.5 K-Nearest Neighbors

The same experiment was conducted again using the K-Nearest Neighbor algorithm with the default parameters. As indicated by the definition of the function provided by the Skelearn library, the K-NN algorithm has a total of 8 parameters that can be modified. The most significant contributors to classifier performance are identified below.

- **n_neighbors**: This parameter represents the number of neighbors to be used, with the default value being 5.
- **weights**: It is the weight function used in prediction. The default parameter is 'uniform', which occurs when all points within a given neighborhood are given equal weight. Conversely, the 'distance' option employs a weighting scheme whereby points are assigned a value inversely proportional to their distance from the query point. Consequently, the nearest neighbours of a query point will exhibit more influence than those situated at a greater distance.
- **algorithm**: Algorithm used to compute the nearest neighbor, the default value is 'auto'.
- **metric**: The metric to be employed for the computation of distances is that of 'Minkowski', which, when p = 2, yields the standard Euclidean distance.
- **leaf_size**: The algorithm may employ diverse data structures to accelerate the nearest neighbor search, including Ball Trees and KD Trees. These tree structures facilitate the data space partitioning, enabling more expedient neighbor searches compared to a brute-force approach. The parameter *'leaf_size'* determines the size of the leaf nodes in these tree structures. A leaf node is a terminal node in the tree that contains a limited subset of the training data points.

Once the model had stored the training and test sets and computed the respective distances for each sample, the accuracy achieved for both sets was found to be 98%. A value that suggests a good performance for the classification of the classes. However, in order to ascertain the reliability of the results obtained, the classification report was also evaluated for both cases.

Moreover, to prevent the algorithm from assigning a misleading score of zero when there were no true positives or false positives, the *'zero_division'* flag was again set to 1 in the calculation of the classification report. This was done for both, training and test sets, to ensure the fairness of the model's performance metrics.

The results of the training set evaluation are presented in the **Table 3**.

Table 3: Classification report on training set KNN

	Precision	Recall	F1-Score	Support
Defense Evasion	0.964179	0.964179	0.964179	335.0
Discovery	0.997547	0.999263	0.998404	4070.0
Execution	0.995071	0.985357	0.990190	1639.0
Harmless	0.800000	0.083333	0.150943	48.0
Impact	1.000000	0.555556	0.714286	9.0
Other	1.000000	0.750000	0.857143	12.0
Persistence	0.998382	0.999190	0.998786	3705.0
micro avg	0.996210	0.990528	0.993361	9818.0
macro avg	0.965026	0.762411	0.810562	9818.0
weighted avg	0.995350	0.990528	0.991433	9818.0
samples avg	0.996290	0.991632	0.994480	9818.0

The observed precision values for each class were relatively high for the training set, with the 'Impact' and 'Other' classes generating 100% of precision in the classification performed by the model. Similarly, high precision values were observed for 'Discovery', 'Execution', and 'Persistence' intents with 99%, these precision values are optimal for the classification task. The classes 'Defense Evasion' and 'Harmless' exhibited the lowest precision values, with 96% and 80%, respectively. However, these values are considered acceptable for classification purposes.

In addition, for the recall values, the majority of the classes exhibited a favorable outcome, with results exceeding 96%. The classes 'Defense Evasion', 'Harmless', and 'Other' exhibited the lowest performance, with 96%, 8%, and 75%, respectively. The lowest recall value observed was that of the 'Harmless' class, indicating that the model performed poorly in detecting attacks belonging to this category.

As a consequence, the results obtained for the average between the values of precision and the values of recall are reflected in the F1-score. In general, the F1 scores were high for all the attacks, except for the 'Harmless' class, which is the most difficult target for the K-NN model to detect due to the small number of samples in this class. It barely reached 15%, influenced by the recall value obtained.

Overall, the results from the training set demonstrate that the algorithm performs well. Nonetheless, the test set is more critical for evaluation because it provides an unbiased assessment of the algorithm's effectiveness, as it has not been influenced by the training process.

Table 4: Classification report on test set KNN algorithm.

	Precision	Recall	F1-Score	Support
Defense Evasion	0.945205	0.965035	0.955017	143.0
Discovery	0.996568	0.998281	0.997423	1745.0
Execution	0.991392	0.985735	0.988555	701.0
Harmless	1.000000	0.047619	0.090909	21.0
Impact	0.500000	0.200000	0.285714	5.0
Other	1.000000	0.400000	0.571429	5.0
Persistence	0.996226	0.998739	0.997481	1586.0
micro avg	0.993550	0.988825	0.991182	4206.0
macro avg	0.918484	0.656487	0.698076	4206.0
weighted avg	0.993261	0.988825	0.988647	4206.0
samples avg	0.992027	0.988468	0.993793	4206.0

Table 4 presents the classification report obtained after classifying the instances. As can be seen, the 'Defense Evasion' class demonstrated a consistent and reliable performance with balanced precision and recall, reaching an F1-score parameter of 95%, indicating an effective classification for these instances. Likewise, the results returned for the classes 'Discovery', 'Execution', and 'Persistence' exhibited a near-perfect outcome for the report. The 'Discovery' attack demonstrated near-perfect performance, as evidenced by the high support and 99% F1-score achieved, indicating that the model can almost perfectly classify these instances. The same performance was observed for the 'Persistence' classes, with high accuracy and recall values, and an F1-score of 99%. Similarly, 'Execution' obtained high precision and recall, with a slight decrease in the recall parameter, suggesting minor errors in the classification of this instance.

Conversely, for the targets 'Harmless', 'Impact', and 'Other', the observed performance declined significantly in comparison to the other attacks. The model exhibited a notable decrease in performance for the 'Harmless' category, as indicated by the low recall value, which suggests that the model failed to identify the majority of instances correctly. Conversely, the 'Impact' attack demonstrated optimal accuracy but a low recall value, which implies that the model correctly identified this class when it predicted them but did not identify a substantial number of true cases. A comparable outcome was observed for the 'Other' class, with absolute precision of 100% but a reduced recall value (40%), indicating that some true instances were not identified.

3.5.1 *Analysis of overfitting or underfitting using K-NN*

.

The K-NN model exhibited signs of overfitting, particularly in the classes with lower support, such as Harmless, Impact, and Other. This was evidenced by a significant decrease in the recall and F1 scores of these classes in the test set in comparison to the training set. The macro average value showed a decline in the test set (0.698) compared to the training set (0.811), indicating a worsened performance in the smaller classes.

For those instances where the number of samples is exceedingly limited, it is to be anticipated that the model will exhibit signs of overfitting, given that a comparison between a support value of just 10 and one of thousands of samples reveals a pronounced imbalance between the classes. A possible solution to this problem would be to evaluate classification algorithms with classes whose number of samples is small apart from the overall data set, which would allow a more nuanced understanding of their performance.

3.5.2 *Comparative Analysis between K-NN vs Random Forest Performance* .

A comparison of the results obtained with the two supervised classification models in the test set revealed that both the *K-Nearest Neighbors* (KNN) and *Random Forest* (RF) algorithms exhibited high overall performance. However, these methods demonstrated specific strengths and weaknesses in classifying specific attacks.

The *K-NN algorithm* demonstrated greater consistency across all classes, particularly in classes with lower support values, such as 'Impact' and 'Other'. It exhibited superior performance in classes with lower support, where accuracy and recall values were more balanced. In contrast, the *Random Forest algorithm* demonstrated slightly superior performance in the majority of classes and on general metrics such as micro and weighted averages. However, it showed significant difficulties with certain smaller classes, such as 'Impact' and 'Harmless'.

Additionally, the *K-NN algorithm* exhibited superior macro-average scores compared to the *Random Forest algorithm*, suggesting more consistent performance across classes. Nevertheless, both models encountered difficulty in detecting the 'Harmless' class. *Random Forest* achieved perfect accuracy, at the cost of a very low recall value for this specific target. The 'Impact' class was particularly difficult to detect for the *Random Forest algorithm*, with low precision and recall values, while *K-NN* performed better in this regard.

3.5.3 *Confusion Matrix for K-NN on the Test Set* .

A confusion matrix was constructed for each class to represent the performance of the classification algorithm in the test set. **Figure 9** depicts the results obtained from this experiment. The analysis will be conducted separately for each class.

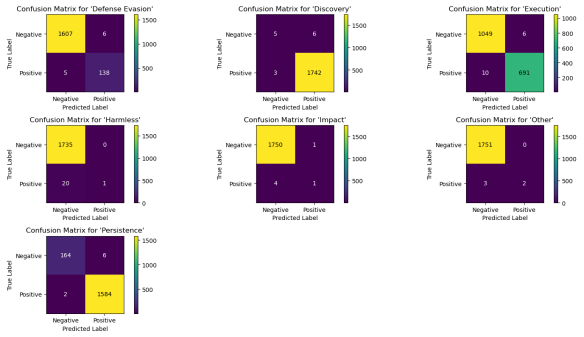

Figure 9: Confusion Matrix for Tests Set using KNN

- **Defence Evation**: A high number of True Positive (TP) values (138) and True Negative (TN) values (1607) were observed, reflecting that the model performed well for this class. However, it had a low number of false positives (FP) and false negatives (FN), with only 6 and 5, respectively, suggesting minimal classification errors. The model demonstrated high accuracy and precision for the 'Defence Evation' class.
- **Discovery**: The model demonstrated an accurate classification of instances of the 'Discovery' class with a high frequency, as evidenced by the substantial number of true positives (TP) observed. The presence of a limited number of false negatives (FN = 3) and false positives (FP = 6) indicated that the K-NN algorithm occasionally makes misclassifications. However, the model demonstrated high precision and recall for this class.
- **Execution**: The model exhibited satisfactory performance in predicting the 'Execution' class, as evidenced by a high number of true positives (TP = 691) and true negatives (TN = 1049). The low incidence of false positives (FP = 6) and false negatives (FN = 10) indicated that while the model generally performed well, it occasionally made minor errors.
- **Harmless**: The model demonstrated limitations in accurately identifying instances belonging to the 'Harmless' class. This was due to the false negative score (FN = 20), which resulted in a significant number of cases in the 'Harmless' class being incorrectly predicted by the model.
- **Impact**: In this instance, the model tended to predict the 'Impact' class with a high degree of accuracy, with a TP value of 1. Conversely, the true negatives yielded a relatively high score of TN = 1750, suggesting that when the 'Impact' class is not present, the model would perform correctly.
- **Other**: The results demonstrated a similar behavior to the 'Impact' class, whereby the model encountered difficulty in identifying the samples belonging to the 'Other' class. A high true negative value was observed (TN = 1751), indicating that the model correctly identified instances belonging to the "Other" class that were not predicted as such. The low value for true positives (TP = 2) and the high value for false negatives (FN = 3) reflected that the model requires improvement in accurately identifying the 'Other' class.
- **Persistence**: The model demonstrated excellent performance for this attack, as evidenced by the high number of true positives (TP = 1584) and true negatives (TN = 164). The low number of false positives (FP = 6) and false negatives (FN = 2) implied that the model rarely misidentified samples belonging to this class.

3.6 Hyper-parameter Tuning for Supervised Learning Algorithms

Each supervised classification model is associated with a set of hyper-parameters, which represent a specific type of variable. In order to ascertain which hyper-parameters will yield the optimal outcome for the model under evaluation, it is necessary to conduct a series of experiments in which the selected hyper-parameters are varied and then run through the model. This process is referred to as hyper-parameter tuning and can be most effectively accomplished through the use of the *'GridSearch'* technique.

GridSearch is an optimization technique employed to identify the optimal combination of hyper-parameters for a machine learning algorithm. In essence, the model is trained sequentially with different sets of hyper-parameters, and the results obtained between each combination are displayed. The optimal value is the one that generates the best result.

3.6.1 *Hyper-parameter Tuning for Random Forest* .

As previously stated in section 3.4, the *Random Forest* model comprises 20 parameters that can be modified. Of these, the most significant in terms of classifier performance was the number of trees to be created (***N_stimators***) and the maximum tree depth (***max_depth***). A dictionary was thus created with the possible parameters to be tuned, namely ***N_stimators*** = [10, 30, 70, 100, 120, 150], while ***max_depth*** = [10, 50, 70, 100, None], *'None'* meaning that nodes are expanded until all leaves are pure or contain less than the minimum samples required to split.

These parameters have been defined in the ***GridSearch*** hyper-tuning to find the best combination of parameters. In addition, the ***'GridSearch'*** technique has been specified with the following parameters.

- **Estimator**: Represent the model to be optimized. In this case, it was the *RandomForestClassifier* with a fixed random state for reproducibility.
- **param_grid**: The dictionary containing the hyper-parameters and their values to be searched by *GridSearchCV* (dictionary).
- **score**: The metric used to evaluate the performance of the model. In this case, it was the weighted F1-score, which balances precision and recall and considers class imbalance.
- **cv**: The number of cross-validation folds. Here 7-fold cross-validation was used, which means that the data was splitted into 7 parts and the model was trained and validated 7 times, each time using a different fold as the validation set. These values were chosen based on the number of different classes in the dataset.
- **Verbose**: Controls the verbosity of the output. A value of 1 means that progress messages are printed.

Once the *GridSearch* model was defined, the *'fit'* method was applied to train the model. The *'fit'* method performed an exhaustive search over the specified parameter grid, evaluating each hyper-parameter combination by cross-validation. The best combination generated was selected using the '.best_params' method, which

allows for the straightforward identification of the optimal parameter combination identified by the *GridSearch*, based on the highest weighted F1-score.

The results demonstrated that the combination that yielded the highest weighted F1-score across all the folds was ***'max_depth': 10, 'n_estimators': 30*** as can be seen in **Figure 10**. This combination achieved a very good balance between precision and recall values, with a 99.2% score.

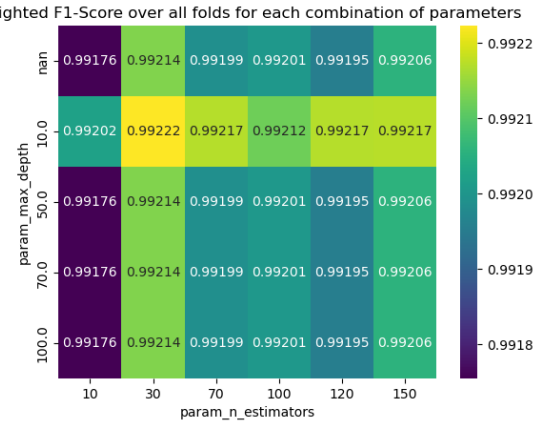

Figure 10: Weighted F1-Score over all folds for each combination of parameters

Once the optimal parameters for the Random Forest model had been identified, the performance of the classifier was evaluated using the previously tuned hyper-parameters. Consequently, only three parameters were defined for the ***'RandomForestClassifier'***: the number of estimators, ***'n_estimator'***, was set to 30; the maximum depth, ***'max_depth'***, was equal to 10; and the ***'random_state'*** was set to 42 to guarantee the reproducibility of the results.

The results demonstrated that the model achieved a high level of accuracy (0.98) and a weighted F1-score (0.99) on the test set, indicating excellent performance. These metrics suggest that the model is highly effective in distinguishing between different classes in the dataset. Nevertheless, to ascertain the classifier's efficacy, Table 5 presents the classification report's results.

Table 5: Classification Report with hyper-parameter tuned for RF.

	Precision	Recall	F1-Score	Support
Defense Evasion	0.971831	0.965035	0.968421	143.0
Discovery	0.998854	0.998854	0.998854	1745.0
Execution	0.991441	0.991441	0.991441	701.0
Harmless	0.666667	0.095238	0.166667	21.0
Impact	1.000000	0.200000	0.333333	5.0
Other	1.000000	0.600000	0.750000	5.0
Persistence	0.999369	0.999369	0.999369	1586.0
micro avg	0.996652	0.990728	0.993681	4206.0
macro avg	0.946880	0.692848	0.744012	4206.0
weighted avg	0.995238	0.990728	0.991536	4206.0
samples avg	0.995729	0.990983	0.991821	4206.0

The results demonstrate that 'Defence Avoidance', 'Discovery', 'Execution', and 'Persistence' attacks are the classes that achieved high accuracy, recall, and F1 scores. This indicates that the model performs well in predicting these classes accurately and consistently.

Classes such as 'Harmless' and 'Impact' exhibited lower recall and F1 scores of 9.5% and 16.6% respectively. This suggests that the model has difficulty in identifying cases belonging to these classes, due to class imbalance. Similarly, the 'Impact' and 'Other' classes demonstrated high accuracy but lower recall values, particularly for the 'Impact' class. Consequently, although the model can correctly predict instances of these classes, it frequently fails to identify cases belonging to these types of attacks.

The macro average reflects a lower overall performance compared to the weighted average, indicating disparities in prediction precision across different classes. The weighted average accounts for class imbalance, resulting in higher performance metrics, particularly for more prevalent classes. In our case, this parameter represents an important metric to be compared concerning the observed during the experiment carried out without any tuning in section 3.4.

3.6.2 *Comparative Analysis between Results Obtained with and without Hyper-Tuning Parameters* .

As can be seen in **Table 6** and **Table 7**, there was an improvement in the weighted parameters measured for the adjusted case. The F1-score demonstrated that the tuned model exhibited satisfactory performance across the imbalance of classes.

Table 6: General metrics obtained without any tuning.

	Precision	Recall	F1-Score	Support
micro avg	0.996649	0.990014	0.993321	4206.0
macro avg	0.994409	0.692603	0.744879	4206.0
weighted avg	0.996664	0.990014	0.991096	4206.0
samples avg	0.995520	0.989560	0.992441	4206.0

Table 7: General metrics obtained for the tuned hyper-parameters.

	Precision	Recall	F1-Score	Support
micro avg	0.996652	0.990728	0.993681	4206.0
macro avg	0.946880	0.692848	0.744012	4206.0
weighted avg	0.995238	0.990728	0.991536	4206.0
samples avg	0.995729	0.990983	0.991821	4206.0

A comparison of the metrics obtained across all the reports reveals a slight improvement in precision, recall, and F1-score for several classes following hyperparameter tuning. This effect is particularly pronounced for the 'Other' and 'Impact' classes, which represent the target with a relatively small number of examples in the dataset. Moreover, both the micro and macro-averaged metrics show a similar trend. Overall, hyperparameter tuning appears to have a positive impact on the classifier's performance.

3.6.3 *Hyper-parameter Tuning for K-NN* .

For the K-NN case, the parameters that have demonstrated to have the greatest impact on the performance of the classifier are listed in the 3.5 section. They represent the hyperparameters that will be tuned at this stage. For each variable, a set of options has

been defined that will be used by the ***GridSearch*** technique to generate all possible combinations. The following options were defined: **'n_neighbors'**: [2, 3, 5], 'leaf_size': [10, 30, 50, 80], **'weights'**: ['uniform', 'distance'], **'metric'**: ['euclidean', 'manhattan', 'chebyshev', 'minkowski'].

The number of neighbors (***n_neighbors***) requires K values to vote the K number of neighbors around the new test data instance and classify it to the maximum voted class label. The number chosen was neither too low to avoid generating overfitting nor too high to avoid overgeneralizing the model predictions, which could lead to underfitting.

In the *Scikit-learn*, framework, the default value for the parameter *'leaf_size* is 30, which has proven effective in many data sets. However, it may not be optimal for all cases. To address this, different values have been provided to ***GridSearch*** to identify the parameter that best suits the current context. Similarly, for the parameter *'metric'*, there are four distinct methods for calculating the distance between the test data and its neighbors. This parameter determines how the 'closeness' or 'similarity' between the data points is measured. In this case, four options were provided ('euclidean', 'manhattan', 'chebyshev', 'minkowski') to identify the most suitable one.

Accordingly, the parameters that were established for the "GridSearch" technique are presented below:

- **Estimator**: Represent the algorithm to be evaluated, *K-Nearest Neighbor*.
- **param_grid**: The dictionary containing the hyper-parameters and their values to be searched by *GridSearchCV*.
- **score**: The metric used to evaluate the performance of the model. As in the previous case, the score evaluated was the weighted F1-score, which balances precision and recall and takes into account class imbalance. In addition, the 'divison_zero' flag was set to deal with cases where there may be a division by zero.
- **cv**: The number of cross-validation folds. Here 7-fold cross-validation was used, which means that the data was split into 7 parts and the model was trained and validated 7 times, each time using a different fold as the validation set. These values were chosen based on the number of different classes in the dataset.
- **Verbose**: Controls the verbosity of the output. A value of 1 means that progress messages are printed.

Once the parameters for ***GridSearch*** had been defined, the model was trained using the *'fit'* method. The results yielded a weighted F1-score of 99% for the optimal combination of parameters identified by ***GridSearch***, specifically: ***{'leaf_size': 10, 'metric': 'euclidean', 'n_neighbors': 5, 'weights': 'distance'}*** as illustrated in **Figure 11**. The columns represent all possible combinations performed by the ***GridSearch***, while the rows represent the weighted F1-scores achieved for each combination.

Upon identification of the optimal combination of parameters, a posteriori evaluation of the K-NN model was conducted, employing the previously identified hyper-parameters, to ascertain whether there was an enhancement in the *KNeighborsClassifier's* performance.

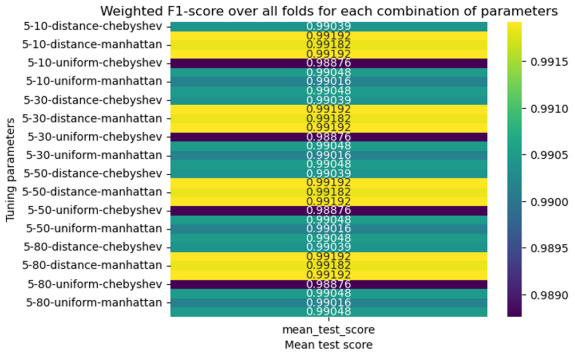

Figure 11: Weighted F1-Score over all folds for each combination of parameters

The data set was trained with the tuned classifier, resulting in a high level of accuracy in the predictions with 0.98, while the weighted ratio between the precision and recall values reached an optimal balance of 0.99 in the F1-score. These metrics offer insight into the performance of the model. However, additional analysis was conducted by calculating the classification report in order to determine the identification of the classes. **Table 8** illustrates the outcomes obtained.

Table 8: Classification Report with hyper-parameter tuned for KNN.

	Precision	Recall	F1-Score	Support
Defense Evasion	0.958333	0.965035	0.961672	143.0
Discovery	0.999425	0.995989	0.997704	1745.0
Execution	0.995690	0.988588	0.992126	701.0
Harmless	0.500000	0.095238	0.160000	21.0
Impact	0.500000	0.200000	0.285714	5.0
Other	1.000000	0.600000	0.750000	5.0
Persistence	0.998739	0.998739	0.998739	1586.0
micro avg	0.996406	0.988825	0.992601	4206.0
macro avg	0.850312	0.691941	0.735136	4206.0
weighted avg	0.994060	0.988825	0.990616	4206.0
samples avg	0.994970	0.988136	0.992084	4206.0

As illustrated in the **Table 8**, the 'Discovery', 'Persistence' and 'Execution' classes exhibited near-perfect accuracy and recall values, accompanied by exceptionally high F1 scores. This evidence suggests that the model was capable of correctly identifying these classes. The "Defense Evasion" attacks achieved an accuracy of 95.8% and a recall value of 96.5%, resulting in an F1 score of 96.1%. This illustrates the reliability of the classification of this class. The instances classified as 'Harmless', 'Impact', and 'Other' again demonstrated the lowest observed scores. The 'Harmless' class achieved an accuracy of 50% and a deficient recall value of 9.52%, resulting in an F1 score of 16%. This reflects that the model had difficulty in correctly identifying instances classified as 'Harmless'. The same behavior was observed for the 'Impact' class. Similarly, the 'Other' class achieved 100% accuracy but a low recall value, resulting in an F1-score of 75%. This suggests that, although the predictions are correct when made, the model often fails to identify instances of this class.

Overall, the weighted and average metrics demonstrated superior performance, as evidenced by their higher scores. Nonetheless, a notably deficient performance was observed in the minority classes. This deficiency can be attributed to their low representation in the dataset or the intrinsic challenges associated with distinguishing these classes.

3.6.4 *Comparative analysis between results obtained with and without Hyper-Tuning Parameters* .

A comparison of the results obtained in the two experiments, one with hyperparameters and one without, revealed a slight improvement in the classification reports. The 'Defense Evasion', 'Discovery', and 'Harmless' attacks exhibited a slight increase in F1 score values, indicating an improvement in the balance between accuracy and recall parameters.

Table 9: General metrics obtained without any tuning.

	Precision	Recall	F1-Score	Support
micro avg	0.993550	0.988825	0.991182	4206.0
macro avg	0.918484	0.656487	0.698076	4206.0
weighted avg	0.993261	0.988825	0.988647	4206.0
samples avg	0.992027	0.988468	0.993793	4206.0

Table 10: General metrics obtained for the tuned hyper-parameters.

	Precision	Recall	F1-Score	Support
micro avg	0.996406	0.988825	0.992601	4206.0
macro avg	0.850312	0.691941	0.735136	4206.0
weighted avg	0.994060	0.988825	0.990616	4206.0
samples avg	0.994970	0.988136	0.992084	4206.0

Furthermore, the overall metrics presented in each case, as shown in **Table 9** and **Table 10**, exhibited slight differences between them. The results indicated an enhancement by the experiment with the adjusted parameters. Given the imbalance in the number of samples belonging to each class in the data set used in the present work, the most important metric for assessment was the Weighted Average of the F1-score. This demonstrated a higher score for the model evaluated with the hyperparameters, increasing from 0.989 (untuned) to 0.991 (tuned). Similarly, the macro average F1-score also demonstrated an improvement.

4 UNSUPERVISED LEARNING – CLUSTERING

In the previous section, a model was trained using various supervised algorithms with the goal of making it as accurate as possible in the classification of SSH attacks.

Actually, in machine learning there are other possible approaches to enable a machine to "learn". In this section, will be explored the clustering approach, but what is clustering? Why is it important? What are the possible algorithms that can be used?

Clustering aims to divide a set of data points into clusters, where a cluster consists of several discrete items that are close to each other. Essentially, for a given set of data points, which in our study represents attack sessions, the objective is to group them based on some similarity that inherently connects them.

Clustering algorithms fall under the broader category of techniques called **unsupervised learning**. This type of machine learning involves an algorithm identifying patterns in an unlabeled dataset. Unlike supervised learning, where the dataset contains predefined target labels that guide the training process, unsupervised learning requires the algorithm to find patterns and similarities based on other criteria, such as geometric considerations. Consequently, the loss is no longer computed as a straightforward difference between obtained and actual labels, requiring alternative methods to evaluate performance.

The goal of a clustering algorithm is to assign a set of data points to specific clusters. After executing the algorithm, the dataset is expected to be divided into distinct clusters and the method of assigning data points to clusters can be varied. There are primarily two classes of clustering algorithms:

- **Hard clustering**
  An hard clustering algorithm takes a set of datapoints:
  $$D = \{(\mathbf{x}^{(1)} y^{(1)}) \dots ((\mathbf{x}^m, y^m)\}$$
  and assign to each datapoint $\mathbf{x}$ a specific cluster, so that finally will return a vector of clusters where each value will be:
  $$\mathbf{y}^i \in \{1 \dots k\}$$
  where k is the number of clusters.

  K-Means In this section will be used K-Means as the hard clustering algorithm. In summary, this algorithm assign clusters to datapoints following these steps:
  (1) Based on the number of cluster K and on the initialization method, an initial set of cluster means is randomly chosen.
  (2) Each data point is assigned to the cluster associated with the nearest mean, based on the distance from the cluster mean.
  (3) The cluster means are updated to minimize the clustering error.
  Now the algorithm returns to the previous step, cyclically updating the cluster means and assignments until an acceptable clustering error is achieved.
- **Soft clustering** A soft clustering algorithm takes in input a set of datapoints:
  $$D = \{(\mathbf{x}^{(1)} y^{(1)}) \dots ((\mathbf{x}^m, y^m)\}$$
  as the previous one, but now each datapoint $\mathbf{x}^{(i)}$ is assigned to a set of clusters, with a vector like:
  $$\mathbf{y}^i = \{y_1^{(i)} \dots y_k^{(i)}\}$$
  where k is the number of clusters. Thus each datapoint will have an associated vector that contains all the k clusters, and each value of $\mathbf{y}^{(i)}$ represent in some way how well the datapoint $\mathbf{x}^{(i)}$ "fits" into a specific cluster. This value is called *degree of belonging*. The degree of belonging can be interpreted as a probability. Each $y_c^{(i)}$ essentially indicates how much the i-th datapoint belongs to a cluster c.

Gaussian Mixure Model-GMM
Regarding the soft clustering algorithm used in this section, in Gaussian Mixture Models (GMM), each cluster will be

a Gaussian distribution in the space of the datapoints. For example, if a data point is near the edge of the Gaussian bell curve, it will have a low degree of belonging to the cluster represented by that Gaussian. Conversely, if it is close to the center, it will have a higher degree of belonging to the cluster represented by that Gaussian distribution.
Each Gaussian (and thus each cluster) will have its own cluster mean and standard deviation, covariance matrix, and effective size. The steps of the algorithm are similar to those of K-Means, but in this case, it is not just the cluster mean that is randomly selected and updated, also the covariance matrix and effective size of the clusters are updated. These parameters are adjusted in each iteration to minimize cluster spread. Data points are then reassigned, and the algorithm continues this process until convergence, in the end providing the final parameters for each cluster.

EVALUATING AND COMPARING CLUSTERS

In clustering the evaluation and comparison steps are essential. There are basically 2 main approaches to compare different clustering results:

Internal Indexes

The goodness of the clustering structure is obtained just using the intrinsic internal information of the clustering (like the compactness and the separation), so without external information.

In this chapter will be used different internal indexes:

- **Silhouette metric** Measures the consistency within the clusters of data based on:
  - How similar a data point is to its own cluster, so the cohesion of the cluster. This quantity is practically the mean distance from the other datapoints of cluster.
  - How similar a data point is compared to the other clusters, a property called so the *separation*. This parameter is the mean distance from the datapoints of the next nearest clusters.

  Calling respectively a and b the previous parameters, the Silhouette is computed as:

$$s = \frac{b - a}{max(a, b)}$$

  This is computed for each sample of the cluster, and then is taken an average. The silhouette ranges form -1 to +1, and a high value indicates that the sample is well matched to its own cluster (and poorly matched to the neighboring), while a *value around zero* indicates overlapping clusters.
  In general if the average silhouette is high (0.5 is already a good value), the clustering configuration is appropriate.
- **Total-Log-Likelihood Score** It's a statistical measure used to assess the fit of a probabilistic model to a given dataset. It quantifies the probability of observing the data under the specified model parameters. In particular the total-log-likelihood score is the sum of the log-likelihoods of individual data points in a dataset.
  In a GMM, this measure quantifies the likelihood of the observed data given the model parameters (means, covariances, and mixture weights of the Gaussian components). It sums the log-likelihoods of the data points, indicating the probability of the data under the model. As previously mentioned, GMM computes the probability of each data point belonging to the clusters. This probability is higher when the data point is close to the mean (the center of the bell curve) and lower when it is near the edge. The total log-likelihood aggregates these probabilities for all the data points assigned to their respective clusters.
  This leads to the following interpretation:
  - A**higher**total-log-likelihood score indicates that the model assigns higher probabilities to the observed data points, implying a better fit.
  - A **lower** total-log-likelihood score suggests that the model does not fit the data well.

  Note that this is an internal index because it evaluates the fit of the model based solely on the data and the model parameters, without any external labels.
- **Inertia** Inertia is a measure of how well the clusters are formed by the K-means algorithm. It is defined as the sum of squared distances between each data point and the centroid of the cluster to which it belongs, so it's similar to the clustering error and follow the same interpretation rules, lower values of inertia indicate tighter clusters.

External Indexes

The goodness of the clustering structure is obtained by mean of some external reference, like the results of other clusters.

Rand Index(RI) An example of external index, it measures the similarity of 2 assignments. Basically given 2 clustering results, with the RI are computed:

- The number of pairs of datapoints that are in the same cluster in both the clustering results.
- The number of pairs of datapoints that are in different clusters in both the clustering results.

Defining respectively as a and b the previous paremeters, the RI is computed as:

$$RI = \frac{a + b}{\binom{m}{2}}$$

If the RI is high, there are a lot of similar clusters, if it's 0, the clusters are different.

4.1 TUNING THE NUMBER OF CLUSTERS

Building upon the dataset introduced in the preceding sections, this chapter delves into the application of two clustering algorithms:**K-means** and **GMM**. An essential hyperparameter to consider is the number of clusters into which the datapoints will be partitioned, as the clustering error heavily relies on this factor. Ideally, the clustering error is minimized when the number of clusters matches the number of data points, resulting in a situation where clustering is not even performed, but where the error is zero.

The objective now is determining the optimal number of clusters that yields the lowest clustering error. Initially, the best number of clusters will be identified while keeping all other hyperparameters at their default settings. Subsequently in the next section,

fine-tuning of these parameters will be attempted to achieve even superior results.

It's worth noting that in all experiments of this chapter, a random_state of 42 has been utilized. The **random_state** parameter influences the random aspects of the algorithms, and it's a best practice to set it randomly as well, as adjusting the random state to obtain specific results is considered cheating.

4.1.1 ***K-Means.*** The tuning of the number of clusters involves iterating through a range of values, typically set between 3 and 17, which seems to encompass a wide enough range of possibilities. For each iteration of the algorithm within this range, two metrics are computed: silhouette and inertia. Then, for each iteration, the results are stored in a list of metrics used to plot the result.

After few seconds of computation the following charts are generated:

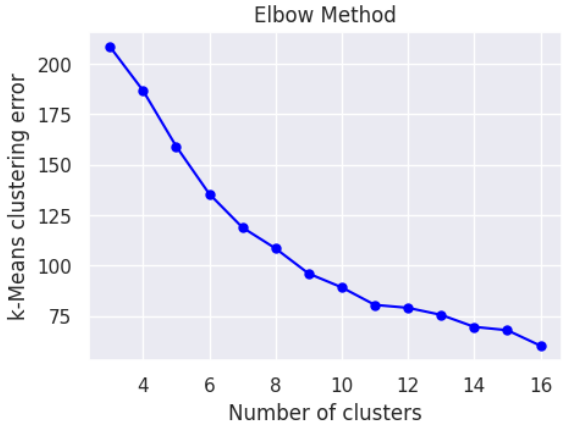

Figure 12: K-means Clusterig Error

Figure 13: K-means Silhouette

Applying the elbow method to the clustering error chart and considering the relative silhouette score, a promising value for the number of clusters appears to be 10.

This value represents a balance between minimizing the inertia and maximizing the silhouette score, hoping in well-separated clusters.

4.1.2 ***Gaussian Mixure Model-GMM.*** Similar to the previous algorithm, the Gaussian Mixture Model (GMM) also employs a for loop to execute the algorithm with varying numbers of clusters, typically ranging from 3 to 17.

For each iteration of the algorithm, two metrics are calculated: silhouette and GMM total-log-likelihood. These metrics are then collected and stored in a list for subsequent visualization.

The resulting metrics are as follows:

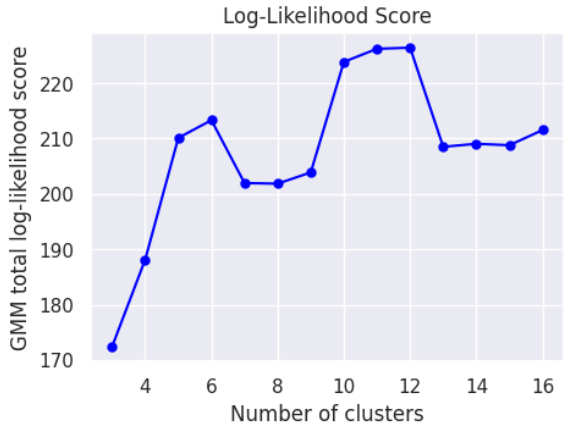

Figure 14: GMM Total Log-Likelihood score

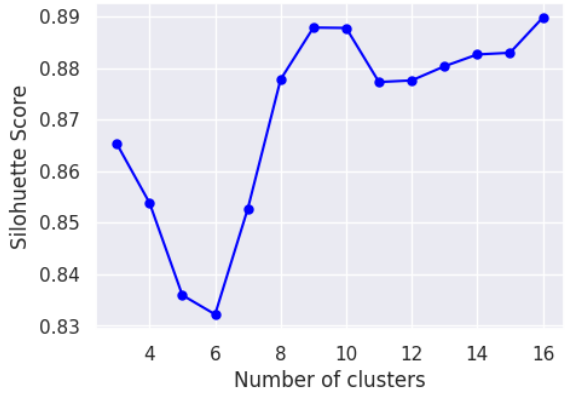

Figure 15: GMM Silhouette score

Considering that both the log-likelihood and silhouette metrics indicate better clustering with higher values, a suitable number of clusters in this case could indeed be 10. This choice represents a favorable balance between the two measurements.

4.2 TUNING OF THE HYPERPARAMETERS

When trying to learn an effective function for modeling data, certain parameters must be specified before the learning process commences. These parameters, known as **hyperparameters**, play a crucial role in machine learning. They serve two primary functions: defining the characteristics of the model and governing the optimization techniques used for clustering.

The selection of appropriate hyperparameters is crucial for the success of the model. and often entails manual tuning. Typically,

hyperparameters are initially set to default values, and this section will present the tuning process for them.

In this chapter, the hyperparameters of both K-means and GMM will be tuned, using the number of clusters discovered previously. Numerous hyperparameters can be tuned for each algorithm, so how to find the best combination of them for the actual dataset?

There are various solutions, including manual exploration using nested 'for' loops. However, the fastest method in terms of code and computation time is **GridSearchCV**, a scikit-learn tool specifically designed for hyperparameter tuning. It performs an *exhaustive search* over a specified grid of hyperparameter values to find the best combination that maximizes the model's performance, evaluated using cross-validation.

4.2.1 ***K-Means***. As seen previously a good number of clusters for K-Means is 10. Now the tuning will be applied to a K-Means with this number of clusters, focusing on the following hyperparameters:

- **Initialization method** (*init*) and **iterations** (*n_init*): in K-Means is crucial to choose in a correct way the initialization of the centroids. A bad initialization can be for instance to set them far from the clusters, leading the algorithm to reach just a local optimum value, instead of the optimal clustering. In sk_learn there is the *n_init* hyperparameter that determines the number of times the algorithm is executed with different centroid seeds. Then the best one will be taken, based on the inertia. By default, n_init = 10.
  However, once chosen the number of times that the algorithm will be executed with different centroids seed, how to choose this seeds?
  This depends on the method used for the initialization of the centroids, that can be done randomly, passing the positions, or using some built-in method. This hyperparameter can be set by means of the *init* field of KMeans().
- **Maximum number of iterations** (*max_iter*): since Kmeans is an iterative algorithm, the centroids are recomputed in order to reduce the clustering error and data points are reassigned in each step. In other terms, the maximum number of iterations serves as a threshold at which the algorithm should stop. By default, this hyperparameter is set to 300 and determining the best value for our case requires further investigation.

Now, passing to the GridSearchCV() this parameter grid:

Listing 1: K-Means Tuning - Grid Parameters

```
param_grid = {
    'init_params': ['k-means++', 'random'],
    'n_init':  list(range(2, 21, 2)),
    'max_iter':  list(range(50, 501, 50))
}
```

the obtained best values of the hyperparameters that will be used for the subsequent experiments are:

- Number of clusters: 10
- Initialization method: k-means++
- Number of initialization: 4
- Maximum number of iterations of the algorithm: 50

How were these results achieved? It was possible to do this by using the *best_params_* attribute of the grid search, which contains all the tuned values previously inserted into the *param_grid* variable. To be clear, below is a piece of code:

Listing 2: K-Means with tuned hyperparameters

```
kmeans = KMeans(n_clusters=10, random_state=42)
grid_search_kmeans = GridSearchCV(kmeans, param_grid
    = param_grid_kmeans, cv=5)
grid_search_kmeans.fit(X)
best_params_kmenas = grid_search_kmeans.best_params_
kmeans_tuned = KMeans(n_clusters=10, init=
    best_params_kmenas['init'], n_init=
    best_params_kmenas['n_init'], max_iter=
    best_params_kmenas['max_iter'], random_state=42)
```

the results obtained with *kmeans_tuned* are:

- Silhouette: 0.882
- Inertia: 84.865

Thus, the silhouette score remains essentially the same, while inertia decreased from 89.701 to 84.865, indicating that the data points are clustered more effectively and are closer to their respective means. Although this might not seem like a significant improvement, the t-SNE representation in the next section reveals that the clusters are more finely grained, and the closer data points are grouped more accurately.

4.2.2 ***Gaussian Mixure Model-GMM***. As determined in the previous section, the optimal number of clusters is 10. Now, which hyperparameters can be adjusted in GMM?

- **Covariance type** (*covariance_type*): As discussed in the introduction, in GMM each cluster is a component, a Gaussian distribution with its own variance, mean, etc. This hyperparameter sets the type of covariance matrix to be used. The following options will be tested:
  - *'full'*: Each component has its own general covariance matrix.
  - *'tied'*: All components share the same general covariance matrix.
  - *'diag'*: Each component has its own diagonal covariance matrix (no correlation between features).
  - *'spherical'*: Each component has its own single variance (spherical shape) covariance matrix.

  The choice of covariance type will influence the shape of the clusters and, consequently, the clustering results.
- **Method of initialization**(*init_params*): This determines the initialization method for the parameters of the Gaussian distributions. There are basically two options, initialize parameters using the kmeans or randomly.
- **Number of iteration of the algorithm** (*max_iter*): This sets the maximum number of iterations for the GMM algorithm, determining how many iterations the algorithm will perform before stopping. The default value is 100. The default value is 100.
- **Convergence tolerance** (*tol*): This parameter controls the convergence tolerance of the Expectation-Maximization (EM) algorithm used to fit the GMM. It specifies the threshold for

the change in the log-likelihood of the data between successive iterations, below which the algorithm stops iterating.

Now, passing to the GridSearchCV() this parameter grid:

Listing 3: K-Means Tuning - Grid Parameters

```
param_grid = {
    'init_params': ['kmeans'],
    'covariance_type': ['full', 'spherical'],
    'tol': [1e-3, 1e-4, 1e-5],
    'max_iter': list(range(50, 300, 50))
}
```

the obtained best values of the hyperparameters that will be used for the subsequent experiments are:

- Number of clusters: 10
- Initialization method: kmeans
- Maximum number of iterations of the algorithm: 50
- Covariance type: full
- Convergence tolerance: 0.001

These results are obtained in the same way as KMeans; however, in this case, hyperparameter tuning has been performed using the silhouette score. The relevant part of the code to explain this new approach is as follows:

```
def silhouette_scorer(gmm, X):
    labels = gmm.fit_predict(X)
    return silhouette_score(X, labels)
...
grid_search_gmm = GridSearchCV(gmm, param_grid_gmm,
    cv=5, scoring = silhouette_scorer)
grid_search_gmm.fit(X)
best_params_gmm = grid_search_gmm.best_params_
gmm_tuned = GaussianMixture(n_components = 10,
    random_state=42, init_params=best_params_gmm['
    init_params'], covariance_type=best_params_gmm['
    covariance_type'], max_iter=best_params_gmm['
    max_iter'], tol=best_params_gmm['tol'])
```

The obtained results are:

- Silhouette: 0.887
- Log-l: 223.853

The results without tuning were essentially the same, with only minor differences in the order of 10^{-4}. Therefore, there is no significant improvement in the metrics compared to the performance obtained with the default hyperparameters. This can also be observed graphically by applying t-SNE to both cases, where the results without tuning were worse.

For this reason in the subsequent experiments will be used the default values of the hyperparameters for the GMM algorithm.

4.3 VISUALIZATION OF THE OBTAINED CLUSTERS

***Notation**: from this chapter onwards, since there are frequent references to the various clusters, the notation CX will be used to indicate cluster X. Therefore, cluster 9 becomes C9.*

Up to this point, the discussion has revolved around finding the optimal combination of the number of clusters and hyperparameters to achieve the best performance of the clustering model. Now, with all the necessary results at hand, how can the clusters be represented graphically?

There are various techniques to accomplish this goal, but for this study case has been chosen to use **t-SNE** (***t**-Distributed **S**tochastic **N**eighbor **E**mbedding*), a very popular dimensionality reduction technique particularly used for the visualization of high-dimensional data. Specifically t-SNE is used for visualizing high-dimensional data in 2 or 3 dimensions, facilitating the interpretation of the structure and relationships within the data. Unlike linear reduction techniques like PCA, t-SNE captures non-linear relationships in the data, preserving local structures.

How can t-SNE be applied to the results of clustering?

Initially, the t-SNE() function is applied to the dataset, specifically reducing it to exactly 2 components. Then, considering the clustering results obtained in the previous section, which essentially comprise a list of clusters corresponding to each data point, a new dataframe is constructed, consisting in 2 columns that represent the 2 components of t-SNE and a third column containing the clustering results, initially from K-means and subsequently from GMM. Ultimately, the dataframe is visualized as a scatterplot using seaborn.

4.3.1 ***K-Means***. With K-means, using the already calculated *kmeans_tuned*, the following steps were used to create the dataframe:

Listing 4: K-Means Tuning - Grid Parameters

```
labels_kmeans_tuned = kmeans_tuned.labels_
df_tsne_kmeans = pd.DataFrame(df_tsne)
df_tsne_kmeans["cluster"] = labels_kmeans_tuned
df_tsne_kmeans.columns = ["x1", "x2", "cluster"]
```

and the final plot is:

Figure 16: TSNE of K-Means with 10 clusters

4.3.2 ***Gaussian Mixure Model-GMM***. Proceeding in the same way described for K-means, the obtained result is:

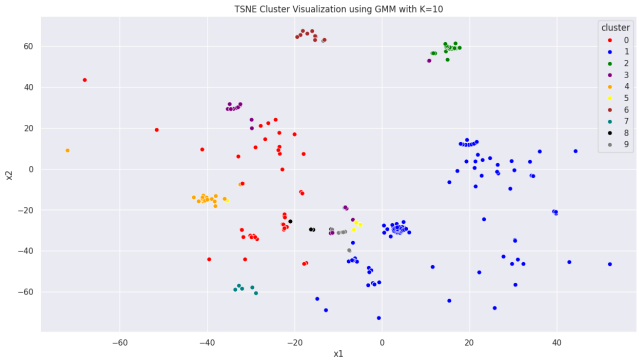

Figure 17: TSNE of GMM with 10 clusters

The clustering results appear promising, at least from a graphical standpoint. Most of the main clusters of closely located data points have been effectively grouped together by both algorithms.

Specifically, in reference to the K-means chart, clusters 2, 6, 8, and 9 are identical. However, differences are observed in the remaining clusters, which will be further analyzed in depth later but are already noticeable from the charts. For instance:

- In K-means, cluster 1 has been separated from cluster 3. This distinction is evident as there is a dense concentration of points separated from others, suggesting that it might be logical to group them into different clusters. Conversely, in GMM, these two clusters are combined.
- GMM identifies cluster 3, which in K-means is simply part of cluster 0.
- the datapoints belonging to clusters 5, 7, and 4 in K-means are split into four clusters (C3, C5, C8, and C9) in GMM, indicating some minor differences.

4.4 CLUSTERS ANALYSIS

In this section, the clustering results for both K-means and GMM will be analyzed. Specifically, the focus is on:

- the **correlation** between the *intents* assigned by the MITRE ATT&CK Tactic to each session and the *clusters* in which the sessions are divided: Are the clusters reflecting the assigned labels? Do they represent specific classes of attacks?
  To compare the intents of the sessions with the clusters, a heatmap will be drawn. This **heatmap** will visualize, for all the data points, which of the 7 labels has been assigned by MITRE and in which cluster they belong. Thus, it will be clear for each cluster which type of sessions it contains, at least according to the MITRE results. Essentially, this heatmap represents the distribution of intents over the clusters.
  To achieve this, the dataframe with all the features will be utilized, and two new columns will be added: the cluster in which the data point has been classified and its original label. Finally, Seaborn will be used to plot the heatmap.
- Examination of the **most frequent words** in each cluster: Since the clusters represent branches of shell sessions of attack, identifying the most frequent commands, attributes, options, directories, filenames, and other information contained in a session can provide insights into the class of attacks the cluster contains.

  To perform and visualize the most frequent words in each session, a **wordcloud** will be utilized. A wordcloud is a visual representation of word frequency, with the size of each word corresponding to its frequency.

KMeans

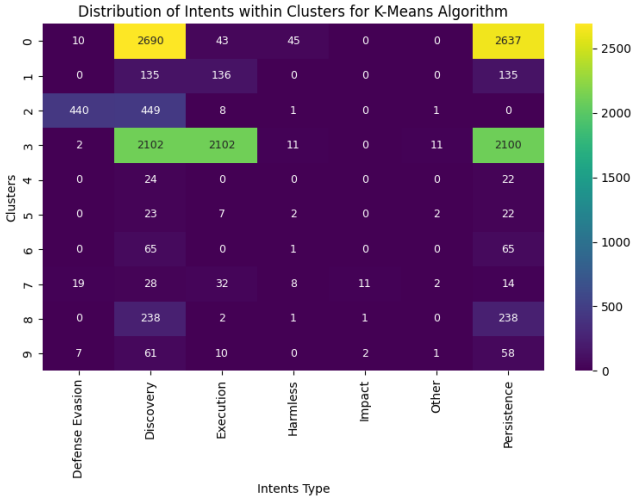

Figure 18: Intent distribution - Kmeans

Figure 19: Words frequency per Cluster - Kmeans

Analysing the higher values in the distribution plot for each cluster, can be noticed that:

- in **cluster 0, 4, 5, 6, 8 and 9** there are sessions classified by MITRA as Discovery and Persistence attacks. Interestingly, the majority of the data points are in cluster 0. This raises the question: why have the other clusters also captured these sessions? Is there an actual difference between the attacks in all these clusters (which, from the MITRE's perspective, are all the same)?
  Firstly, the t-SNE representation indicates a geometric difference. The common feature among all these clusters is their position on the left side of the chart. However, for example, cluster 9 is very distant from cluster 6, and groups of sessions like clusters 8, 6, and 9 are indeed separate portions of datapoints.
  Diving deeper, the frequency analysis shows that all these clusters have the commands **mv** and **rm** among the most

frequent words, that is consistent with Discovery and Persistence attacks. Specifically, techniques used for persistence often involve configuration changes, such as credential modifications, which necessitate moving and removing files in the file system, as 'mv' and 'rm' do.

However, there's more to consider. Taking C6 as example:

(1) The **ss** command is the most used and is not even present in the other clusters. This is a command used in Linux to dump socket statistics and is particularly useful for *monitoring network connections*. It can be used for different malicious purposes, such as obtaining detailed network topology information, identifying key network devices, and pinpointing weak points in the network infrastructure.
(2) Another word particularly used is **dev**, referring to the **/dev** folder in the Linux filesystem, which contains device files (also sometimes known as device special files and device nodes) and provides access to peripheral devices such as hard disks or network interfaces. By modifying and injecting files in this folder (and here the use of rm and mv) is possible for instance the redirect of the network traffic or accessing raw network interfaces to monitor or inject traffic, which can be used for sniffing data or conducting MITM attacks.
(3) The **mount** command is used to *mount new devices* to the system tree, as well as new partitions or filesystems. After gaining access to the network, the attacker could mount a remote share to copy sensitive data or place malicious files on the remote system.

The presence of these commands suggests that cluster 6 likely contains **network attacks**, which might range from a simple discovery to *traffic manipulation*, *sniffing*, or *MITM* attacks. This means that the classification of MITRA is not wrong, but the clustering result in this case provides a more fine-grained classification.

Applying the same reasoning to the other clusters that have apparently classified only Discovery and Persistence attacks could lead to similar insights, indeed each cluster might represent a distinct subset of attacks characterized by specific command usage patterns, indicating different types of activities within the broader categories of Discovery and Persistence.

- in **cluster 1 and 3**, the majority of attacks are classified as Discovery, Execution, and Persistence. However while C3 contains around 2100 datapoints with these labels assigned by MITRE, C1 has just 135 sessions. This suggests that C3 represents a broader subset of these attacks, whereas C1 might indicate a more specific subset.

  From a visual point of view the datapoints appear distinct, C1 is composed by a bunch of items that may share some correlation.

  However, analysing the most common words in the clusters, it's difficult to find differences between them. Both clusters contain sessions that use the same words, with the exception of 'cd' and 'sh,' which are present only in C3 but are also the least common words in the cluster, making this difference insignificant.

  Additionally, all common words (39) have nearly identical frequencies. The top 5 words are exactly the same in both clusters, and even at lower frequencies the deviations are negligible. This suggests that this clustering may not have produced meaningful distinctions. Cluster 1 could potentially be merged into Cluster 3, at least based on the presented analysis.
- in **cluster 2**, the primary attack types are Defense Evasion and Discovery attacks. This classification appears comprehensive, indeed in this cluster the most frequent words include "**proc**", that refers to the **/proc** direcory of the linux filesystem. In this folder there are information that can be used:
  - for **discovery** purposes, indeed files like */proc/cpuinfo* or */proc/meminfo* provide details about the CPU and system memory usage. These details can then be the starting point for attacks targeting hardware capabilities or memory vulnerabilities.
  - for **defense evasion**, in fact the */proc* directory also contains information about every process running on the system. Attackers often exploit trusted processes to hide and masquerade their activities, such as malware.

  Overall, the most frequent words are closely related to these types of intents, indicating that the MITRE classification is reliable and that C2 accurately groups these sessions.
- in **cluster 7** there are no dominant intent, instead it includes in small portion all the label that MITRA was able to assign. However, in the most frequent words there are commands and directories that can provide insights into the attacker's objectives:
  - **nohup**: This command is used to *run other commands or scripts in the background* and/or keep them running even after the *user logs out*. Attackers exploit this for various malicious activities, such as a malicious process continuing running in background so that the attacker can make it less noticeable, or even making a malware running after the disconnection of the user.
  - **/tmp directory**: This directory *stores temporary files* for various purposes in the Linux system. Since */tmp* is writable by all users, attackers can use it to store files needed for the attack, which will be cleaned up after a system reboot.
  - **/bin directory**: This directory contains essential *user command binaries* necessary for system operation, like 'ls', 'cp', 'mv', and 'bash'. An attacker might replace a legitimate binary with a malicious version, adding "extra" functionalities like a backdoor with a reverse shell towards his machine or a keylogger.

  These indicators suggest the preparation of an environment for executing a **backdoor**, **malware**, or **Trojan** that needs to remain hidden within the system. This, combined with the presence of the 'wget' command, hints at various scenarios, such as downloading a malicious script from a remote server, embedding the code inside a standard Linux program from */bin*, and executing it in the background with 'nohup'. The results of the script could be temporarily stored in the */tmp* directory and then deleted soon after.

GMM

As in Kmeans, firstly are presented the analysis of the GMM clustering results in terms of distribution of the intents and word frequency in the sessions:

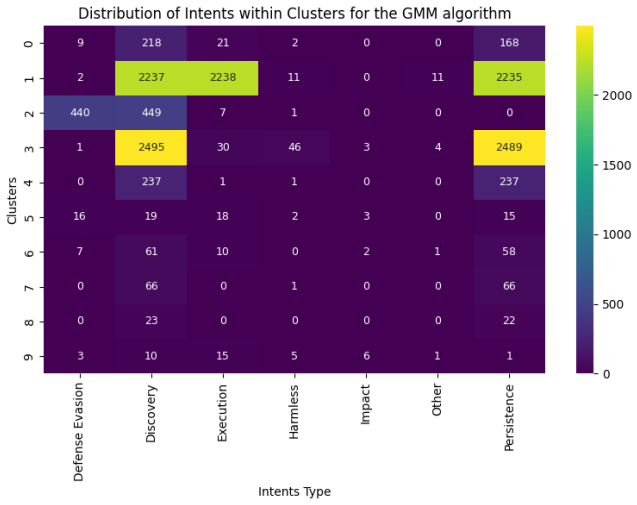

Figure 20: Intent distribution - GMM

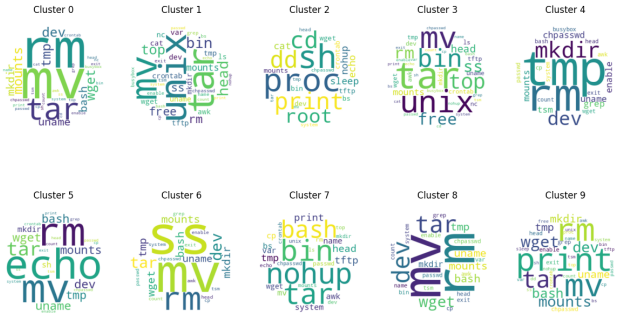

Figure 21: Words frequency per Cluster - GMM

Analysing the higher values in the distribution plot for each cluster, results reveals that:

- in **cluster 0, 3, 4, 6, 7 and 8** mainly consist of sessions associated to Discovery and Persistence by MITRA.
  Even in this scenario, a cluster encompasses nearly all the datapoints corresponding to these labels, even thought there is a significant distinction. In Kmeans, there was the cluster 0 containing all sessions, whereas in this case they are split between C0 and C3.
  Graphically, this division seems sensible as there is clear separation among the points. Moreover, almost all the sessions related to discovery and persistence are in C3 (the violet zone), thus justifying the allocation of other points to a different cluster, like if they are a more specific class.
  However, from a frequency analysis perspective, does this separation make sense? Upon closer examination of both the frequency analysis and of the raw SSH sessions, there appear to be minimal differences in terms of intent. Both types of attacks primarily involve attempting to change the password of a user within the system, downloading data from a remote server, and a prolonged phase of information gathering, aimed at capturing as much information as possible from the victim machine.
  The key disparity lies in the type of information targeted in the sessions. For instance, in C1, there is frequent usage of the **free** command, which provides a detailed report on the system's memory usage (absent in Cluster 0), and the **ss** command, previously mentioned, which provides network information. Conversely, in C0, there is more prevalent use of the **uname** command, which retrieves information about the operating system, kernel, and hardware.
  Hence, while there is a distinction in the execution and implementation of the attacks, the fundamental intent of both clusters remains the same: discovery and persistence.
- in **cluster 1** there are all the datapoints related to Discovery, Execution, and Persistence attacks.
  This marks a significant departure from the K-means results in the preceding section, indeed previously these datapoints were allocated into two separate clusters, C1 and C3. However, from the analysis was evident that probably this separation was not necessary, and here in GMM there is the evidence.
  GMM recognized that these data points are not significantly different and can be amalgamated into a single cluster.
- in **cluster 2** are predominantly observed attacks associated with Defense Evasion and Discovery according to MITRA. This classification is consistent with the results obtained from K-means, even bearing the same numerical labels. Therefore, the description provided earlier remains applicable here.
- in both **cluster 5 and 9** there are attacks classified by the MITRA as having different intents. The rationale behind this classification warrants examination: is there a correlation between these clusters?
  Taking Cluster 5 as an example, the graphical representation via t-SNE suggests that this classification might be forced, as the data points of C5 are in close proximity to other clusters. Furthermore, from the word frequency analysis indicates that these sessions could be classified simply as discovery and persistence. Indeed, even in this sessions, the attacker is trying to change the password, modify some system file, and perform the classic information gathering commands. The difference is in the implementation. For instance, 'echo' emerges as the predominant term, a feature absent in other clusters of the same type like C0, but this does not alter the underlying intent of the attack, echo is just used to pass input to other commands through a pipe, so this cluster could probably have been included in another one, such as C0.
  However a consideration can been done. While avoiding this separation is possible, there are cases where it can prove beneficial. For instance, segregating this cluster could enable the grouping of sessions originating from a particular attacker who employs similar styles and methodologies.

5 LANGUAGE MODELS EXPLORATION

In this section, we aim to furnish a comprehensive overview detailing the functioning of our implementation. Following this, we will proceed to dissect the underlying steps involved in its operation. Subsequently, we will undertake an evaluation of the outcomes obtained, taking into careful consideration the impact of different hyperparameters.

5.1 Text Representation with Doc2Vec

The first step involved representing the SSH attack sessions in a numerical form suitable for input into the machine learning model. We used Doc2Vec, a text representation technique that generated dense vectors for text documents.

Each attack session was transformed into a TaggedDocument. After dividing the data into train and test sets using the custom split (2.5), we created two TaggedDocuments: one for the train set and one for the test set (even though the test set was not used). The TaggedDocument associated a list of words (the attack full session) with a unique label (session ID). The Doc2Vec model was trained using the attack sessions, creating numerical vectors representing each session. These vectors were used as inputs for the neural network model.

To delve deeper into the implementation, we use the custom *"train_doc2vec_model"* function to prepare the data for Doc2Vec. In this function, we pass the vector size, which is a hyperparameter of the model. Within the function, it creates a vocabulary and trains on the train data.

We did not use the TaggedData associated with the test set because we train the model using *train_doc2vec_model* and then use the *infer_vector* method to obtain vectors for both the train and test sets. As the name of the method suggests, it infers a vector based on the prior training.

5.2 Building the Neural Network Model with Keras

The second step was building the neural network model using the Keras API within TensorFlow, which was used to classify the SSH attack sessions. We defined a custom *"create_model"* function to construct the NNM. We used a feedforward neural network with only one hidden dense layer consisting of neurons with ReLU activation functions, introducing non-linearity to the model. The output layer had seven neurons with sigmoid activation, corresponding to the seven possible attack classes. The sigmoid activation function was appropriate for multi-label classification as it returned a probability for each class.

5.2.1 ***Compiling the Model.*** The model was compiled with the binary_crossentropy loss function, suitable for multi-label classification, and the Adam optimizer, a stochastic gradient descent-based optimization algorithm. Keras simplified this process with its API.

5.3 Training and Evaluating the Model

The dataset was divided into two parts: training and test sets. The training set was used to train the model, while the test set was reserved for evaluating the model's performance on unseen data. This split was crucial for assessing how well the model generalized to new data.

In the training phase, 70% of the dataset was utilized to train the model, enabling it to learn from the data. Following this, the model's performance was assessed using the remaining 30% of the data, referred to as the test set. This helped in evaluating the model's performance on data it had not seen before, providing an unbiased estimate of its accuracy and generalization capability.

In multi-label classification, each instance could belong to multiple classes. To handle this, we used the MultiLabelBinarizer from scikit-learn, which transformed the labels into a binary format suitable for neural networks.

Each label set was transformed into a binary vector where each element corresponded to a class. If an instance belonged to a class, the corresponding element was 1; otherwise, it was 0.

5.3.1 ***Model Training.*** Training the model involves several key steps:

- Data Conversion: The training data (both features and labels) is converted into tensors, the native data format for TensorFlow.
- Model Fitting: The neural network model is trained on the training data. During training, the model learns to map input features (Doc2Vec vectors) to output labels (binary vectors).
- Monitoring Performance: Throughout the training process, the model's performance is monitored using the loss function (binary_crossentropy) and accuracy metrics on both training and validation sets.
- Loss and Accuracy: The model's loss and accuracy are measured on the test set to determine how well it performs on unseen data.
- Classification Reports: These reports include precision, recall, and f1-score for each class, providing insights into how well the model performs across different categories.

5.4 Hyperparameter Experiments

Hyperparameter tuning is a critical process in machine learning, involving the optimization of parameters that control the learning process. Here, we explore several key hyperparameters:

- Vector Size: The size of the vectors generated by Doc2Vec affects the representation of the text data.
- Learning Rate: The learning rate controls how quickly the model adjusts its weights during training.
- Epochs: Defines the number of times the entire dataset is passed through the model during training. More epochs can potentially improve the model's performance by allowing it to learn more complex patterns in the data. However, too many epochs may lead to overfitting.

5.4.1 ***Refinement of the splitting strategy and categorization of hyperparameter types.*** For each model, we employed the same splitting strategy to facilitate a consistent evaluation of different hyperparameters. This approach allowed us to effectively compare the performance of various configurations and identify the optimal ones that generalized well across our models while

maintaining high accuracy. For this reason, we encountered the same support across different model reports.

In our implementation, we experimented with vector sizes of 100, 300, and 800. Moreover, we experimented with learning rates of 0.001, 0.01, and 0.1. A lower learning rate could lead to more precise but slower convergence, while a higher learning rate could speed up training but risk overshooting the optimal solution.

5.4.2 ***The architecture of the neural network.*** The architecture of the neural network, including the number and size of hidden layers, also acts as a hyperparameter. In our neural network architecture, we incorporated one hidden dense layer, each comprising a specific number of neurons (following the equation 1) and ReLU activation functions.

$$hidden\_layers\_neurons = (vector\_size * \frac{2}{3}) + 7 \quad (1)$$

Theoretical predictions can guide us, practical experimentation is often necessary, there are some common rules of thumb for determining the number of hidden neurons. It should be between the size of the input layer and the size of the output layer, moreover, the number of hidden neurons can be calculated as $\frac{2}{3}$ (or 70% to 90%) of the size of the input layer, plus the size of the output layer [4] and additionally, the number of hidden neurons should be less than twice the size of the input layer [3].

The number of layers and neurons per layer can significantly impact the model's ability to learn complex patterns. Moreover, the choice of activation functions is crucial in shaping the behavior of the neural network. We opted for ReLU activation functions in the hidden layers due to their effectiveness in introducing non-linearity, enabling the network to model more intricate data patterns. For the output layer, we employed the sigmoid activation function, particularly suited for multi-label classification tasks like ours.

We trained individual models for each combination of vector size and learning rate, enabling a systematic assessment of each hyperparameter's influence. Following training, we evaluated these models on the test set and compared their performance metrics, including loss, accuracy, precision, recall, and F1-score.

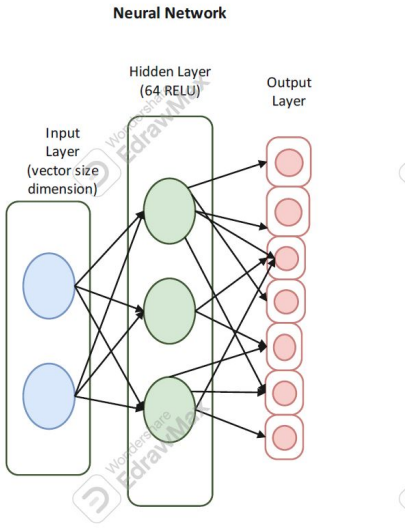

Figure 22: Simple schema which emulates our Neural Network Diagram, Done with Edraw-max Free Version (Our number of neurons in the hidden layer depends on the previous formula is not 64 as the figure show)

5.5 Results

5.5.1 ***How to read Training and Validation Loss plots?*** The x-axis represents the number of epochs, which is the number of times the learning algorithm has passed through the entire training dataset. The y-axis represents the loss, which measures how well the model's predictions match the actual results. A lower loss indicates a better model performance. The plot includes two lines, the blue line represents the training loss and the orange line represents the validation loss. Is possible to see an example in Figure 23 (**this plot is not generated from our data**).

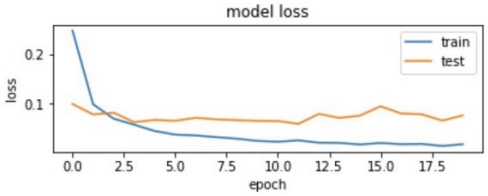

Figure 23: Train and validation loss Example plot (not from our data)

5.5.2 ***Vector size 100.*** In this section, we analyze our model using a vector size of 100 and learning rates of 0.1, 0.01, and 0.001.

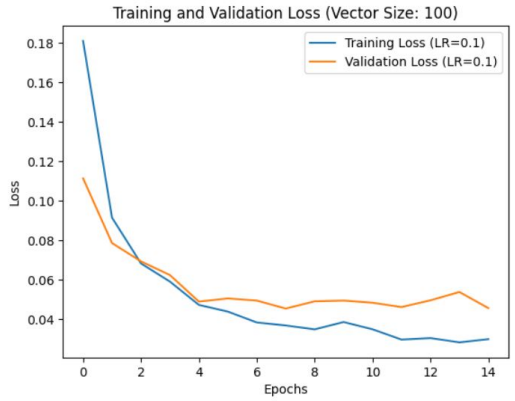

Figure 24: Train and validation loss (vec. size 100, Learning rate 0.1)

This plot (24) shows the training and validation loss for a model with a vector size of 100 and a learning rate of 0.1 over 15 epochs.

Initially (epoch 0), both the training and validation losses start at relatively high values (training loss starts around 0.18), this behavior is common as the model begins learning from random initial weights (untrained state). As training progresses through successive epochs the training loss initially decreases sharply but then varies, showing multiple peaks and troughs. Similarly, the validation loss demonstrates considerable variability. However, it does not decrease as sharply as the training loss. The validation loss shows some fluctuations and begins to plateau around epoch 5, but at a higher loss value than the training loss. The fluctuations in both training and validation losses suggest that the learning rate of 0.1 might be too high, causing the model to oscillate and fail to converge smoothly (could lead to overfitting or poor generalization capability).

Given these observations, it would be advisable to reduce the learning rate to achieve a more stable training process and potentially yield better generalization.

Table 11: Classification Report on Test Set (vec. size 100, Learning rate:0.1)

Class	Precision	Recall	F1-Score	Support
Defense Evasion	0.89	0.94	0.91	239
Discovery	1.00	1.00	1.00	2908
Execution	0.98	0.93	0.95	1170
Harmless	0.67	0.06	0.11	34
Impact	0.00	0.00	0.00	7
Other	1.00	0.50	0.67	8
Persistence	0.99	0.99	0.99	2645
Micro Avg	0.99	0.98	0.98	7011
Macro Avg	0.79	0.63	0.66	7011
Weighted Avg	0.98	0.98	0.98	7011
Samples Avg	0.99	0.98	0.98	7011

Let's analyze the test set(table: 11). Here, precision, recall, and F1-score metrics give us valuable insights into how effectively our model is distinguishing between various threat categories. For the Defense Evasion class, the model demonstrates high precision (0.89) and recall (0.94), resulting in a strong F1-score of 0.91, indicating a robust ability to classify instances of this class correctly. The Discovery class shows perfect precision, recall, and F1-score, reflecting the model's excellent performance. The Execution class also performs well, with high precision (0.98), recall (0.93), and F1-score (0.95). However, the Harmless class exhibits significant challenges, with precision at 0.67 and very low recall at 0.06, leading to a poor F1-score of 0.11. The model fails to classify any instances of the Impact class correctly, with all metrics at 0.00. The Other class has high precision (1.00) but moderate recall (0.50), resulting in an F1-score of 0.67. The Persistence class performs excellently, with metrics near 1.00. Overall, the Micro Average is high at 0.98 for precision, recall, and F1-score, indicating strong overall performance. The Macro and Weighted Averages are strong, though the Macro Avg shows lower performance, reflecting challenges in minority classes such as Harmless and Impact.

Support values give us an understanding of the distribution of different threat types in our dataset. For example, the 'Discovery' class has a support of 2908, indicating a significant presence of this threat type in our test set. On the other hand, classes like 'Harmless,' 'Impact,' and 'Other' have much lower support, suggesting that these threat types are less prevalent in our split.

Table 12: Classification Report on Train Set (vec. size 100, Learning rate:0.1)

Class	Precision	Recall	F1-Score	Support
Defense Evasion	0.96	0.97	0.97	239
Discovery	1.00	1.00	1.00	2907
Execution	0.99	0.97	0.98	1170
Harmless	1.00	0.17	0.29	35
Impact	1.00	0.29	0.44	7
Other	1.00	0.89	0.94	9
Persistence	1.00	1.00	1.00	2646
Micro Avg	0.99	0.99	0.99	7013
Macro Avg	0.99	0.76	0.80	7013
Weighted Avg	0.99	0.99	0.99	7013
Samples Avg	0.99	0.99	0.99	7013

The metrics here (tab: 12) show a similar trend to those of the test set (tab: 11). This means that the model is capable of learning the patterns of the train set. For the Defense Evasion class, the model shows very high precision and recall, resulting in a strong F1-score of 0.97. The Discovery class maintains perfect performance with precision, recall, and F1-score all at 1.00. The Execution class performs well with precision (0.99), recall (0.97), and F1-score (0.98). The Harmless class has high precision (1.00) but very low recall (0.17), leading to a low F1-score of 0.29. The Impact class shows high precision (1.00) but low recall (0.29), resulting in an F1-score of 0.44. The Other class, despite having little support, shows good performance with precision (1.00), recall (0.89), and F1-score (0.94). The Persistence class performs excellently with perfect scores. Overall, the Micro Average is very high at 0.99 for precision, recall, and F1-score. The Macro and Weighted Averages are high, though the Macro Avg reflects lower performance in minority classes such as Harmless and Impact.

The model with a vector size of 100 and a learning rate of 0.1 demonstrates strong overall performance, especially in the 'Discovery' and 'Persistence' classes. However, there are challenges

in minority classes such as 'Harmless' and 'Impact'. The training set shows slightly better performance compared to the test set, indicating potential overfitting issues. However, by checking the performance on the test set, we can rule out overfitting. For the Defense Evasion and Persistence classes, both the train and test sets show high performance, indicating the model's robustness in these classes. The Discovery class consistently performs perfectly across both sets, showcasing the model's strength. Minority classes, such as Harmless and Impact, experience a significant performance drop, especially in the test set, indicating the need for better handling of class imbalance

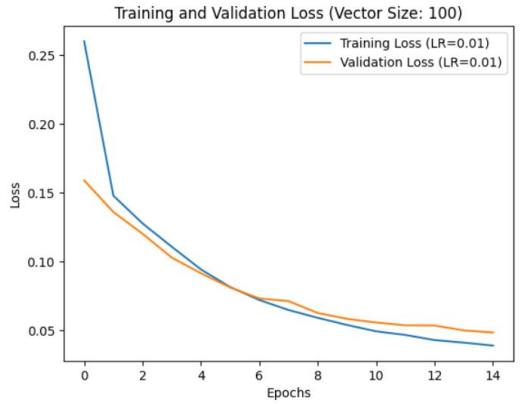

Figure 25: Train and validation loss (vec. size 100, Learning rate 0.01)

At the beginning of figure 25, training loss starts at the highest point on the y-axis and decreases sharply, leveling off around epoch 4. The validation loss starts high but decreases more gradually, intersecting with the training loss around epoch 5 before continuing to decrease. This graph suggests effective learning without overfitting, as the validation loss continues to decrease alongside the training loss.

Comparing this figure 25 to figure 24, the most notable difference is the behavior of the validation loss. In 24, the validation loss plateaued after a certain point, which could suggest overfitting. In contrast, this graph (25) shows the validation loss decreasing steadily, which is indicative of a well-fitting model that is learning generalizable patterns from the training data.

Table 13: Classification Report on Test Set (vec. size 100, Learning rate:0.01)

Class	Precision	Recall	F1-Score	Support
Defense Evasion	0.94	0.90	0.92	239
Discovery	1.00	1.00	1.00	2908
Execution	0.96	0.95	0.95	1170
Harmless	1.00	0.09	0.16	34
Impact	0.50	0.14	0.22	7
Other	0.75	0.38	0.50	8
Persistence	0.98	1.00	0.99	2645
Micro Avg	0.98	0.98	0.98	7011
Macro Avg	0.88	0.63	0.68	7011
Weighted Avg	0.98	0.98	0.98	7011
Samples Avg	0.98	0.98	0.98	7011

Analysing Table 13, the Defense Evasion class exhibits a precision of 0.94, recall of 0.90, and F1-score of 0.92. The Discovery class maintains perfect performance with precision, recall, and F1-score all at 1.00, demonstrating the model's excellent handling of this class. The Execution class shows high performance with precision at 0.96, recall at 0.95, and an F1-score of 0.95. The Harmless class, however, struggles with a precision of 1.00 but a very low recall of 0.09, leading to a poor F1-score of 0.16, indicating difficulty in correctly identifying these instances. For the Impact class, precision is at 0.50, but recall is low at 0.14, resulting in an F1-score of 0.22, showing that the model is not sufficiently effective in classifying these instances. The Other class has a precision of 0.75 and a recall of 0.38, resulting in an F1-score of 0.50, indicating good precision but moderate recall. The Persistence class performs excellently with a precision of 0.98, recall of 1.00, and F1-score of 0.99.

Table 14: Classification Report on Train Set (vec. size 100, Learning rate:0.01)

Class	Precision	Recall	F1-Score	Support
Defense Evasion	0.99	0.92	0.95	239
Discovery	1.00	1.00	1.00	2907
Execution	0.97	0.97	0.97	1170
Harmless	1.00	0.06	0.11	35
Impact	1.00	0.43	0.60	7
Other	1.00	0.33	0.50	9
Persistence	0.98	1.00	0.99	2646
Micro Avg	0.99	0.99	0.99	7013
Macro Avg	0.99	0.67	0.73	7013
Weighted Avg	0.99	0.99	0.98	7013
Samples Avg	0.99	0.99	0.98	7013

Instead here (table 14) the Defense Evasion class shows a precision of 0.99, recall of 0.92, and an F1-score of 0.95, reflecting very high performance with a slight drop in recall. The Discovery class continues to exhibit perfect performance with precision, recall, and F1-score all at 1.00. The Execution class has high metrics with precision and recall both at 0.97 and an F1-score of 0.97. The Harmless class, however, shows a precision of 1.00 but a low recall of 0.06, resulting in an F1-score of 0.11, indicating issues with correctly identifying all instances despite high accuracy when it does. The Impact class maintains perfect precision at 1.00 but with a low recall of 0.43, leading to an F1-score of 0.60. The Other class has a precision of 1.00 and a recall of 0.33, resulting in an F1-score of 0.50, showing moderate performance. The Persistence class exhibits excellent performance with near-perfect precision, recall, and F1-score at 0.99.

Comparing the models with vector size 100 and learning rates of 0.1 and 0.01, the lower learning rate of 0.01 generally results in improved performance, particularly in the test set. For the Defense Evasion class, precision improved from 0.89 to 0.94, as well as the F1-Score, which increased from 0.91 to 0.92. Conversely, recall decreased slightly from 0.94 to 0.90. The Discovery class maintained perfect scores across both learning rates, showcasing the model's robustness for this class. The Execution class showed consistent high performance, with only minor differences in metrics. The Harmless class saw a significant improvement in precision from 0.67 to 1.00 and a slight increase in recall from 0.06 to 0.09, resulting in a

better F1-score from 0.11 to 0.16, though still indicating difficulties. The Impact class saw precision improvement from 0.00 to 0.50 and a slight increase in recall from 0.00 to 0.14, leading to a notable increase in F1-score from 0.00 to 0.22. The Other class is the only one that has shown worsened performance . The Persistence class remained consistent. Overall, the model with a learning rate of 0.01 shows a more balanced performance with better generalization, particularly for minority classes.

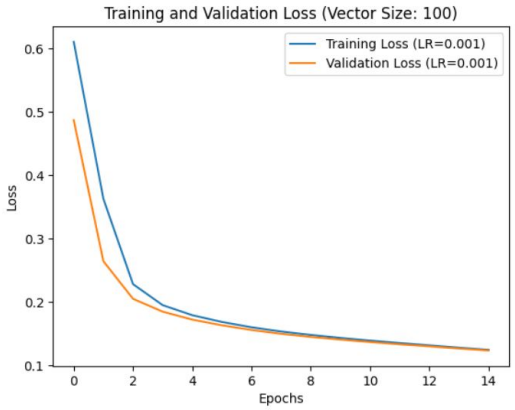

Figure 26: Train and validation loss (vec. size 100, Learning rate 0.001)

In Figure 26, both the training and validation losses exhibit a downward trend, indicating a reduction in loss over epochs. Initially high, the training loss decreases sharply as epochs progress and stabilizes around epoch 5, reaching a lower value. The validation loss also starts high but decreases more gradually compared to the training loss. It intersects with the training loss around epoch 5-6 before continuing to decrease at a slower rate. This graph suggests effective learning without overfitting, as both losses decrease. However, upon closer examination of the y-axes, it becomes apparent that the loss values are higher compared to the previous graphs (Figure 25 and Figure24). Despite the seemingly better balance between training and validation losses, the overall loss is higher when transitioning from a learning rate of 0.1 to 0.001.

Table 15: Classification Report on Test Set (vec. size 100, Learning rate:0.001)

Class	Precision	Recall	F1-Score	Support
Defense Evasion	0.30	0.01	0.02	239
Discovery	0.99	1.00	1.00	2908
Execution	0.88	0.86	0.87	1170
Harmless	0.50	0.03	0.06	34
Impact	0.00	0.00	0.00	7
Other	0.00	0.00	0.00	8
Persistence	0.91	1.00	0.95	2645
Micro Avg	0.94	0.93	0.94	7011
Macro Avg	0.51	0.41	0.41	7011
Weighted Avg	0.91	0.93	0.92	7011
Samples Avg	0.94	0.93	0.93	7011

Here, the test set values have notably worsened compared to lr 0.1. For instance, taking Defense Evasion as an example, the precision drops from 0.89 (with lr 0.1) to 0.30 in the current case, indicating a substantial decline. Similarly, the F1-scores also decline significantly. This decline trend applies to all classes; for instance, the most noticeable case is Impact and Other, where precision, recall, and F1-score all reach 0. Given these observations, a learning rate of 0.001 doesn't seem to be the optimal choice compared to previous ones, a hypothesis also supported by the previously analyzed graphs.

Table 16: Classification Report on Train Set (vec. size 100, Learning rate:0.001)

Class	Precision	Recall	F1-Score	Support
Defense Evasion	0.43	0.01	0.02	239
Discovery	0.99	1.00	1.00	2907
Execution	0.88	0.86	0.87	1170
Harmless	1.00	0.03	0.06	35
Impact	0.00	0.00	0.00	7
Other	0.00	0.00	0.00	9
Persistence	0.91	1.00	0.95	2646
Micro Avg	0.94	0.94	0.94	7013
Macro Avg	0.60	0.41	0.41	7013
Weighted Avg	0.92	0.94	0.92	7013
Samples Avg	0.94	0.93	0.93	7013

Similar to the case observed for the test set, a comparable trend can be noted for the training set (tab. 16), where the values deteriorate and reaffirm the earlier notion of an excessively low learning rate. For instance, taking Defense Evasion as a reference, the precision decreases from 0.99 (with lr. 0.01) to 0.43 in the current scenario, reflecting a significant decline.

For the model trained with a vector size of 100 and a learning rate of 0.001, the test set loss is 0.12 and the accuracy is 91.01%. When the vector size remains at 100 but the learning rate is increased to 0.01, the test set loss decreases to 0.048, with an accuracy of 73.33%. However, with the same vector size of 100 and a higher learning rate of 0.1, the test set loss remain at 0.04, while the accuracy increase at 89.57%.

These results suggest that decreasing the learning rate from 0.01 to 0.001 leads to a higher loss.

5.5.3 ***Vector size 300***. In this section, we have a vector size of 300 and learning rates of 0.1, 0.01, and 0.001.

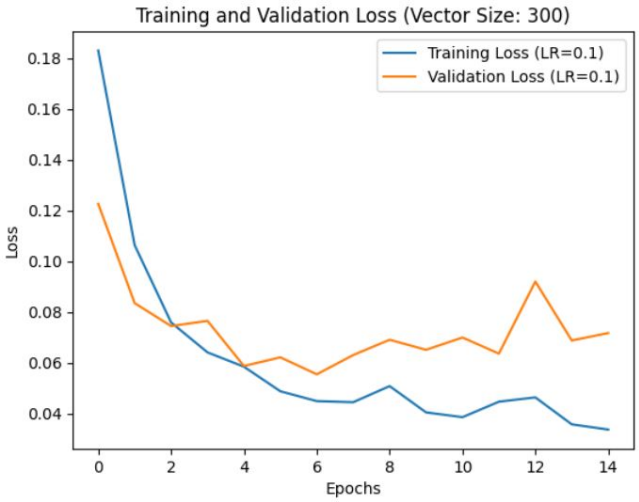

Figure 27: Train and validation loss (vec. size 300, Learning rate 0.1)

In Figure 24 initially, the training loss decreases significantly. However, around epoch 7, we notice fluctuations. These issues could be due to a learning rate that is too high, causing the model to skip the optimal solution. On the other hand, the validation loss also decreases initially, but it shows more variability and after epoch 7, it starts to increase slightly. This suggests that the model might be overfitting (performs worse on unseen data).

Table 17: Classification Report on Test Set (vec. size 300, Learning rate 0.1)

Class	Precision	Recall	F1-Score	Support
Defense Evasion	0.92	0.92	0.92	239
Discovery	1.00	1.00	1.00	2908
Execution	0.92	0.97	0.94	1170
Harmless	0.67	0.12	0.20	34
Impact	0.00	0.00	0.00	7
Other	0.67	0.50	0.57	8
Persistence	0.98	0.99	0.99	2645
Micro Avg	0.98	0.98	0.98	7011
Macro Avg	0.74	0.64	0.66	7011
Weighted Avg	0.97	0.98	0.98	7011
Samples Avg	0.98	0.98	0.98	7011

Table 18: Classification Report on Train Set (vec. size 300, Learning rate 0.1)

Class	Precision	Recall	F1-Score	Support
Defense Evasion	0.97	0.96	0.96	239
Discovery	1.00	1.00	1.00	2907
Execution	0.93	0.98	0.96	1170
Harmless	1.00	0.20	0.33	35
Impact	1.00	0.29	0.44	7
Other	0.83	0.56	0.67	9
Persistence	0.99	1.00	0.99	2646
Micro Avg	0.98	0.99	0.99	7013
Macro Avg	0.96	0.71	0.77	7013
Weighted Avg	0.98	0.99	0.98	7013
Samples Avg	0.98	0.99	0.98	7013

Analyzing the performance between the training (table 18) and test (table 17) sets, with a learning rate of 0.1, we can observe:
For the training set the model achieves high precision, recall, and F1-scores across most classes. "Discovery," "Execution," and "Persistence" classes have particularly high scores close to 1.00, indicating excellent performance. However, the "Harmless," "Impact," and "Other" classes have significantly lower F1-scores. In the test set, the model's performance is generally lower compared to the training set. While "Defense Evasion", "Discovery and "Persistence" still maintain high F1-scores, the "Harmless," and "Impact" classes continue to show very low F1-scores, with "Impact" having a zero F1-score, indicating no correct predictions for this class.

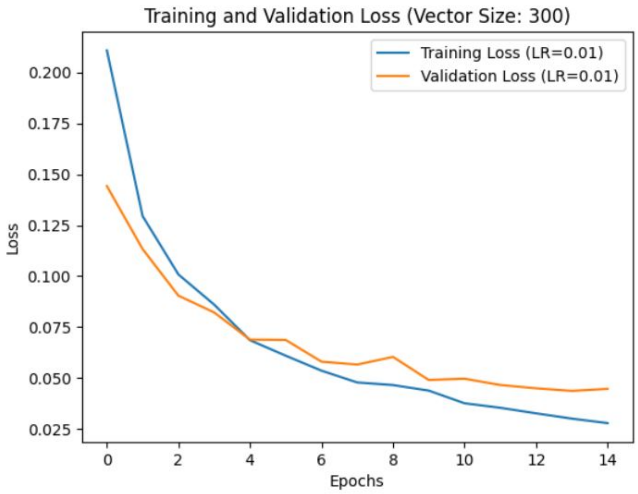

Figure 28: Train and validation loss (vec. size 300, Learning rate 0.01)

This graph (Figure 28), like the previous one (Figure 27), tracks the training and validation loss over epochs, but with a notable difference in the learning rate, which is now set at 0.01 compared to the earlier rate of 0.1. There we see that both the training and validation losses exhibit a steady decline as the number of epochs increases. The lower learning rate seems to provide a smoother descent for both losses, suggesting a more stable learning process. The absence of significant spikes or fluctuations indicates that the model is not experiencing the same degree of overfitting observed in the previous graph around epoch 7. In this graph, shows a consistent decrease in validation loss, which implies better generalization to new data. Moreover, the smoother curves could be attributed to the lower learning rate, which allows the model to make smaller, more precise adjustments to its weights.

In conclusion, the comparison between the two graphs underscores the importance of choosing the right learning rate. While a higher rate can accelerate the initial learning process, it may also lead to instability and overfitting. A lower learning rate, as shown in the current graph, can foster a more gradual but potentially more reliable learning curve, leading to a model that generalizes better.

Table 19: Classification Report on Test Set (vec. size 300, Learning rate 0.01)

Class	Precision	Recall	F1-Score	Support
Defense Evasion	0.91	0.92	0.92	239
Discovery	1.00	1.00	1.00	2908
Execution	0.97	0.95	0.96	1170
Harmless	0.71	0.15	0.24	34
Impact	0.50	0.14	0.22	7
Other	1.00	0.62	0.77	8
Persistence	0.99	0.99	0.99	2645
Micro Avg	0.99	0.98	0.98	7011
Macro Avg	0.87	0.68	0.73	7011
Weighted Avg	0.99	0.98	0.98	7011
Samples Avg	0.98	0.98	0.98	7011

Table 20: Classification Report on Train Set (Vector Size: 300, Learning rate 0.01))

Class	Precision	Recall	F1-Score	Support
Defense Evasion	0.97	0.97	0.97	239
Discovery	1.00	1.00	1.00	2907
Execution	0.98	0.98	0.98	1170
Harmless	1.00	0.17	0.29	35
Impact	0.80	0.57	0.67	7
Other	1.00	0.78	0.88	9
Persistence	0.99	1.00	0.99	2646
Micro Avg	0.99	0.99	0.99	7013
Macro Avg	0.96	0.78	0.83	7013
Weighted Avg	0.99	0.99	0.99	7013
Samples Avg	0.99	0.99	0.99	7013

When comparing the model with vector size 300 and a learning rate of 0.01 (table 20 and table 19) to the one with a learning rate of 0.1 (table 18 and table 17), there are some notable improvements. In the test set, the lower learning rate of 0.01 results in better F1-Score performance for the classes with less support. For example, the F1-Score for the Impact class increases from 0 to 0.44. In the training set, precision, recall, and F1-scores remain high for "Defense Evasion", "Discovery", "Execution" and "Persistence" classes. However, the "Harmless" class remains low.

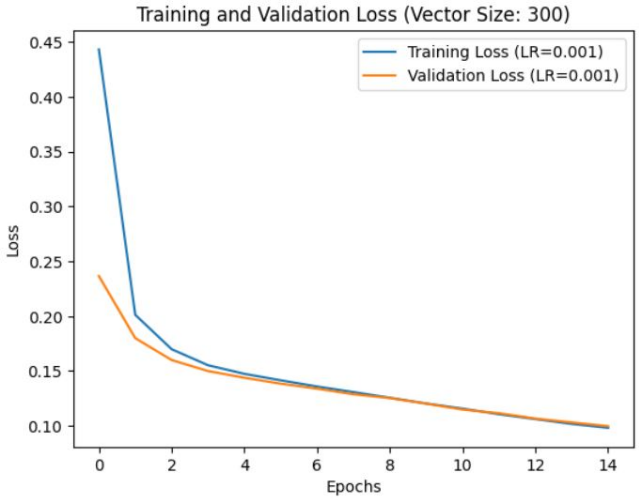

Figure 29: Train and validation loss (vec. size 300, Learning rate 0.001)

As we reflect on the progression from the previous graphs, we notice a trend towards stability and improved generalization. The first graph (Figure 27), showed us a model that was learning quickly but perhaps too eagerly, leading to fluctuations and potential overfitting. The second graph (Figure 28), demonstrated a more balanced approach, with smoother curves and a validation loss that continued to decline alongside the training loss.

Now, with the third graph (Figure 29), the further reduced learning rate seems to have refined the model's learning process even more. The training loss descends steadily without any abrupt drops, and the validation loss mirrors this trend, suggesting that the model is not just memorizing the training data but truly learning from it.

After comparing the three, one would conclude that Figure 29 is the preferable choice for model training due to its stability and consistent behavior. However, despite these favorable characteristics, it's important to note that the loss values in this last graph are significantly higher, indicating that a learning rate of 0.001 might not be the best choice.

Table 21: Classification Report on Test Set (vec. size 300, Learning rate 0.001)

Class	Precision	Recall	F1-Score	Support
Defense Evasion	0.84	0.33	0.47	239
Discovery	0.99	1.00	1.00	2908
Execution	0.91	0.88	0.90	1170
Harmless	1.00	0.03	0.06	34
Impact	0.00	0.00	0.00	7
Other	0.00	0.00	0.00	8
Persistence	0.93	0.99	0.96	2645
Micro Avg	0.95	0.95	0.95	7011
Macro Avg	0.67	0.46	0.48	7011
Weighted Avg	0.95	0.95	0.94	7011
Samples Avg	0.96	0.95	0.95	7011

Table 22: Classification Report on Train Set (vec. size 300, Learning rate 0.001)

Class	Precision	Recall	F1-Score	Support
Defense Evasion	0.93	0.30	0.45	239
Discovery	0.99	1.00	1.00	2907
Execution	0.92	0.89	0.90	1170
Harmless	1.00	0.06	0.11	35
Impact	0.00	0.00	0.00	7
Other	0.00	0.00	0.00	9
Persistence	0.93	0.99	0.96	2646
Micro Avg	0.96	0.95	0.95	7013
Macro Avg	0.68	0.46	0.49	7013
Weighted Avg	0.95	0.95	0.94	7013
Samples Avg	0.96	0.95	0.95	7013

With a further reduction in the learning rate to 0.001 (table 21 and table 22), the model's performance changes again. In the test set, the performance is comparable to the training set, showing poor results across different classes. The model is not able to generalize well with these hyperparameters. In the training set, the high scores for "Discovery," "Execution," and "Persistence" slightly deteriorate, though they remain relatively high. However, the "Other"

and "Impact" classes notice a drop, compared to the learning rate of 0.01 for example.

5.5.4 ***Vector size 800.*** In this section, we have a vector size of 800 with the same 3 learning rate.

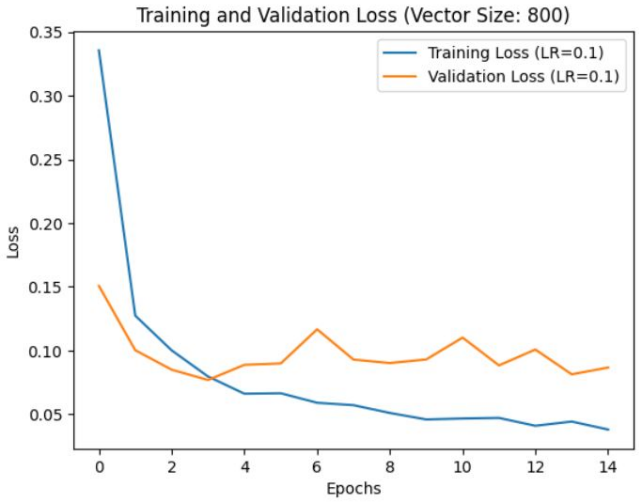

Figure 30: Train and validation loss (vec. size 800, Learning rate 0.1)

In Figure 30 at the beginning, both the training and validation losses start at approximately 0.35 (validation loss around 0.15). As the epochs progress, the training loss decreases more rapidly than the validation loss. Wherease, the validation loss shows more fluctuations compared to the training loss. These fluctuations can occur due the learning rate, which affects how quickly the model adapts. If it's too high, the model might overshoot optimal weights, leading to fluctuations. The learning rate of 0.1 seems to be effective as the losses are decreasing, but if the validation loss starts to increase while the training loss continues to decrease, it might be a sign to lower the learning rate to prevent overfitting and fluctuations.

Table 23: Classification Report on Test Set (vec. size 800, Learning rate 0.1)

Class	Precision	Recall	F1-Score	Support
Defense Evasion	0.89	0.91	0.90	239
Discovery	1.00	1.00	1.00	2908
Execution	0.94	0.95	0.95	1170
Harmless	0.33	0.09	0.14	34
Impact	0.00	0.00	0.00	7
Other	0.31	0.62	0.42	8
Persistence	0.98	0.98	0.98	2645
Micro Avg	0.98	0.98	0.98	7011
Macro Avg	0.64	0.65	0.63	7011
Weighted Avg	0.97	0.98	0.97	7011
Samples Avg	0.98	0.98	0.97	7011

The test set (table 23) results showed that the model maintained high performance for the "Discovery" and "Persistence" classes with F1-scores close to 1.00. The "Defense Evasion" class had an F1-score of 0.90, showing consistency between training and test sets. The "Execution" class also performed well with an F1-score of 0.95. However, the "Harmless" class had a very low F1-score of 0.14, and the "Impact" class had an F1-score of 0.00, indicating no correct classifications for this class. The "Other" class showed some improvement with an F1-score of 0.42. The overall micro average F1-score was 0.98, and the macro average was 0.63, indicating that while the model performs well overall, it struggles with minority classes.

Table 24: Classification Report on Train Set (vec. size 800, Learning rate 0.1)

Class	Precision	Recall	F1-Score	Support
Defense Evasion	0.95	0.94	0.94	239
Discovery	1.00	1.00	1.00	2907
Execution	0.96	0.98	0.97	1170
Harmless	1.00	0.23	0.37	35
Impact	1.00	0.29	0.44	7
Other	0.53	0.89	0.67	9
Persistence	0.99	0.99	0.99	2646
Micro Avg	0.98	0.99	0.99	7013
Macro Avg	0.92	0.76	0.77	7013
Weighted Avg	0.98	0.99	0.98	7013
Samples Avg	0.98	0.99	0.98	7013

The model (table 24) performed excellently on the "Discovery" class with perfect precision, recall, and F1-score of 1.00, similar to smaller vector sizes. The "Defense Evasion" class also showed strong performance with an F1-Score of 0.94. The "Execution" and "Persistence" classes maintained high scores, with F1-scores of 0.97 and 0.99, respectively. However, the "Harmless" class had a low F1-score of 0.37 despite perfect precision, indicating issues with recall (0.23). The "Impact" and "Other" classes showed moderate results, with F1-scores of 0.44 and 0.67, respectively. Overall, the model's weighted average F1-score was high at 0.98, but the macro average was lower at 0.77 due to poor performance on the minority classes.

When comparing the performance of models with vector sizes of 100 and 300 to the vector size of 800 at a learning rate of 0.1, the larger vector size did not lead to significant improvements. The training set results for vector size 800 showed slightly better precision and recall for certain classes, but the F1-scores remained consistent with smaller vector sizes. The test set results also did not show substantial improvement with the larger vector size, with minority classes still underperforming. This indicates that increasing the vector size to 800 does not significantly enhance model performance for this dataset and problem, as the issues with class imbalance and difficult-to-predict classes persist.

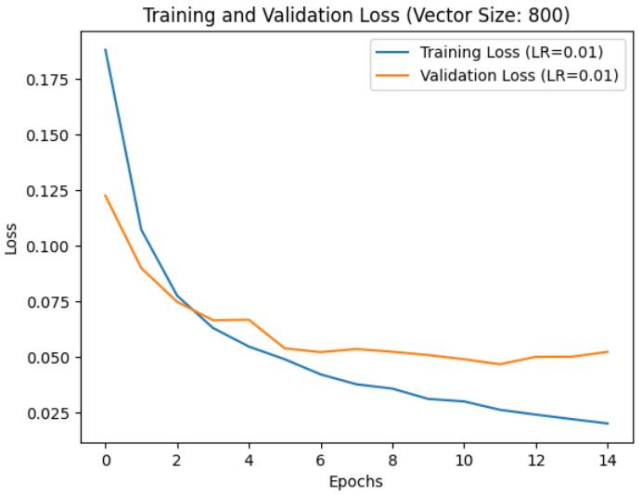

Figure 31: Train and validation loss (vec. size 800, Learning rate 0.01)

The previous graph (Figure 30) with a higher learning rate of 0.1 showed more pronounced fluctuations in the validation loss. In contrast, the current graph (31) with a lower learning rate of 0.01 shows a smoother decline in both training and validation losses, indicating a more stable learning process. In both graphs, the training loss decreases at a faster rate than the validation loss. However, the gap between the training and validation loss in the current graph is narrower compared to the previous one, suggesting better generalization. The fluctuations in the validation loss are less volatile in the current graph, which could mean that the model is not overfitting as much as in the previous one. Similarly, as observed in the previous case with different vector sizes, here we can see that the loss is higher compared to the previous learning rate with the same vector size.

Table 25: Classification Report on Test Set (vec. size 800, Learning rate 0.01)

Class	Precision	Recall	F1-Score	Support
Defense Evasion	0.81	0.96	0.88	239
Discovery	1.00	1.00	1.00	2908
Execution	0.98	0.94	0.96	1170
Harmless	0.86	0.18	0.29	34
Impact	1.00	0.14	0.25	7
Other	1.00	0.75	0.86	8
Persistence	0.99	0.98	0.99	2645
Micro Avg	0.98	0.98	0.98	7011
Macro Avg	0.95	0.71	0.75	7011
Weighted Avg	0.98	0.98	0.98	7011
Samples Avg	0.98	0.98	0.98	7011

Table 26: Classification Report on Train Set (vec. size 800, Learning rate 0.01)

Class	Precision	Recall	F1-Score	Support
Defense Evasion	0.94	0.99	0.97	239
Discovery	1.00	1.00	1.00	2907
Execution	1.00	0.99	0.99	1170
Harmless	1.00	0.26	0.41	35
Impact	1.00	0.57	0.73	7
Other	1.00	1.00	1.00	9
Persistence	1.00	0.99	0.99	2646
Micro Avg	1.00	0.99	0.99	7013
Macro Avg	0.99	0.83	0.87	7013
Weighted Avg	1.00	0.99	0.99	7013
Samples Avg	0.99	0.99	0.99	7013

For the test set with a vector size of 800 and a learning rate of 0.01, the "Defense Evasion" class shows a precision of 0.81, recall of 0.96, and an F1-score of 0.88. This indicates high recall but somewhat lower precision, suggesting the model captures most instances but has some false positives. "Discovery" maintains perfect scores (1.00) across all metrics, indicating consistent performance. "Execution" shows high precision (0.98) and recall (0.94), leading to a strong F1-score (0.96). The "Harmless" class has a lower precision (0.86) and recall (0.18), resulting in an F1-score of 0.29, indicating poor performance for this class. "Impact" shows high precision (1.00) but very low recall (0.14), leading to an F1-score of 0.25, suggesting issues with recall. The "Other" class has perfect precision (1.00) but lower recall (0.75), resulting in an F1-score of 0.86. "Persistence" maintains high performance with a precision of 0.99, recall of 0.98, and an F1-score of 0.99. Overall, the micro average, macro average, weighted average, and samples average are high, with scores close to 0.98, indicating good generalization for most classes except for minority classes like "Harmless" and "Impact".

For the training set with the same vector size and learning rate, the "Defense Evasion" class achieves a precision of 0.94, recall of 0.99, and an F1-score of 0.97, indicating strong performance. "Discovery" again shows perfect scores (1.00) across all metrics. "Execution" maintains high precision and recall, leading to an F1-score of 0.99. The "Harmless" class has perfect precision (1.00) but a lower recall (0.26), resulting in an F1-score of 0.41, indicating poor recall. "Impact" has perfect precision (1.00) but a lower recall (0.57), resulting in an F1-score of 0.73. "Other" maintains perfect scores (1.00) across all metrics. "Persistence" shows high precision and recall (0.99), leading to an F1-score of 0.99. The overall averages (micro, macro, weighted, and samples) are high, with scores close to 0.99, indicating strong training performance across most classes.

Compared to the previous configurations with a vector size of 100 and learning rates of 0.01, the performance for the "Defense Evasion" class has improved in recall (from 0.90 to 0.96) but decreased in precision (from 0.94 to 0.81), leading to a similar F1-score. "Discovery" maintains perfect performance across all configurations. "Execution" shows consistent performance with slight variations in precision and recall. The "Harmless" class shows worse performance in precision, but there is a slight improvement in recall and F1-Score. "Impact" continues to show issues with recall across all configurations, but precision remains high. "Other" shows improvement across all the metrics. "Persistence" maintains consistently

high performance across all configurations. The overall averages indicate that increasing the vector size to 800 and using a learning rate of 0.01 helps in achieving slightly better generalization and performance stability.

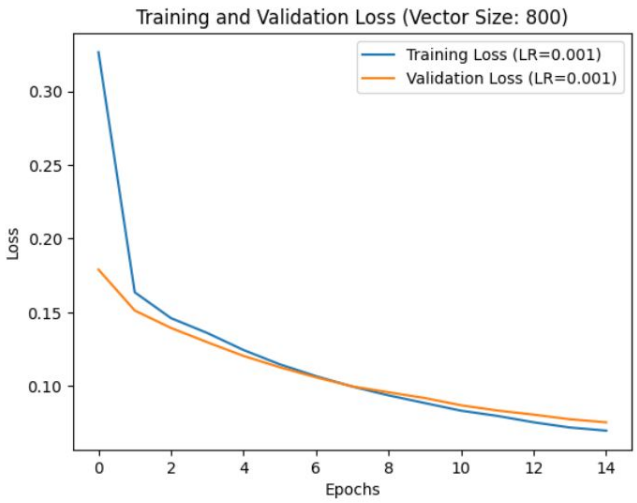

Figure 32: Train and validation loss (vec. size 800, Learning rate 0.001)

The latest graph (Figure 32) shows a more gradual decrease in both training and validation losses, which is characteristic of a lower learning rate like 0.001. However, this graph also shows an increase in loss, suggesting that this lower learning rate might not be effective in this case. The model does not improve significantly and struggles to converge, which indicates that the chosen learning rate is too low.

In contrast, the second graph (LR=0.01) struck a balance between the two, with a smoother decline in loss values and fewer fluctuations, indicating a more stable learning process. The latest graph shows the closest convergence between training and validation losses towards the end of the epochs, suggesting good generalization without significant overfitting. Moreover, it has the least fluctuations, which could be due to the lower learning rate allowing the model to adjust more smoothly to the data.

In conclusion, while the latest graph with the lowest learning rate of 0.001 shows a conservative training approach, it results in an increase in loss, indicating poor model performance. The moderate learning rate of 0.01 appears to be more effective, balancing stability and performance better than both the lowest and highest learning rates.

Table 27: Classification Report on Test Set (vec. size 800, Learning rate 0.001)

Class	Precision	Recall	F1-Score	Support
Defense Evasion	0.90	0.59	0.71	239
Discovery	0.99	1.00	1.00	2908
Execution	0.94	0.90	0.92	1170
Harmless	1.00	0.09	0.16	34
Impact	0.00	0.00	0.00	7
Other	0.00	0.00	0.00	8
Persistence	0.96	1.00	0.98	2645
Micro Avg	0.97	0.96	0.97	7011
Macro Avg	0.69	0.51	0.54	7011
Weighted Avg	0.97	0.96	0.96	7011
Samples Avg	0.97	0.96	0.96	7011

As shown in Table 27, the model's performance on the test set is very similar across all configurations using this learning rate (0.001).

Table 28: Classification Report on Train Set (vec. size 800, Learning rate 0.001)

Class	Precision	Recall	F1-Score	Support
Defense Evasion	0.96	0.64	0.77	239
Discovery	0.99	1.00	1.00	2907
Execution	0.95	0.92	0.94	1170
Harmless	1.00	0.06	0.11	35
Impact	1.00	0.14	0.25	7
Other	0.00	0.00	0.00	9
Persistence	0.96	1.00	0.98	2646
Micro Avg	0.97	0.97	0.97	7013
Macro Avg	0.84	0.54	0.58	7013
Weighted Avg	0.97	0.97	0.97	7013
Samples Avg	0.97	0.97	0.97	7013

In table 28 the model achieves high precision, recall, and F1-score for classes such as "Discovery" and "Persistence". However, for classes like "Defense Evasion" and "Execution," although precision is relatively high, recall is lower. Classes like "Harmless" and "Impact" have low precision, recall, and F1-score, indicating significant misclassifications. The "Other" class shows no correct predictions, with precision, recall, and F1-score all being 0. However, the model's effectiveness varies significantly between different classes, with some classes being well-predicted while others are not.

When comparing this configuration to others (with vector size of 800), it becomes clear that a learning rate of 0.01 performs significantly better across several key metrics. The learning rate of 0.01 achieves a more balanced performance, especially for classes that are challenging for the 0.001 learning rate, such as "Defense Evasion" and "Impact." The higher learning rate helps in faster convergence and better recall for most classes, improving the model's ability to generalize from the training data to unseen data in the test set. The overall accuracy and F1-scores are higher for the learning rate of 0.01, demonstrating its superiority in handling the complexities of the dataset.

5.6 Considerations and Best hyperparameter combination

While analyzing the results, it was noted that some tables showed precision, recall, and F1-score for classes like 'Harmless' and 'Impact' as 0.00 in multiple instances. This could indicate challenging prediction scenarios for these classes or potential issues in model training.

By systematically experimenting with different hyperparameters, including vector size, learning rate, and neural network structure, we optimized the performance of the model. This comprehensive approach ensured that we selected a model that not only performed well on the training data but also generalized effectively to unseen data, providing reliable and robust classification of SSH attack sessions.

The best hyperparameter combination appears to be:

Learning Rate	Vector Size	Epochs
800	0.01	3

We chose 2-3 epochs to prevent overfitting, as indicated by the validation loss trends (Figure 32 in Section 5.5.4). This combination provided the best balance between model stability, generalization, and performance across various classes. The larger vector size of 800 allowed for a more detailed representation of the SSH attack sessions, while the lower learning rate of 0.01 ensured smoother convergence and reduced the risk of overfitting, which is particularly critical for less prevalent threat types, it's worth noting that the loss is also quite small.

From the classification reports for this hyperparameter combination, we could see that the precision was high across all classes, indicating the model's effectiveness in minimizing false positives. This was particularly strong for classes like 'Discovery' and 'Persistence', which consistently showed high performance across various vector sizes but achieved optimal balance here. The recall was also high, signifying the model's efficiency in identifying true positives and minimizing false negatives. This combination improved recall for harder-to-detect classes like 'Defense Evasion' and 'Harmless' compared to smaller vector sizes, showcasing better handling of diverse class distributions. The F1-score showed balanced performance across all classes. The vector size of 800 with a learning rate of 0.01 resulted in the highest F1-scores.

5.6.1 ***Comparative Analysis.*** When comparing vector sizes 100, 300, and 800, vector size 100, while showing high precision and recall for 'Discovery' and 'Persistence', fell short for other classes. The training and validation loss for vector size 100 also exhibited more fluctuations, indicating instability and potential overfitting, especially with higher learning rates.

The vector size 800 provided the richest representation, yielding the best overall performance. It had stable training and validation loss patterns, confirming that this combination effectively balanced fitting and generalization (robust generalization across various attack classes).

6 STEPS PERFORMED TO ACHIEVE THESE RESULTS

Before concluding, we want to summarize all the steps we took to achieve these results. At the beginning, we did not use a predefined vocabulary but instead used a regex to separate the bash commands. For example, we used spaces, hyphens, slashes, and other characters as separators. We chose these separators because, for instance, the command:

```
rm -rf /etc
```

would create a list of commands like this one ["rm", "rf", "etc"]. So far, everything seemed fine, but inside the full session, there were often temporary files or the usage of $ for declaring bash variables. At this point, the list of commands, which had become a list of features, increased significantly.

After using either Bag of Words or Term Frequency-Inverse Document Frequency, we tried to use PCA to reduce the number of features. With three components, the explained variance was higher than 85%. We did not encounter problems in supervised learning, even though using PCA makes interpretability more difficult with fewer components. However, we faced issues with clustering. Apart from the execution times of t-SNE or UMAP, the results were very poor.

At that moment, we analyzed the full sessions with a more discerning eye and realized that many sessions contained parts encoded in base64. Consequently, we wrote a function to decode the encoded parts. In addition, for the reasons explained before, we tried another approach by creating a vocabulary of bash commands. All the commands we included in the file *features.txt* (used as the vocabulary) were selected based on whether they could be used to perform an attack or not. We used https://gtfobins.github.io/ as a reference.

7 CONCLUSION

In this study, different machine-learning techniques were examined, with supervised and unsupervised classification models serving as the primary subjects of evaluation for the set of SSH sessions recorded from a honeypot deployment. As demonstrated, one of the most crucial steps in the assessment of each algorithm is the pre-processing techniques applied to the type of data under study. In this case, both numerical and non-numerical characters had to be cleaned, processed, and decoded for the correct operation of each section.

The supervised methods were employed to assess the performance of the *'Random Forest'* and *'K-Nearest Neighbors'* algorithms in two distinct scenarios: with the default parameters and with the optimal combination identified through the *'GridSearch'* method. The weighted F1-scores were analyzed, and the effects of varying algorithm parameters on the models were examined. The results demonstrated that the "K-Nearest Neighbors" algorithm exhibited greater consistency across all evaluated metrics and classes. However, given that some classes have a reduced number of samples, it is recommended that future studies separate these classes from the main dataset. This allows for the evaluation of the supervision

algorithms separately for classes with higher and lower numbers of samples, while ensuring that the performance metrics of each model are not affected, avoiding overfitting.

In the **unsupervised learning** chapter, we explored the GMM and K-means algorithms to classify the dataset into different clusters. The results are quite surprising, indeed the clusters more or less reflect the separation of intents performed by MITRA, and this was achieved without any label training, solely by applying mathematical methods to discover common patterns among the data points.

Even more impressively, some classifications differed from the MITRA results, and upon further analysis, some of these classifications were found to be correct and even provided a finer-grained classification, highlighting specific subsets of attacks that were distinct from the rest.

Both GMM and K-means clustering were addressed in this chapter, and it's challenging to determine which performed better. The metric values are similar, and both results are valid from an analytical perspective. As noted during the discussion, each algorithm excels in certain aspects and falls short in others.

The final section of our study explored the application of Doc2Vec and neural networks to classify SSH attack sessions. The experiments revealed that varying the learning rates and vector sizes significantly impacted the model's performance. The moderate learning rate of 0.01, when paired with a larger vector size of 800, emerged as the most effective configuration, providing a balanced approach to stability, performance, and generalization. This configuration proved crucial for the accurate detection and classification of SSH attack sessions (Section 5.6).

The neural network configuration, guided by the formula 1, ensured that the network could handle complex data patterns while maintaining efficiency. The use of ReLU activation functions in the hidden layers and the sigmoid activation function in the output layer was essential for introducing non-linearity and enabling multi-label classification tasks, respectively. These configurations facilitated the network's ability to learn and generalize from the data effectively.

Overall, this project was a great opportunity, full of fascinating aspects to understand and experiment with. It underscored the importance of complementing theoretical lessons with strong practical applications to achieve significant results. Practical experimentation not only reinforces theoretical knowledge but also provides deeper insights and a better understanding of real-world challenges and solutions. The combination of theory and hands-on practice proved to be immensely valuable in this study, leading to meaningful and impactful outcomes in the realm of SSH attack detection and classification.

REFERENCES

[1] Asmita. 2017. Different Data: A Comparative Study. *Journal or Conference Name (if applicable)* 12, No.8 (2017), 10. https://minerva-access.unimelb.edu.au/bitstream/handle/11343/216910/2017_Asmita_Different_Data.pdf

[2] Zach Bobbitt. 2022. Classification Report in sklearn. (2022). https://www.statology.org/sklearn-classification-report/ Accessed: 2024-06-01.

[3] Michal Aibin Gabriele De Luca. 2024. Neural Network Architecture: Criteria for Choosing the Number and Size of Hidden Layers. (2024). https://www.baeldung.com/cs/neural-networks-hidden-layers-criteria Accessed: 2024-06-04.

[4] Sandhya Krishnan. 2021. How do determine the number of layers and neurons in the hidden layer? (2021). https://medium.com/geekculture/introduction-to-neural-network-2f8b8221fbd3 Accessed: 2024-06-04.

[5] Semrush. 2024. TF-IDF: What It Is How to Use It for SEO. (2024). https://www.semrush.com/blog/tf-idf/