POLITECNICO DI TORINO

Master's degree course in Computer Engineering

Master's degree Thesis

User Behaviour Classification by means of Unsupervised Learning optimized by DNN

Internal Supervisor

Prof. Fulvio Risso

External Supervisor

Dr. Ivan Romero

Candidate

Alessandro Bonifazi

DECEMBER 2019

Summary

Abstract

In the last years, Machine Learning has been enjoying a novel surge of use in applications and problems from different domains, leveraging its power and allowing automation in various situations.

This is for sure because of the big increase in data availability, in better computational power and in the improvement of Machine Learning techniques.

There are three main learning paradigms in Machine Learning which influence data collection and feature engineering: supervised learning, unsupervised learning and reinforcement learning.

Supervised learning uses labelled training datasets to create models, so a method to label the data and find the so-called ground truth value it is needed. To the contrary unsupervised learning exploits unlabelled datasets, usually to create models that can discriminate between patterns in the data. Reinforcement learning is an agent-based process for modelling problems for decision making, in order to maximize a reward in a situation.

The three biggest categories of problems that can exploit Machine Learning are classification, regression and clustering. The goal of classification is to map from a series of input data to a limited and smaller set of discrete output values like mapping a mugshot to the name of the person in the photo. Regression problems are like classification ones but with the difference that the output is in the continuous domain and therefore can take the any form within a range of real values. These two kinds of problems usually expect labelled data and therefor are best suited for supervised learning.

The aim of clustering is to find a structure in the data, grouping similar objects together while increasing the gap between groups. Very often this kind of problem is faced by means of unsupervised learning.

Computer networking can also benefit from this technology, as network problems can fall in one of these subcategories that can leverage Machine Learning:

- An example is traffic classification, the categorization of network traffic into appropriate classes, which has been applied to a wide range of applications from quality of service (QoS) provisioning in internet service providers (ISP) to security related applications in firewalls and intrusion detection systems.
- A forecast of a future event, like a machine failure or the users' demand can be formulated as a regression problem, making possible for instance, to obtain a real-time traffic prediction for dynamic resource management.
- Clustering problems usually concern outlier detections through pattern recognition, such as identifying anomalous traffic and suspicious activities in the network.

The aim is to study how the users exchange information (and which kind of information) through the network and find out a solid methodology in order to cluster local area network (LAN) users that share similar behaviour, allowing to exploit this discovery for resource management and anomalous activities identification. The main challenges of a developer that is going to face a problem of this kind are:

- which dataset should be used to train the model; which are the most meaningful data features that needs to be extracted and how they should be represented;
- which is the best machine learning approach to follow and finally which is the correct algorithm to exploit.

This work starts with capturing traffic in the interested LAN; the exploitation of a dataset obtained in this way, instead of relying on a public one, allows to make use of data that models perfectly the real behaviours of the LAN hosts.

After the data cleaning phase, from the network flows are extracted the relevant features that are used to train the model. In this project two different set of features are analysed: the former is based on the flow IP destination distribution for each host while the latter exploits the per-user distribution of the kind of service being carried by the flow. In particular, the users IP destination distributions are represented in a peculiar matrix format.

These different features are used to train different times a novel clustering model that was developed for the sake of this project.

This unsupervised model starts with the aid of a deep neural network which is trained in a supervised manner in order to minimize a target function that symbolize the degree of users' similarity. After the neural network training, and the target function convergence, it is possible to build a confusion matrix, which represents the distributions of the neural network predictions for each host.

This confusion matrix is exploited to cluster together users that have been forecasted to have a similar prediction distribution; the similarity metric being used is very simple and consists in the difference between two users' prediction distributions.

What is obtained from this analysis is a hierarchical clustering model, with a peculiar dendrogram, where a similarity threshold drives the criterion for a user to be considered inside or outside a cluster.

What emerges from this study is that the clustering results between the IP destinations case and the service categories case have remarkable differences. The clustering configurations show how it is possible to predict the user's demand and how this demand is shared among them, allowing to inspect it in different levels of details.

The fact that the IP-based and the service-based clustering show different results gives the opportunity to obtain information from different perspectives. Instead, if the wish is to consider them both at the same time, it is possible to look for the most similar clustering configuration between the two results.

1. Introduction

The context of this project involves traffic classification delimited by the boundaries of a local area network. Traffic classification is important to many applications, such as network resource management and pricing, quality of service control, intrusion detection.

As briefly mentioned in the abstract, in order to solve a classification problem, the training data needs to be labelled, as it falls in supervised learning domain. For instance, if we are interested in building a classifier that is able to recognize from a network flow the service being carried, each flow in the training dataset must be paired with its service; in this way, after the model has been trained, the classifier will be able to identify which are the services being borne by unknown flows.

Is it clear that establishing the ground-truth values for data labels is a very important and delicate phase because just a mistake can compromise the goodness of the resulting classifier.

The aim of the project, hence, is to exploit Machine Learning, more precisely unsupervised learning, in order to find patterns and structure in the dataset that will allow also to find ground-truth values for the classifying phase.

Going deeper into details, the interest is in the user behaviour inside the LAN. The goal is to group users that generate similar traffic in the same cluster allowing further insights analysing the obtained results, in addition to, for example, detect anomalous behaviours and use the groups as ground-truth values for classification tasks.

Going with this approach, finding a method for labelling the dataset it is no more needed. Finding a ground-truth criterion is not as simple as assigning a class to a user because his job involves scientific research or because he is a data analyst or a programmer.

The aim is to study how the users exchange information (and which kind of information) through the network and find out a solid methodology in order to cluster together similar behaviours, making possible to analyse the results to figure out which kind of traffic and behaviour corresponds to each group and why certain users are considered to be the same type of users.

It is possible to exploit this kind of information in various ways, for example it may happen that all the users belong to one out of four big clusters while one of them is grouped alone; this means that the traffic relative to that user is substantially different from the others and a further analysis can account this dissimilarity.

Also, as mentioned before, it is possible to use the cluster membership as the ground-truth value for some supervised learning task.

For instance, if we are interested in resource management, we would like to share the network resources based on the kind of traffic a user is generating; after obtaining the results of the clustering phase we can train a supervised model, pairing each user traffic with the label pointing to the group it belongs. Following this path, we will obtain a network traffic classifier that will be able to map in real time the traffic given as input to the respective group, which is the result of its prediction.

Like in every machine learning process, in order to achieve these results and build such a model, a big quantity of data is needed. In this case the data collecting phase is about sniffing in the internal interface of the LAN's edge router.

After that phase it is important to clean the data and go through the feature engineering phase, one of the most crucial part of the model's design, which consist in using domain knowledge of the data in order to create features to make the machine learning algorithm work.

During this step, the data is studied and analysed in order to comprehend which properties of the data flow are most relevant for the current task; should the machine learning algorithm be trained taking into account the IP addresses of the two peers? The average packet size of the flow? Its duration? After this analysis the algorithm can finally be trained using the selected features.

The following chapters are organized as following:

- Chapter 2, State of the art: will be presented how clustering works and its state-of-the-art technologies, also regarding network traffic unsupervised learning.
- Chapter 3, System Architecture: this chapter is split into three parts and explains how the built model works.
    - 3.1, Data collecting
    - 3.2, Features Engineering
    - 3.3, Model
- Chapter 4, Implementation: here will be shown the actual implementation of the logic discussed in chapter 3.
- Chapter 5, Results: in this chapter it is exposed what kind of results were obtained and how they interact with the state-of-the-art.
- Chapter 6, Conclusions: summary of what has been obtained and inspirations for future works.

2. State of the Art

As referenced in the introduction, clustering tries to find different groups within elements in data.

Clustering is included in unsupervised machine learning as a process which segments dataset by some shared attributes.

The goal of clustering is to find a natural grouping in data such that all the elements belonging to the same cluster are more similar to each other than those from different groups.

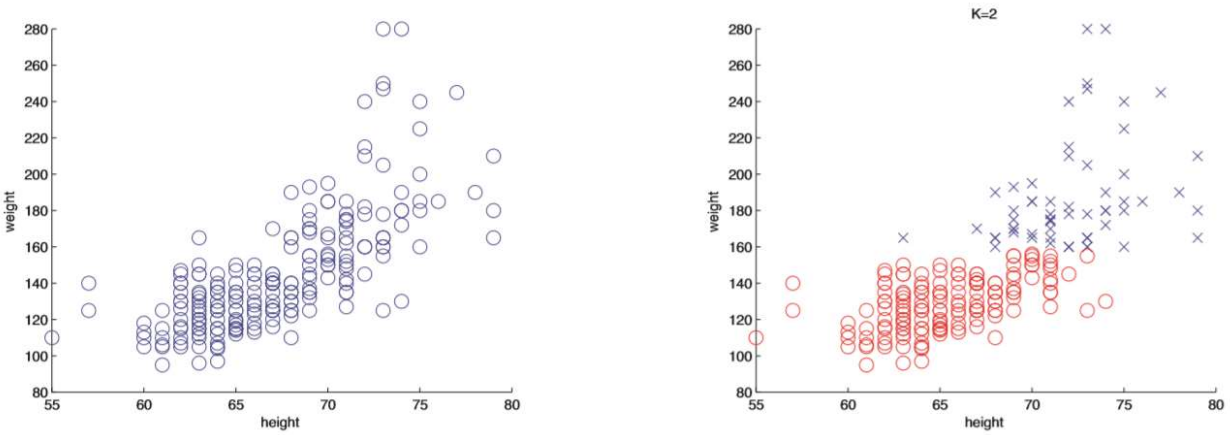

Figure 1 - On the left the 2D plot of a dataset, on the right a possible clustering outcome with 2 clusters.

The images above show a 2-dimensional plot, representing heights and weights of a group of 280 people. The plot on the left is the original data while the one on the right show a possible clustering result using a number of clusters K = 2. It can be noticed, however, that there might be various clusters although it is not clear how many. The outcome on the right is just one of the possible results of a clustering analysis exploiting one of the existing algorithms and its parameters; considering a higher number of clusters, for example, would have led to a different configuration.

There are various distinctions that are possible to find while considering different clustering approach; the biggest discrimination among these algorithms is definitely the concept of hard clustering and soft clustering.

A hard clustering algorithm constrains each element of the dataset to be assigned only to one cluster; it is impossible for the same object to belong to two different groups, like in the example shown before.

Soft clustering, also known as fuzzy clustering, does not impose the limitation on the number of groups an element should be part of; sometimes we do not need a binary answer and following this approach an object can be assigned to more than one group. Some soft clustering algorithms, for example, assign a probability distribution to each element, expressing the chances that that same element belongs to each one of the detected clusters.

Most of the algorithms falls in the hard clustering domain, due to the fact that most of the problems can be formulated through it and do not need soft clustering properties.

2.1.1. Centroid-based Clustering

Centroid-based clustering, also known as partitional clustering, learns through the generation of various partitions in the dataset and then evaluates them by some criterion.

In this kind of clustering, a cluster is represented by a vector, the centroid, which may be not necessarily a member of the dataset and it is called centroid. Different algorithms have different ways to calculate and detect each cluster centroid.

Each element of the dataset is assigned to the cluster whom centroid is the closest to. It is clear, hence, that a similarity metric (that allows us to define what “close” means) must be chosen.

The most famous algorithm of this family is K-Means.

The K-Means algorithm uses iterative refinement to produce the final result; giving as input the desired number of clusters K and the dataset, the algorithm starts with an initial estimate for the K centroids, which can be either selected from the samples or randomly chosen.

The base version of the algorithm is composed of two steps:

1. Data assignment step: in this step each sample is assigned to the nearest centroid, based on the squared Euclidean distance.
2. Centroids update step: in this step the clusters' centroids are recomputed by taking the mean of all the data points assigned to that cluster.

K-Means, which is guaranteed to converge, iterates between these two steps until a stopping criterion is met (for example samples are no more changing clusters in the first step).

It is important to notice that the obtained result may not be the best possible outcome because of the randomicity in the starting centroids selection.

In figure 2 different moments of K-Means execution are shown:

a) The plot shows the dataset samples in a 2-dimensional plane.
b) K = 2, the two centroids are randomly chosen.
c) Data assignment step.
d) Centroid update step.
e) Data assignment step (based on the new centroids position from the previous step).
f) Centroid update step and convergence of the algorithm (in the next data assignment step the samples membership will not change).

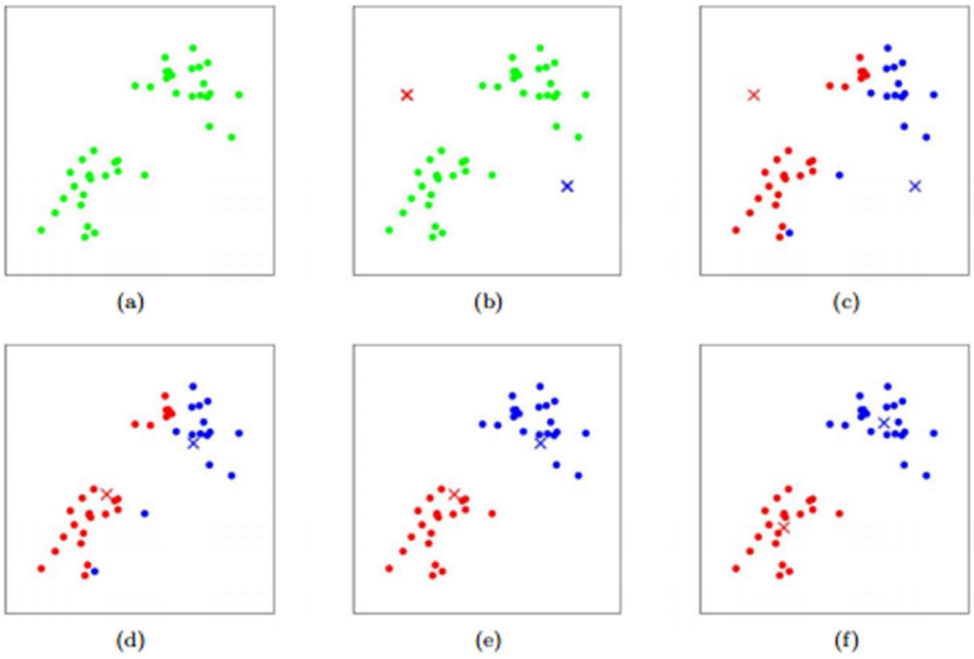

Figure 2 - K-Means algorithm steps snapshot.

Two big limitations of this algorithm are the initial selection of the number of clusters and the random centroids initialization that can bring to a poor result. While for the second problem different variations of the base algorithm were developed, like K-Means++, in order to improve the centroids selection, the first one remains.

It is hence required to the analyst to evaluate different clustering results through different K values; the mean distance of a sample to its cluster centroid it is used as a metric for the goodness of the result.

This metric as a function of K is plotted and the “elbow point”, where the rate of the metric decrease sharply shifts, can be used to roughly determine K. A number of other techniques exist for validating K, including cross-validation and the silhouette method.

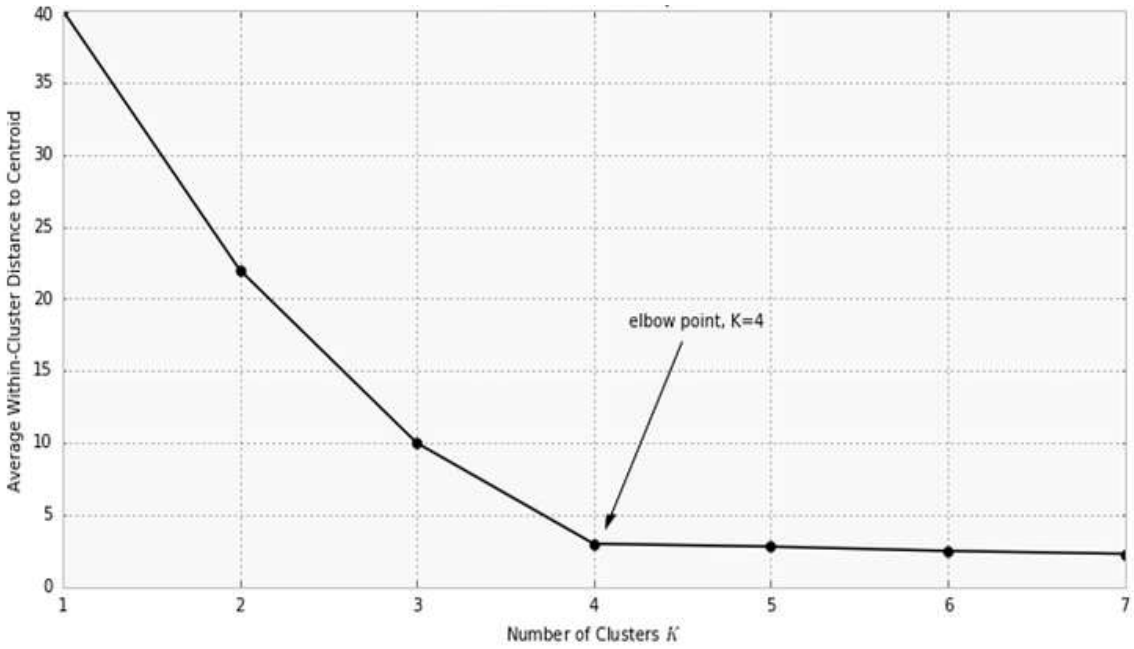

Figure 3 - Elbow point example.

Another key limitation of K-Means is its cluster model. The concept is based on spherical clusters that are separable so that the mean converges towards the cluster centre; since data is split halfway between cluster means, this can lead to suboptimal splits, like in figure 4 where the dataset structure is not spherical. It works well on some datasets and fails on others.

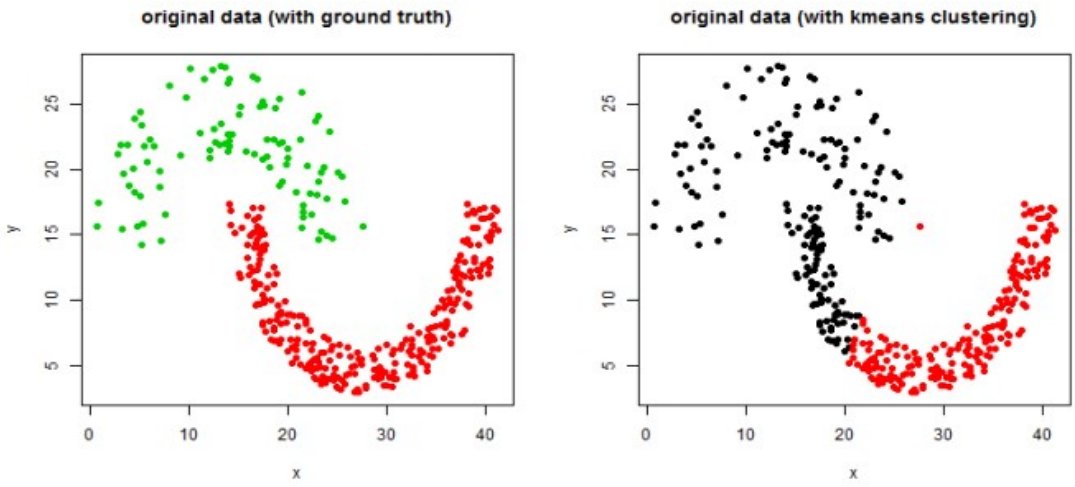

Figure 4 - A suboptimal K-Means split.

A K-Means variation that is worth mentioning is Fuzzy C-Means which is its soft clustering version where each data point has a fuzzy degree of belonging to each cluster.

2.1.2. Hierarchical Clustering

This method of cluster analysis seeks to build a hierarchy of clusters and is based on the idea that objects are more related to nearby objects than farther away ones; there are two strategies that can be followed to achieve this result.

The agglomerative strategy, also called bottom-up, starts with each object in its own cluster and recursively merge together the most similar one, up to having only one group with all the elements.

The divisive strategy, or top-down, is the opposite: it starts with all the elements in the same cluster and then recursively split them down until each object ends up in the same cluster.

In both cases the result of hierarchical clustering is usually presented in a dendrogram.

A dendrogram is a diagram that shows the hierarchical relationship between objects. Its main use is to work out the best way to allocate elements to clusters. The key to interpret such graph is to focus on the height at which two objects are joined together; in figure 5 we can see that E and F are most similar, as the height of the link that joins them is the smallest. The next two most similar are A and B.

An important thing to remember is that a dendrogram cannot tell you how many clusters you should consider: again, in figure 5, the (incorrect) interpretation is that the dendrogram shows that there are two clusters.

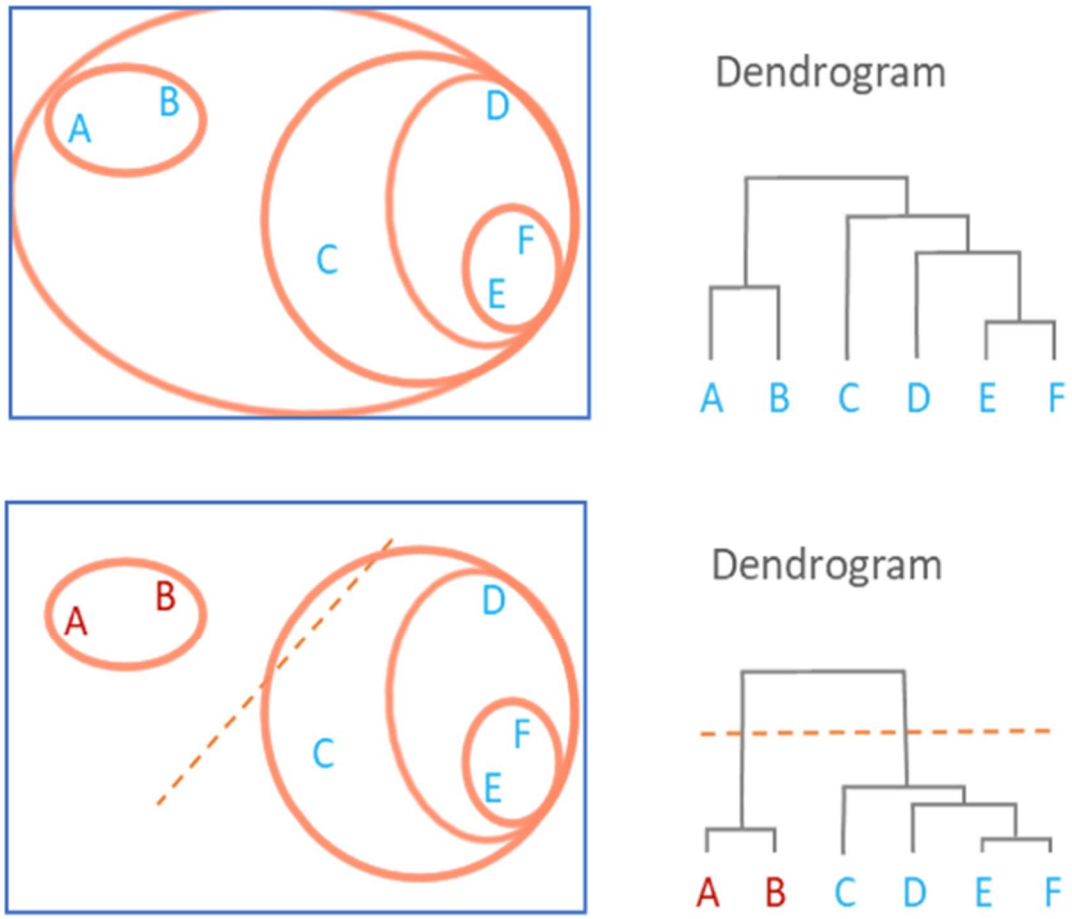

Figure 5 - On the top the dendrogram resulting from a hierarchical clustering. On the bottom one of its interpretations.

An example of hierarchical clustering algorithm is the single-linkage clustering, which follows the agglomerative strategy.

At each step, the two clusters separated by the shortest distance are combined. The definition of shortest distance is what differentiates between the different agglomerative clustering methods.

In single-linkage clustering the distance between two clusters is the distance between their two nearest elements; instead, in complete-linkage clustering the distance between two clusters is the distance between their two farthest elements.

This underlines the importance of defining a cluster dissimilarity metrics when using a hierarchical clustering algorithm.

One advantage of hierarchical clustering is that the analyst has not to make assumptions on the number of clusters, like in partitional clustering algorithm; instead he has to infer the best result by means of the dendrogram.

2.1.3. Density-based Clustering

In density-based clustering the aim is to merge together elements that are defined in areas with higher density. Elements far away in sparse areas, which would be clustered alone, are usually considered to be noise.

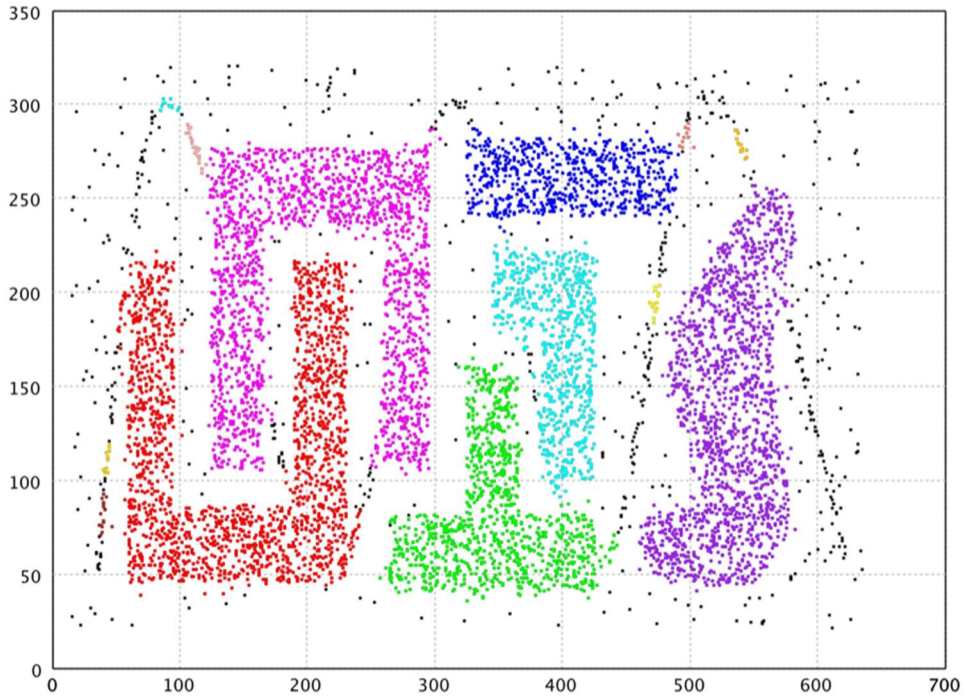

Figure 6 - Result of a density-based clustering. Different colours identify different clusters.

Clusters detected by this kind of algorithms are dense areas in the dataset space, separated from each other by sparser areas.

Density-based algorithms detect themselves the ideal number of clusters, so it is not needed for the analyst to provide this value as an input. However, they usually require a pair of parameters in order to work properly: the maximum radius of a data point neighbourhood (ε) and the minimum number of data points in another data point neighbourhood (minPts).

DBSCAN, the most famous density-based clustering algorithm classifies each data sample in one of the three following class:

- *Core point*, if there are at least minPts points at a distance of ε.
- *Border point,* if it has less than minPts in its neighbourhood ε but it is in a core point neighbourhood.
- *Noise point*, if it is not a border nor a core point.

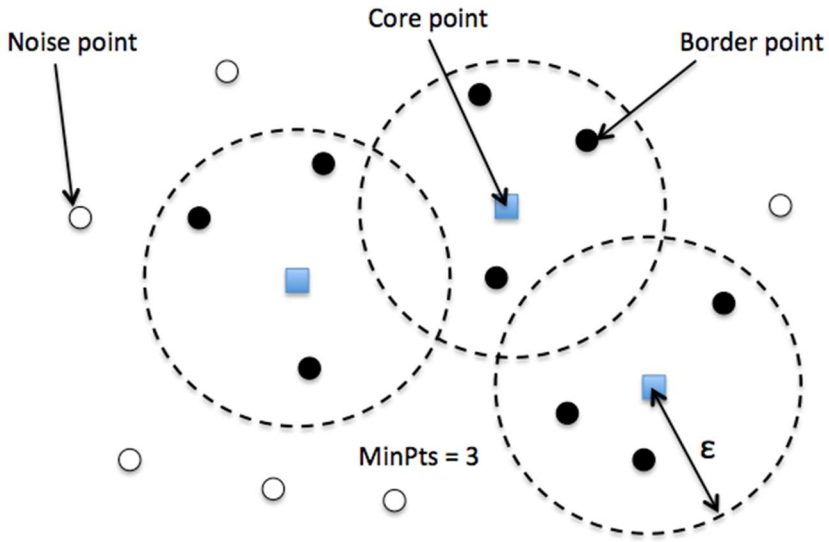

Figure 7 - Core, noise and border points.

All the data points inside the point p's neighbourhood are said to be *directly reachable* from p (with respect to the selected minPts and ε).

A point p is *density-reachable* from a point q if there is a chain of points p_1, ..., p_n with $p_1 = q$ and $p_n = p$ such that p_{i+1} is directly reachable from p_i.

The steps to the DBSCAN algorithm are:

1. Pick a random data sample not yet processed and it not classified as outlier (noise point).
2. Compute its neighbourhood to determine if it's a core point. If yes, start a cluster around it, otherwise classify that point as a noise point and return at point 1.
3. Expand the cluster created in the last step by adding all the directly reachable point from the core point that created the cluster. If other neighbourhoods exist find all density-reachable points and add them to the cluster. If an outlier is added in this phase, convert it to a border point.
4. Repeat until all samples are or inside a cluster or labelled as noise points.

The biggest advantage of DBSCAN and generally all the density-based algorithms is that they can find arbitrarily shaped and non-linearly separable clusters; also, differently to the algorithms presented so far it knows how to recognize points of noise and hence it is robust to outliers.

One DBSCAN problem, however, is that depending on the order points are processed, some border points can end up being assigned to one cluster or to another one (if for example are reachable from both neighbourhoods). This makes this algorithm not entirely deterministic. However, this is usually not a big deal because DBSCAN is always deterministic in identifying core and noise points.

2.2. User Traffic Clustering

The previously introduced algorithms and techniques are vastly exploited by the research community in the networking area. In this section will follow a brief overview of some recent works on network and Internet user behaviour traffic clustering.

Leng *et al.* [1] proposed a method to analyse users' internet browsing and preferences in a campus. To apply clustering algorithms, the authors represented each user with a profile vector through user behaviour modelling. They based their model on the topic model (which is usually used for document classification) in order to generate an original profile matrix. They also exploit TF-IDF (term frequency – inverse document frequency) and LSA (latent semantic analysis) for dimensionality reduction in order to prepare the data they have in the best possible way before applying any clustering solution. The authors finally extract users' interests by clustering users with similar browsing habits into groups by means of K-Means++. They also examine their results with demographical information (such as age and gender) to help them explain their findings accordingly and to prove that their algorithm works efficiently.

De Oliveira *et al.* [2] exploited cluster analysis in order to partition Internet users based on their hourly traffic utilization. The authors resorted to two data traces measured in two different ISPs, using Principal Component Analysis (PCA) as an exploratory tool. They used a hierarchical clustering approach obtaining a dendrogram for each ISP trace, which led the authors to identify three different clusters that were interpreted as:

- Users with high transfer rate in all periods of the day.
- Low/high transfer rate in the morning/afternoon.
- Low transfer rate in all periods of the day.

The experimental results on different datasets convey that the method can obtain up to 80% overall accuracy, and, after a log transformation, the accuracy is improved to 90% or more.

Nogueira *et al.* [3], exploiting the results obtained in [2], compare the use of Discriminant Analysis and artificial Neural Networks for the classification of the Internet users. The classification is based on a predefined set of clusters (the ones detected in [2]) which, in the first case, is used to define the function that best discriminates among clusters and, in the second case, is used to train the neural network. Their findings indicate that for the dataset in examination Discriminant Analysis outperforms their conventional feed-forward three layers Neural Network; also, they state that Neural Network results have a more complex interpretation than the Discriminant Analysis ones.

Kumpulainen *et al.* [4] show how multi-layer clustering can be used to monitor not only the patterns in the network, but also to characterize servers and devices that are generating the traffic. The authors did not used any Deep Packet Inspection (DPI) techniques, relying only on IP packet header information. The multi-layer clustering they proposed allows them to divide the data into two groups of low and high levels of traffic, in the first phase; in the second phase these two groups are scaled and clustered separately to form a number of behavioural traffic patterns describing typical hourly behaviour. Behavioural profiles of the IP addresses are formed by studying how the traffic generated by each address is distributed between the traffic patterns. The IP addresses are combined into groups of similar behaviour in the third clustering phase using their proportions spent in each traffic pattern.

Gu *et al.* [5] proposed a network anomaly-based botnet detection system that is independent of the protocol and structure used by botnets. Their work has foundation on the essential definition and properties of botnets (bots within the same botnet will exhibit similar communication patterns and similar malicious activities patterns). The authors perform clustering analysis on two different logs written by two different network monitors:

- The *C-plane* monitor is responsible for logging all the network flows in a convenient way.
- The *A-plane* monitor is responsible for detecting suspicious activities.

The authors extract a number of features from the raw logs and apply clustering algorithm in order to find groups of machines that show very similar communication (*C-plane*) and activity (*A-plane*) patterns. Finally, they define a cross-plane correlator that combines the two clustering results. The C-plane clustering works on flows represented as a vector c_i with the following features:

- *Number of flows per hour*: number of TCP/IP flows in c_i that are present in each hour out of the total analysis time.
- *Number of packets per flow*.
- *Average number of bytes per packets.*
- *Average number of bytes per second transferred by the flow.*

After having extracted these features for each flow in the *C-plane* log, they exploit them to perform a two-steps clustering (the first more coarse-grained, the second more refined) through X-Means algorithm. X-Means is a K-Means variant that does not require the user to choose in advance the number K of clusters (it runs internally K-Means multiple times and perform an efficient clustering validation to detect the best number of clusters). The *A-plane* clustering behaves in a similar way, splitting the clustering analysis in two steps: at first the authors cluster the clients according to the type of their malicious activity and then they cluster them again differently according to the specific activity features.

Once obtained the clustering results, they perform the cross-plane correlation, checking the clusters in the two planes to find out intersections that reinforce evidence of a host being part of a botnet. In order to do this, they compute a botnet score for each host on which they have witnessed at least one kind of suspicious activity and they filter out all the hosts below a certain threshold score.

Finally, this process, groups the remaining most suspicious clients according to a similarity metric that considers both C-plane and A-plane clusters these hosts have in common.

3. System Architecture

What is accounted by this project it is an ad-hoc solution for a specific well-known local area network. The objective is not to train a machine learning algorithm on a generic dataset and, once a good degree of accuracy is obtained, sell the product as a good model for any network. The system is thought as a custom-made model for a specific network populated by specific hosts.

Most of the research works on this topic (like the ones described in section 2.2) exploit public datasets in order to train their algorithm. This system plans to use as training set the effective LAN traffic by means, for example, of a software that is able to sniff the packets that transit on the edge router.

Using as dataset the real traffic from the interested LAN, in this case, is a good path to follow because the collected data models better the users' behaviour in the network. A disadvantage of this approach is that of course the data gathering phase is more complex and time consuming.

For this kind of problem, where the objective is a specific network, it is absolutely not advisable training the model on a dataset and then exploit that same model on a LAN different from the one that is the source of the training dataset.

An example of the pipeline to build the final model is:

- Collect the traffic inside the network for a given period of time (or use past traffic captures but incurring in the risk of them being too old).
- Train the deep neural network in order to minimize the similarity between users' MAC addresses (more information on section 3.3).
- Get the resulting confusion matrix and obtain clustering results (more information again on 3.3)
- Choose the most suitable clustering configuration.

At the end of the clustering process what will be obtained is, for each host (identified by a MAC address), a label that symbolises the membership to a cluster. This label can be exploited hereafter, for instance, as a ground-truth value for an online classifier placed on the edge router.

Hence a “classifier” module can be laid in pipe after a traffic sniffer module; this last piece of software should analyse all the traffic crossing the edge router and, for each Ethernet frame that is passing through, extract its MAC address along with, as a consequence, its cluster membership.

One thing to be aware of is the validity of the model; it is worth noticing that the data used as a training set may not be forever valid. One simple case that can mine the consistency of the system in the time is the addition of a new host in the LAN; of course it will not invalidate the other MAC address – cluster couples but for sure the model will not be able to cluster it.

Another question that may rise is about the time-constancy of the user behaviour. Can we consider a user behaviour constant in time? Clearly it changes during the time span of a day (for example during the launch break), but how this behaviour changes on longer periods? After three months from the days of captures, the collected traffic will be as valid as at the beginning?

3.1. Data Collecting

The dataset was obtained sniffing on the edge router (from the interface that is on the side of the private network) of the target LAN. The aim is to typify that LAN's users, therefore the data gathered in this way will model better the network's hosts.

The main software tools exploited in this phase in order to capture the traffic and elaborate the resulting information are tcpdump, tshark and Python's libraries numpy and pandas.

The capture sessions lasted approximately 8 hours every day, from 9.00 until 17.00 with some variability. For an ad-hoc analysis it is advised to start and stop the daily sniffing sessions always at the same moment and for a longer period of time (the best solution would be to stop capturing when the last host leaves the place).

Tcpdump was used on the edge router with the task of sniffing the traffic in transit on the LAN's side interface. Ethernet captured frames from this process are saved in a pcap file.

Tcpdump's parameter -s 80 was used and allowed to capture just the first 80 bytes of each frame, ignoring all the followings; 80 bytes are enough in order to get information from the IP protocol header and TCP/UDP protocols header. There is no interest in the TCP/UDP payload because most of the time it is encrypted and hence useless for this kind of analysis.

Every time that approximately 200MB of traffic are captured (the accurate amount is 200 million bytes), the information is written in a new pcap file, thanks to the parameter -C 200. In the end, for each day of capture, will be present N pcap files, which N-1 of them will be 200MB big.

After having obtained pcap files, the captured traffic information is extracted in a csv file by means of a network tool called tshark.

During this step the bytes inside the pcap file (given as input to tshark) are read and, for each frame, a new row is inserted in the csv file, represented as a series of the current Ethernet frame's attributes. These properties are:

- Ethernet level properties: frame.number, frame.time, frame.time_epoch, frame.len, eth.src, eth.dst.
- IP level properties: ip.len, ip.src, ip.dst, ip.proto.
- TCP and UDP level properties: tcp.srcport, tcp.dstport, tcp.stream, tcp.len, tcp.seq, tcp.connection.syn, tcp.connection.sack, tcp.ack, tcp.connection.fin, tcp.connection.rst, tcp.window_size, tcp.analysis.bytes_in_flight, tcp.analysis.ack_rtt, udp.srcport, udp.dstport, udp.stream.

For an accurate insight on their meaning visit the *Wireshark Display Filter Reference* [6]. Not all of these attributes will be meaningful for the following analysis but, in order to avoid repeating the conversion from pcap to csv in the future due to a missing attribute, they were all saved in the csv file.

Then, other information is extracted from the pcap file, thanks to the software nDPI which allows to know which is the type of service and which is the application carried by each data flow.

nDPI allows application-layer detection of protocols, regardless of the port being used. This means that it is possible to both detect known protocols on non-standard ports (e.g. detect http non ports other than 80), and also the opposite (e.g. detect Skype traffic on port 80). This is because nowadays the concept of port = application no longer holds.

Hence, a new csv file is created with each row representing a flow and each column being the flow's *5tuple* (IP address source, IP address destination, transport layer protocol, transport layer port source and transport layer port destination) and three new attributes obtained by means of nDPI:

- service: Identifies the last protocol being used by the flow; examples of the values found are SSL, HTTP, ICMP and QUIC.
- category: Identifies the typology of service offered by the flow, such as Web, Mail, File Transfer and Streaming.
- app: Tries to identify the application (or group of applications) carried by the flow, like Netflix, Telegram, Spotify and Google.

After this step we own two csv file: the first one with a list of all the Ethernet frames captured on the interface (with their properties) and the second one with the service, category and app information for each flow.

Since the analysis will be flow-based, the list of sniffed frames needs to be collapsed into the flows they belong to. Thanks to pandas library all the frames that share the same 5tuple was merged into a row of a new csv file; this new file will hence contain all the flows present in that 200MB pcap.

The flows are detected executing two *GROUPBY*s in waterfall to all the file's frames:

- The first *GROUPBY* is done on the columns [ip.proto, ip.src+ip.dst, port.src+port.dst].
- The second *GROUPBY* is done on the *5tuple* columns.

The first one is exploited to group together all the frames belonging to the same flow, regardless of the travel direction of the frame. It may happen that frames belonging to different flows will fall in this group; this is not a problem, because the second *GROUPBY* will be able to distinguish them.

The second *GROUPBY* in fact, is executed on each group detected by the first one, merging all the frames that share the same *5tuple*. This allows to separate and identify those rare cases where two flows have the same ip.src+ip.dst and port.src+port.dst; this second *GROUPBY* enables also to verify the correctness of the result checking for the presence of 2 groups for each TCP flow (one for each direction) and one group for each UDP flow.

For more details on the algorithm, check the implementation in section 4.1.

After the flow detection, each row of this new csv will be populated with the following flow's properties:

- Flow's *5tuple*: ip.proto, ip.src, ip.dst, port.src, port.dst.
- macsrc: MAC address of the LAN's host.
- time: time of the first frame of this flow.
- totsize: flow's size in bytes, obtained summing all its frames' size.
- avgsize: average frame size for this flow.
- duration: flow duration (last frame time subtracted by the first frame time).
- rtt: minimum round-trip time obtained by the TCP ACK.
- inlan: number of bytes that entered in the LAN from this flow.
- interframe: average time the host waits in sending two following frames of this flow.

An important step that is done after the creation of this flows' csv is adding to it the information on the services obtained thanks to nDPI. Hence, the data from the nDPI csv was merged into this last file through a *JOIN* operation on the flows' *5tuple*. In this way the flows' csv is enriched with the columns service, category and app.

In addition, a new csv that groups all the flows by the host that generated them is created. From the csv that list all the Ethernet frames are done two *GROUPBY*s, first on the ip.src attribute and then on the *5tuple* properties. The result is a csv that contains the list of flows ordered by hosts; the columns are the flows' *5tuple* and the host MAC address.

This whole pipeline is done for each pcap file obtained capturing on the edge router.

Finally, all the flows belonging to the same day (and coming also from different flows' csv) are collected and merged in a new and final csv file. Of course, this procedure takes into account the elimination of duplicate flows that are present in more than one file and the recalculation of interesting fields like duration, totsize and avgsize. Again, for the actual implementation of this step, check section 4.1.

The same procedure is applied for the csv files that group flows by the hosts who generated them.

After having collected in the previously explained format all the flows captured in 25 days, it is time to move to the feature engineering phase, where we will take the most important flows' fields in order to exploit them as features for the Machine Learning algorithm.

3.2. Feature Engineering

The previous phase's objective was cleaning the raw `pcap` capture files obtained sniffing on the edge router and converting them into a list of flows, along with all their interesting parameters: it was chosen to save these flows in a `csv` format, a file for each day of capture.

The so-obtained flows will be exploited to train the Machine Learning algorithm; in particular, as better explained in the following section 3.3, a Deep Neural Network will be trained in order to map each flow to the MAC of the host who generated it.

This means that the Network will ask as input training set a list of flows (or, more precisely, a list of features that represent the flows) paired each one with its source MAC address.

The output will be a MAC address and the Network will be trained in order to make the predicted MAC equal to the one given as input (along with the flow).

The MAC address was chosen as label because it is the most faithful and reliable way to represent a LAN's host; it is important to take into consideration in fact, that a host's IP address may not always be static because, very often, inside a private network, is configured a DHCP server which is in charge to dynamically assign IP addresses to the hosts.

It is worth noticing that a MAC address cannot be mapped one by one to a physical user, but it is correlated in an unambiguous way to a device with a network card connected to the edge router. For instance, some hosts detected by this mechanism may be:

- user A's laptop,
- user A's smartphone,
- a network connected printer,
- the edge router itself (its internal interface).

The flows from the previous phase are represented through a series of properties (the ones listed before). It is not feasible training the Deep Neural Network giving as input flows in this kind of representation.

It is true that introducing in the system all the flows' properties we are giving more information to the Network, but it is also true that not all the information we have is relevant for this kind of problem.

This is a classic phenomenon that arises when analysing data in high-dimensional spaces called *Curse of Dimensionality*. When you have a lot of input dimensions (the flows' features) the problem becomes computationally expensive and difficult to solve, needing more and more data to obtain a representative training sample in order to be able to generalise to unseen data.

Hence, introducing too many features, many of them being dependent and linked to other ones (like port.dst of a TCP flow and app), we are introducing redundancies and we risk falling into *Curse of Dimensionality*.

Therefore, it is mandatory to select the best flows' properties to be used as features in order to train the Deep Neural Network.

For this project two different solutions have been studied: one based on a flows representation that takes into account the IP address of the peer outside the LAN (ip.dst) and one that consider the information that pertains to the service carried by the flow (service, category and app). We are going to analyse both ideas in the next two sections.

3.2.1 Ip destination matrix

In this solution only the ip.dst field of each flow, which is the IP address of the flow endpoint outside of the LAN, is considered as the only fundamental property and as a criterion to train the Deep Neural Network.

Resting on this idea, finding a way to represent correctly this property as a feature (or features) is needed.

At a first glance it may seem a good idea providing as inputs to the Neural Network couples of type (MAC, ip.dst) for each flow in the dataset (remember that the MAC address is the label and ip.dst the only feature of a sample, in this case).

The problem with this approach is that having so few sample features (just one in this case) it is unlikely that the Network will end up working in the way we expect. Consider, for instance, an IP address that is visited by more than one host during the capture period, like 172.217.164.174 that corresponds to the Google's search engine *www.google.com*. In this scenario, we will feed more times the Neural Network this IP address, paired however not always with the same MAC address. Which host we expect the Network will predict when, completed the training phase, we will ask for *www.google.com*? The Network will answer always with the same MAC address, one of the many that contacted that IP, without any chance for us to know with which criterion. Clearly this is an undesired behaviour.

Rather than pairing one IP address at a time to each MAC, a better idea could be associating an IP address list (all those visited by that host), which means the ip.dst of all the flows that belong to him; following this strategy we will not incur into the previous problem.

Unfortunately, a Neural Network asks a constant size inputs (constant number of features) not allowing to exploit this approach. This is because it is natural that different users have a different quantity of visited IP addresses.

The idea that was envisioned is to represent the IP address series as an RGB image, namely a 256x256x256 matrix; this because:

- the range of values of an IP address octet is from 0 to 255,
- it is possible to treat each sample as an image, allowing us to exploit Convolutional Neural Networks,
- similar IP addresses, usually belonging to the same subnetwork, will be represented near.

A Convolutional Neural Network is a Deep Neural Network with particular layers, called convolutional, specialized in learning patterns at a spatial level, which perfectly fits to images.

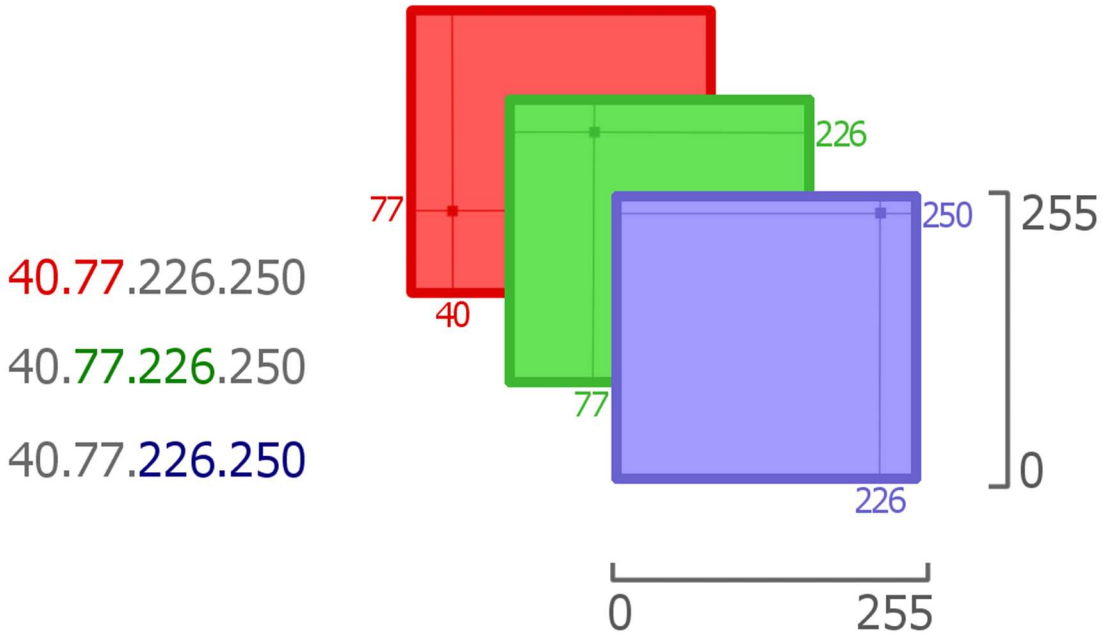

Figure 8 - The IP address' octets are exploited to detect one point in each image's channel.

The logic according to which every single IP address is mapped into an RGB matrix is:

1. The first two octets of the address are taken and used respectively as abscissa and ordinate in order to detect a point in the red channel of the matrix.
2. The value to be written in that position is the number of times (frequency) that IP address appears as ip.dst in the flows generated by the currently considered host.
3. The second and third octets are taken and exploited to find a point in the green channel.
4. The value to be written follows step 2's logic.
5. The last two octets are taken and used to find a point in the blue channel.
6. The value to be written follows step 2's logic.

In this way we locate three points on the matrix, one for each channel. This procedure is performed for each IP addresses that is visited by the host and, once a matrix for each different IP is obtained, these matrices are summed up; according to this logic we will end up having just one image that encloses the information about all the IP addresses contacted by a specific host.

However, stockpiling all the IP addresses in a single image we stumble into the problem where we have just one sample for each host for training the Neural Network, which is clearly unfeasible.

In order to get a large image dataset, it was chosen to have a host IP matrix for each 5 minutes of capture. Hence, the values to be written in the respective channels (steps 2, 4 and 6) are the number of times that IP address appeared in that 5-minutes interval; this means, of course, that the same address can appear in more following matrices for the same user.

At the end of this step we have our dataset of the first type ready, namely a list of couples (MAC, IP matrix for 5-minutes interval).

All the IP matrices belonging to a day are stored on disk in a hdf5 (hierarchical data format 5) file.

This format was chosen because:

- It is designed to store and organize large amount of data.
- Allow to reduce the number of reads on disk for images in the same file.
- It is hierarchically organized like the file system, allowing to organize a directory-like structure inside the file. This feature was exploited creating subgroups for each 5-minute interval, containing all the respective IP images.

For more details on the actual implementation, check section 4.2.1.

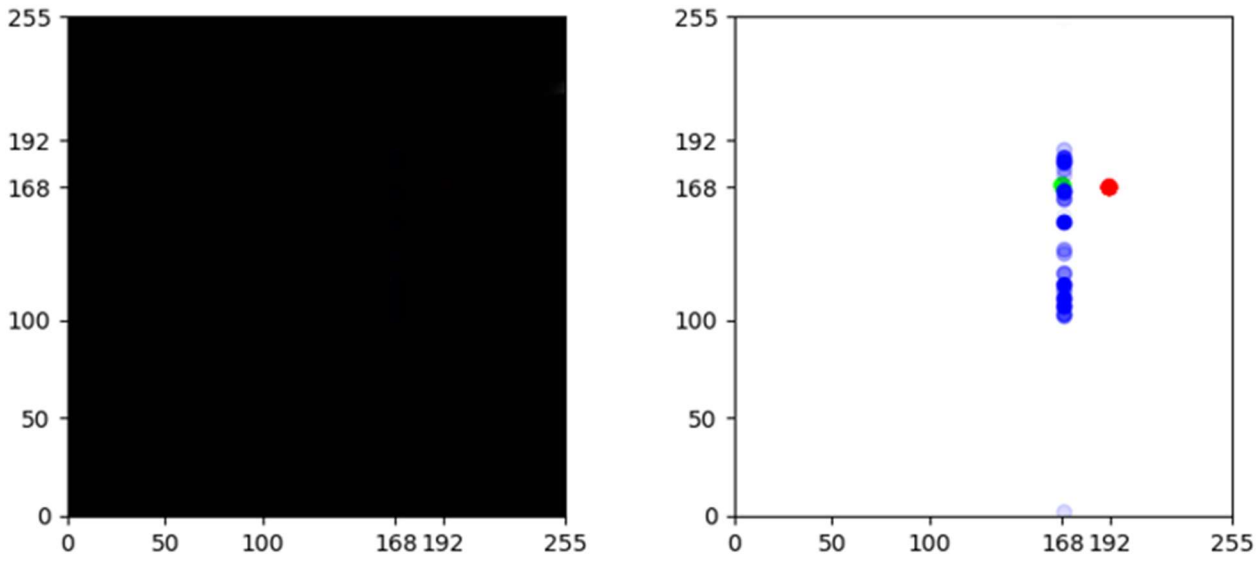

Figure 9 - On the left, an actual IP matrix resulting from this process. On the right, the same IP matrix but on each detected position is written 255, while the point's alphaness refers to the IP frequency. Both images carry the same information, they are just displayed in different ways. We can think these images belong to the edge router because of the stockpile of points in the area around the subnet 192.168.169.1/24 while everything else is empty: it is very unlikely that the edge router is the endpoint of a conversation with a peer outside the LAN; probably all the depicted flows were DNS request.

As inferred from the picture above, it is difficult distinguishing at bare eye where the points identified by the algorithm are located; this is because in a 5-minutes interval the visits' frequency

on the same IP address are limited, thus not so often the red, green and blue values are high, resulting in a nearly all black image.

This is not a problem for the Convolutional Neural Network because, although is difficult for a human being distinguishing two of these images, the Network works at the bit-level and can clearly detect the differences between one matrix and the other one.

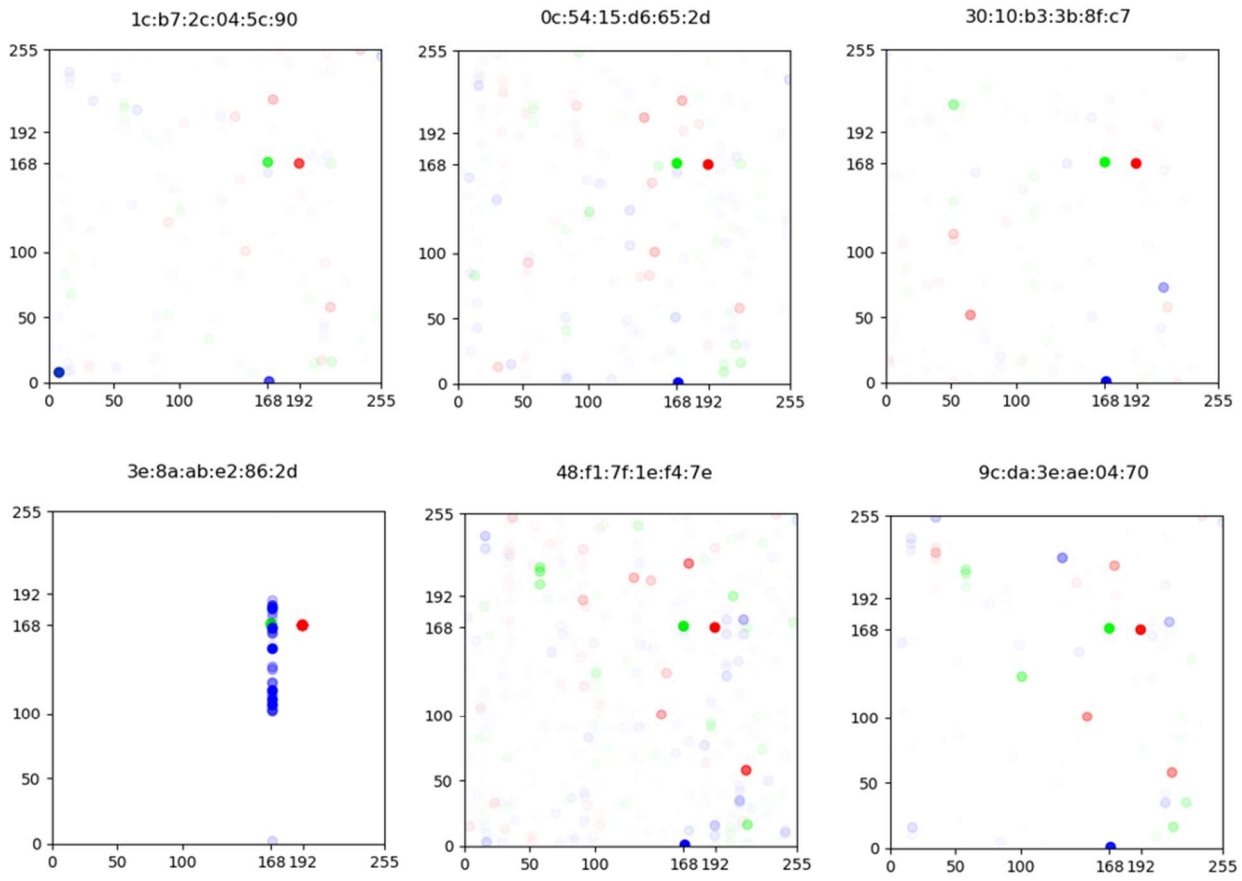

Figure 10 - Some IP matrices from Wednesday 10 April 2019, one-hour long interval, from 12.00 to 13.00.

3.2.2 Encoded Ip destination matrix

Before analysing the second solution, which is based on flows' application-layer information, it is worth showing a variation of the first one on the IP matrices.

As the title says, the idea is to encode the image matrix in a new format. The motivations that encouraged to consider this option are:

- The original matrix is 256x256x256, but most of the cells are filled with zeros.
- A more compact representation requires less RAM when reading the dataset in order to feed the Neural Network.
- A more compact representation allows the network to speed up the training phase.

The aim is to reduce the dimensionality of the IP matrices while keeping the information they bring; following this path, we will be able to reduce the spatial overhead of the previously chosen representation.

This is a classic unsupervised learning problem called *dimensionality reduction*, namely the process of reducing the utilised number of features (in this case the 256x256x256 cells).

It was chosen to exploit a non-linear type approach of dimensionality reduction, by means of a particular type of Neural Network called Autoencoder.

An Autoencoder is exploited in order to find out efficient encodings in an unsupervised manner; its objective is to learn an alternative representation of a data structure, reducing the noise it carries.

Along with this component that encodes (Encoder), the Autoencoder is composed by another element, Decoder, which learns how to rebuild the original input from the 'reduced' representation produced by the Encoder.

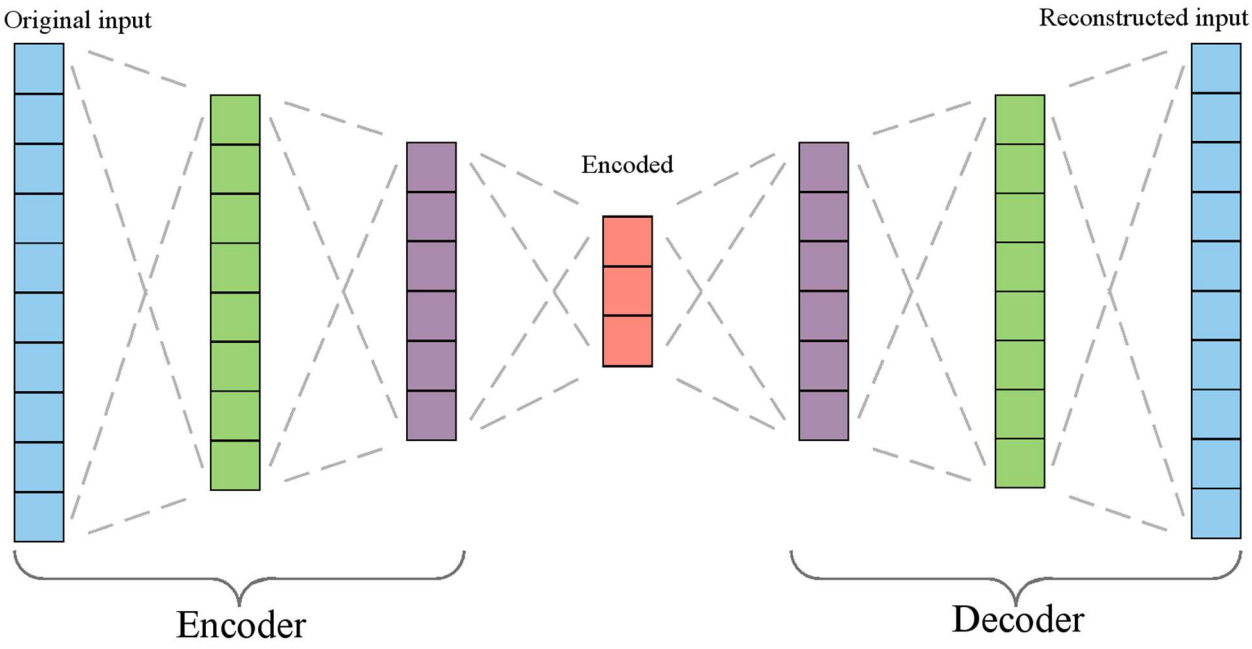

Figure 11 - Autoencoder architecture.

The Autoencoder is trained in order to make correspond the Encoder input with the Decoder output, meaning that the Network is able to successfully move from a representation to the other one; in other words, the transformation realized by the Encoder produces a negligible information loss.

For this kind of problem, the developed Autoencoder is composed by some convolutional layers, in order to treat better the input images.

After various tests it was concluded that a good encoded representation for an IP matrix is a 16x16x16 image. It was chosen to keep the data as an image always for exploiting Convolutional Neural Network's properties, in particular the one already built for the 256x256x256 images.

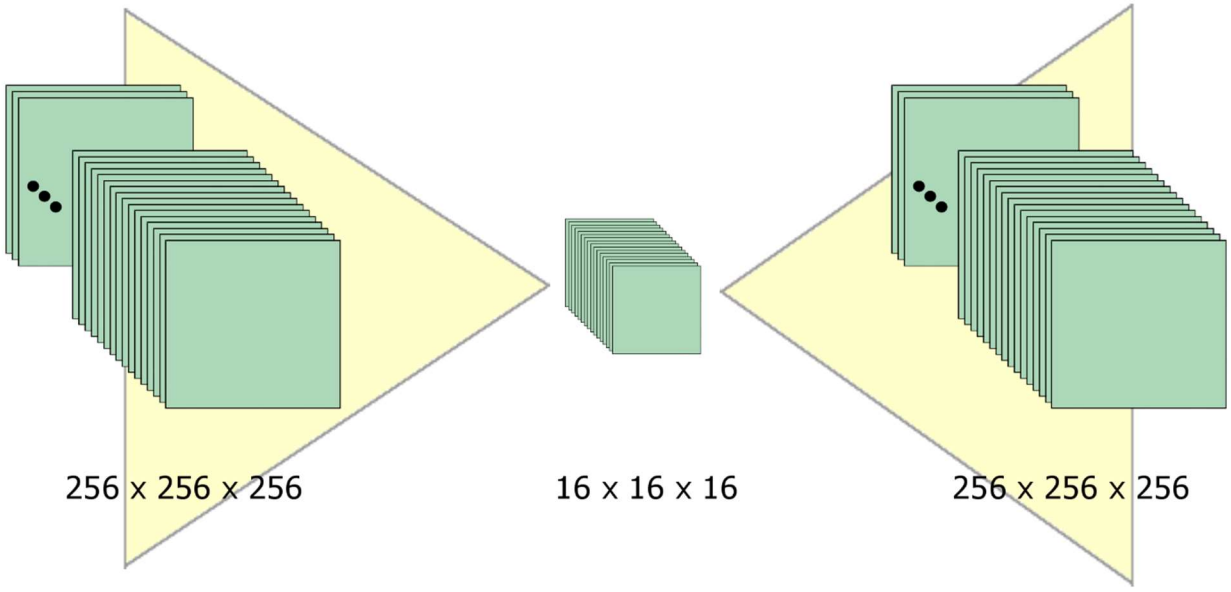

Figure 12 - Input and output sizes of the developed Encoder and Decoder

The developed Autoencoder reaches an accuracy of 99.9%. Also these images are saved in hdf5 files, one for each day of capture. For more details on the actual layers' architecture, the reader should continue on section 4.2.3.

This is a separate solution respect to the original IP matrices one; both will train a Deep Neural Network and will bring different results.

3.2.3 Service distribution vectors

This solution is based on the exploitation of the application-layer information gathered thanks to nDPI and stored in the fields service, category and app in order to train the Deep Neural Network. The accurate meaning of these fields is explained in section 3.1.

Resting on this idea, finding a way to represent correctly this property as a feature (or features) is needed.

Differently from the first case, in this situation, it was chosen to use as features an array containing the probability distributions of the values in service, category and app for each host.

Consider for this example only the category property, and a host who generated 200 flows, 70 of them being of streaming category, 120 being web traffic and 10 being mail traffic. This means that his service distribution vector will have a value of 0.35 in the array position relative to category = streaming, 0.6 in category = web, 0.05 in category = mail and 0 in all the other values.

Remember that it is mandatory to keep all the vector missing fields to 0, because the Deep Neural Network requires constant-size inputs; hence each host needs to have arrays of the same length.

Likewise to the previous case, it is not possible to create a vector on all the flows captured for each user, we will end up having just one sample for each MAC address. For this same cause it was chosen to create a sample for each 5 minutes of capture, which means having a vector with the flows' probability distributions from a 5-minute interval.

Finally, it was chosen to subdivide this solution into three different configurations as follows:

- In the first configuration only the information from category field is exploited. In particular a training set sample is the per-host distribution of its category values in a 5-minute interval. Of course, the label is the host's MAC address.
- The second configuration adds to the first one the app field; this means that each element of the vector refers to a value of the couple (category, app). What we obtain from this is a longer array which shows a finer granularity on the application-level layer of each flow. For instance, if in the first configuration a host presented a value of 0.2 in category = chat, in

this one he will have that the sum of the values on the couples (chat, telegram), (chat, slack), (chat, …) is always 0.2.

- The last configuration relies on the second one and adds to the couple the service field; in this way each element of the vector refers to the triad (category, app, service). This solution adds the information about how the service is carried; for instance, on a service where category = web it can provide information on the security level of the flow by distinguishing cases where service = http or service = ssl.

This solution is thus subdivided in three parts, each one of them will separately train the Deep Neural Network and will allow to observe different results.

3.3. Model

As mentioned in the previous section on feature engineering, the first component of the model is a Deep Neural Network that will be trained on the detected features.

More precisely a Convolutional Neural Network for treating the IP matrices (encoded and not) and a normal Deep Neural Network for fitting the service distribution vectors will be built. It is important to note down that the MAC addresses with few IP images are removed from the analysis (also from the application-layer one) allowing to detect 44 hosts that are actually present and active in the LAN.

Which is the objective of this Network? It was said that it will have MAC addresses as labels and in this way, depending on the input it will be able to predict the correct MAC.

The purpose of this Neural Network is obtaining a kind of *confusion matrix* from its predictions, and nothing else.

A *confusion matrix* is a matrix that displays the performances of a Machine Learning algorithm, usually a supervised learning one. Each column represents the instances of a predicted class while each row depicts the instances of an actual class.

In this case, the actual confusion matrix is modified as follows: the number of different predictions for each instance is divided by the total number of predictions (for that same instance) in order to obtain a statistical distribution. In this way each row can be read as: "Of all the instances belonging to the class *MAC A*, a percentage of them are predicted as belonging to *MAC A*, another one to *MAC B* and another one to *MAC C*", like in the following table.

	00:21:cc:d6:dc:9a	c2:22:09:f2:5f:e8	d0:17:c2:dc:55:3f
00:21:cc:d6:dc:9a	0.63	0.17	0.2
c2:22:09:f2:5f:e8	0.11	0.81	0.08
d0:17:c2:dc:55:3f	0.07	0.15	0.78

Figure 13 - Example of the custom confusion matrix. Each row represents how the Neural Network predicted the input samples, tagged with that MAC address.

The idea is to utilise the confusion matrix rows as the bearing information for clustering, grouping together users that show a similar distribution. The clustering algorithm, as thoroughly explained in section 3.3.2, calculates a dissimilarity score between hosts comparing their rows in the matrix, namely the way these users are classified by the Deep Neural Network.

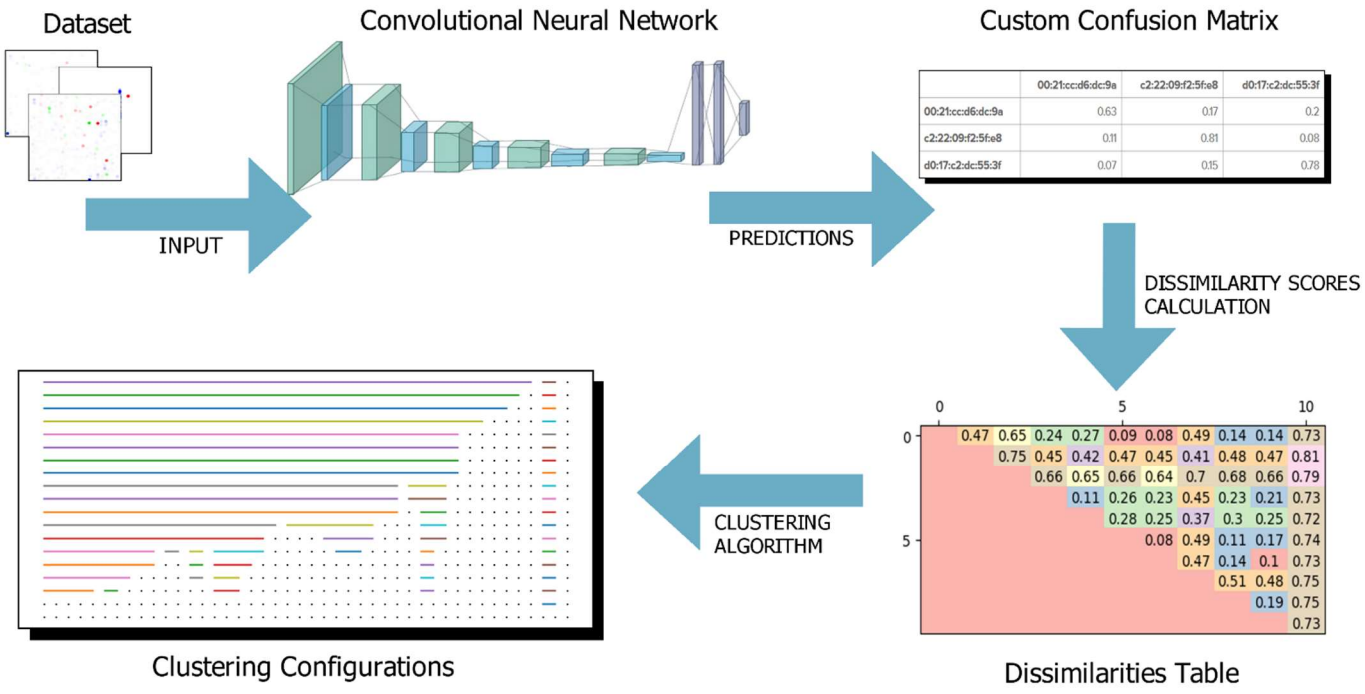

Figure 14 - Model's architecture and its various steps in order to obtain the clustering results; in particular, it is shown the case of original IP images as input.

In the image above it is possible to see the various components of the System, starting from the dataset created through the data collecting and feature engineering steps, up to the hosts final clustering.

The next sections will analyse with more details the Deep Neural Network and the clustering phase.

3.3.1 The Deep Neural Network

In this section we are going to show the Neural Network model and how it is exploited in order to optimize our unsupervised learning approach. Specifically, two Deep Neural Networks were developed, one fed with IP matrices as inputs and with service distribution vectors; both exploit the Python's library *Keras*.

What changes between them is only the input and hence the layers' structure; the objective is always the same: training the Network in order to predict MAC addresses while reducing as much as possible the similarity between the MAC themselves.

The network is built so that it is possible to check the general dissimilarity degree among all the MAC addresses at the end of each training epoch.

The dissimilarity between two hosts is calculated by adding all the elements of the row that is produced from the difference between the two hosts' rows (in the custom confusion matrix) and dividing by two.

	00:21:cc:d6:dc:9a	c2:22:09:f2:5f:e8	d0:17:c2:dc:55:3f
00:21:cc:d6:dc:9a	0.63	0.17	0.2
c2:22:09:f2:5f:e8	0.11	0.81	0.08
d0:17:c2:dc:55:3f	0.07	0.15	0.78
difference between 00:21:cc:d6:dc:9a and c2:22:09:f2:5f:e8	0.52	0.64	0.12

dissimilarity = (0.52 + 0.64 + 0.12) / 2 = 0.64

Figure 15 - Table showing how the dissimilarity score between the MACs 00:21:cc:d6:dc:9a and c2:22:09:f2:5f:e8 is calculated. First their row is subtracted and then the resulting elements are added up; the sum is finally divided by two.

The objective function is:

$$F = \min\left[DISS - \sum_{i,j}^{N} diss\left(MAC_i, MAC_j\right)\right]$$

Where:

- N is the number of different MAC addresses,
- $diss\left(MAC_i, MAC_j\right)$ is the function that calculates the dissimilarity score between two MACs,
- $DISS = \frac{N!}{2\ (N-2)!}$ which represents the overall score obtained when all the couples of MAC address have maximum dissimilarity, that means dissimilarity = 1.

In particular, in the training phase, the Network performs the following steps:

1. It receives as input a batch of samples with the relative labels.
2. It updates every layer's weights.
3. It predicts the MAC addresses for another batch of samples.
4. It builds the custom confusion matrix from the predictions.
5. It calculates for each MAC couple their dissimilarity.
6. It calculates the objective function fitness, which is dependent on the dissimilarities, and it saves the result.

These phases compose a *training step* and they are executed each time an input batch is provided to the Deep Neural Network. This *training step* is executed until every single sample (in this case every IP matrix or service distribution vector) is given to the Network. This *training steps* series is called epoch. The training phase is concluded after a prefixed number of epochs.

At the end of each *training step*, hence, the fitness, which is an index of similarity between all the MAC addresses, is calculated. We want to minimize that value in order to ease to the clustering algorithm the job of distinguish one host from another one.

The Deep Neural Network performs an optimization algorithm each time the training phase is concluded, namely at the last epoch, trying to avoid objective function local minima; more precisely what the Network does is:

1. For each training epoch, selects the *training step* with the best fitness, which means the lowest.
2. It obtains the Neural Network weights at those *training steps* (each *training step*, the resulting weights and fitness are saved).
3. It creates a new weights' configuration calculating the arithmetical average of those weights detected in the previous steps.
4. It injects the new weights' configuration in the Neural Network.

5. Calculates the fitness as usual, in order to test if the new configuration outperforms the previous ones.
6. If the fitness has improved, a new training phase is started again with this new weights' configuration; otherwise a new training phase is started but the previous weights (the ones before step 3) are restored.

The training phase is repeated a very high amount of times, until the fitness reaches an asymptote and does not improve any more. At that point the Deep Neural Network is finally trained.

For a more in-depth look at how the Network is built and trained, refer to section 4.3.

Following this procedure it is possible to exploit the trained Deep Neural Network to predict MAC addresses from their traffic representations, in order to obtain a custom confusion matrix that we know presents a minimum degree of similarity between MACs and that can be used in the next step.

It is certain that training the Network, the similarity score between hosts will follow a negative trend; this is because the Network tries to distinctly distinguish each host so that its every prediction on a new sample will be correct, meaning that the *backpropagation* updates the weights in order to obtain a diagonal confusion matrix. A diagonal confusion matrix, as shown below, features the maximum possible dissimilarity score between hosts (dissimilarity = 1), which is what we look for.

	00:21:cc:d6:dc:9a	c2:22:09:f2:5f:e8	d0:17:c2:dc:55:3f
00:21:cc:d6:dc:9a	1	0	0
c2:22:09:f2:5f:e8	0	1	0
d0:17:c2:dc:55:3f	0	0	1

Figure 16 - A diagonal custom confusion matrix, obtained from a Neural Network that predicts correctly from each traffic sample given as input; it is easy to see that the dissimilarity score between each MAC address couple is 1.

3.3.2 Custom Clustering

Once the Deep Neural Network is trained and the custom confusion matrix with minimum host similarity is obtained, it is possible to build a table that shows all the dissimilarity scores for each MAC address couple.

This table is the starting point for the clustering algorithm.

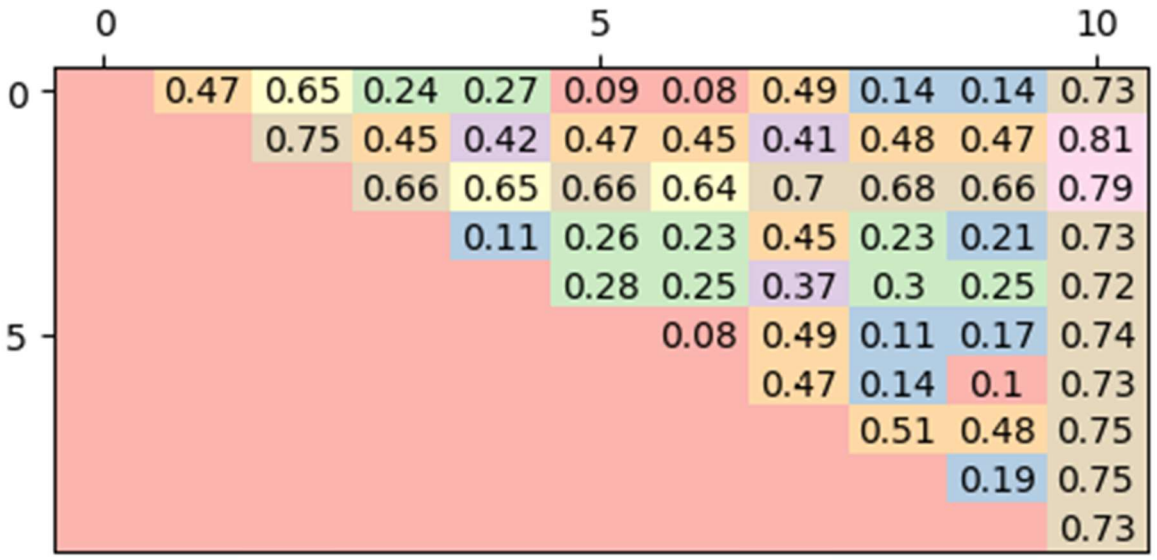

Figure 17 - Example of dissimilarity table that shows the dissimilarity scores among 10 hosts. Each row and column identify a MAC address.

A high value in a cell means that the two hosts' dissimilarity score is high and that means the two MAC addresses are not so much similar; to the contrary, a low value notifies that those two hosts were recognized to be so much similar by the Neural Network that their prediction profile (the row in the custom confusion matrix) has very little differences. This means that the Network struggled to distinguish one from the other one.

The Deep Neural Network is trained on the traffic generated by the users, thus makes sense declaring that two MAC addresses with very low dissimilarity have a comparable traffic and feature a similar behaviour in the LAN.

It is worth noticing the differences between *Figure 18* and *Figure 19* that show two dissimilarities tables on all the 44 MAC addresses participating at the analysis. The first table is obtained from a not sufficiently trained Neural Network, with a fitness value of 787.45; the second one is obtained after the training phase (in the way it was discussed in the previous section) with a fitness of 360.39.

The former, not having tried to reduce the similarities between hosts, shows smaller values inside the cells compared to the latter, denoting that, for the first Network, the hosts are less distinguishable than for the second one.

It was chosen to reduce the similarities between MAC addresses so that it was possible to put in the same cluster hosts that are actually similar: a low dissimilarity score in the second figure gives more warranties on the real similarity of two hosts' behaviours compared to a low score in the first one.

The idea is grouping together in one cluster all the MAC addresses that show a lot of similarity, allowing to safely state that they have a similar behaviour.

In order to do so it is indispensable to establish a threshold value that will help to agree on what means high or low dissimilarity.

If, for instance, considering the previous table featuring 10 MACs, we take the threshold at a value of 0.1, we can state that MAC number 0, 5 and 6 are very similar and can be treated in the same way by putting them into the same cluster.

This need of establishing a threshold opens the way to different scenarios with different clustering configurations, all valid, where the analyst has chosen different values.

By choosing a very small threshold, it is likely that no MAC address will be merged in a cluster with other hosts, producing a clustering configuration where one MAC = one cluster.

Selecting instead a big threshold, like 0.8, it is probable that it will be obtained a configuration with just one cluster grouping together all the hosts.

All the intermediate values will produce instead a different number of clusters with different hosts inside them. *Figure 20* shows how driving the threshold value makes possible obtaining different clustering configurations.

As explained before, increasing the threshold, the clusters dimensions increase and the clusters number reduces; vice versa decreasing the threshold, the hosts start popping out from the clusters and being distinguishable from those MACs that before were their cluster companions.

The clustering algorithm that is being utilised can be described as hierarchical and the graph above remembers a dendrogram: each neighbouring configuration differs of a number of hosts that enters into or exits from a cluster, depending on the direction we are looking the graph.

Differently from a dendrogram however, there is not an optimal value where to “cut” the graph, meaning choosing the threshold. It is in fact the analyst’s duty studying the different configurations, their meaning and the needs of its LAN in order to select the most appropriate threshold value for that situation.

If the reader is interested in the discussion about the results, he should continue to chapter 5; otherwise he can continue in chapter 4 that is about the actual implementation of the model.

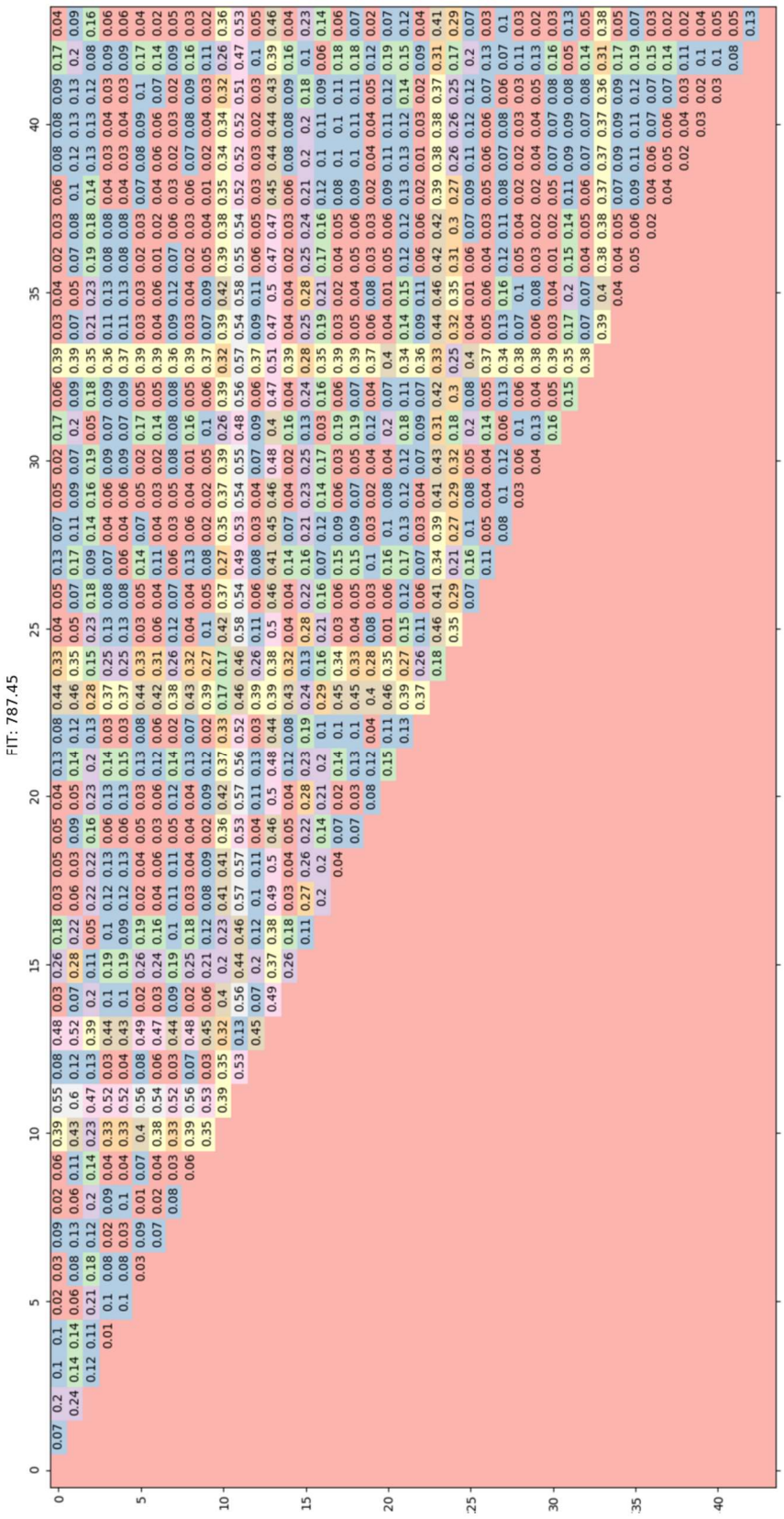

Figure 18 - Dissimilarities table obtained from a Deep Neural Network trained until the objective function reached fitness 787.45.

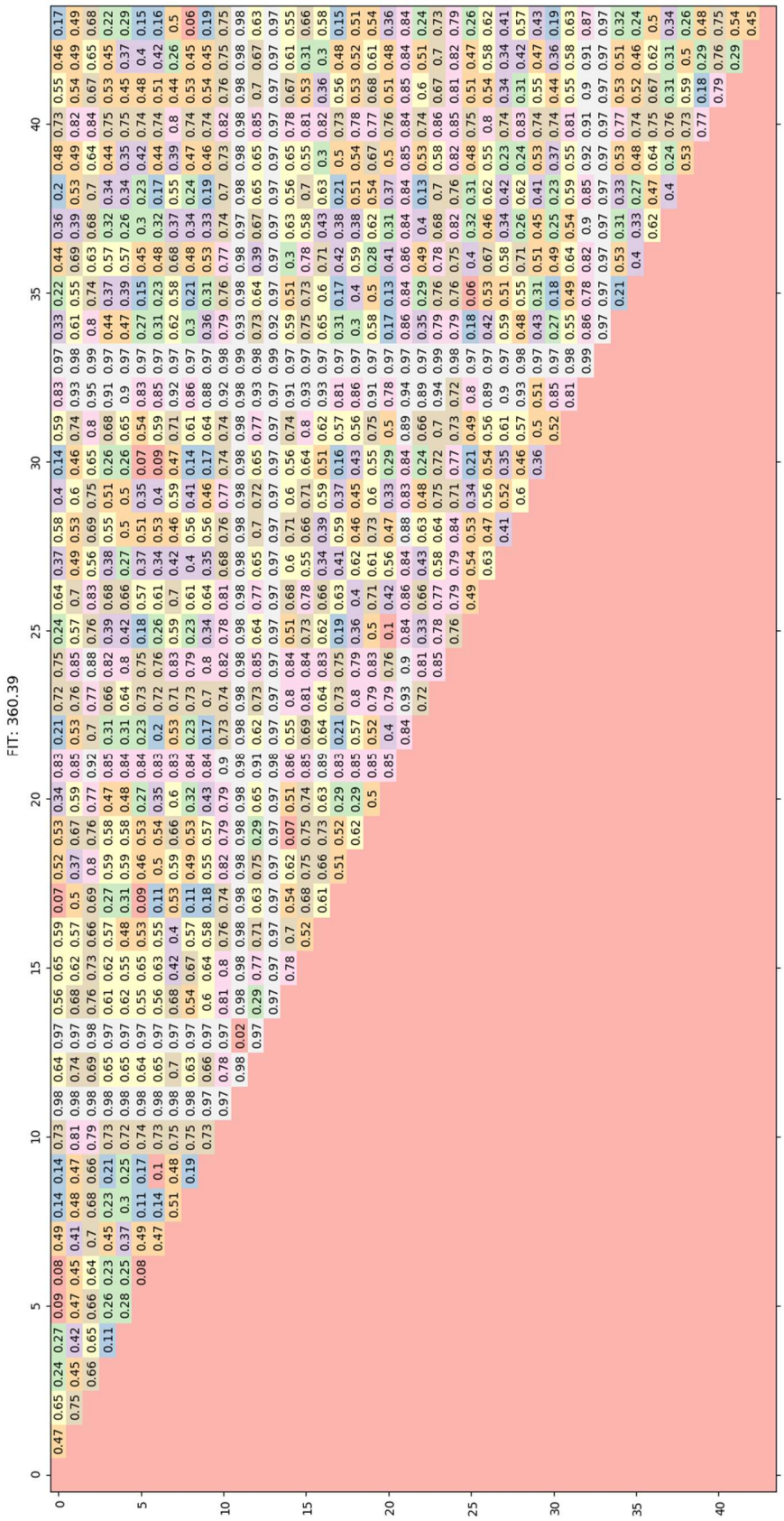

Figure 19 - Dissimilarities table obtained from a Deep Neural Network trained until the objective function reached fitness 360.39.

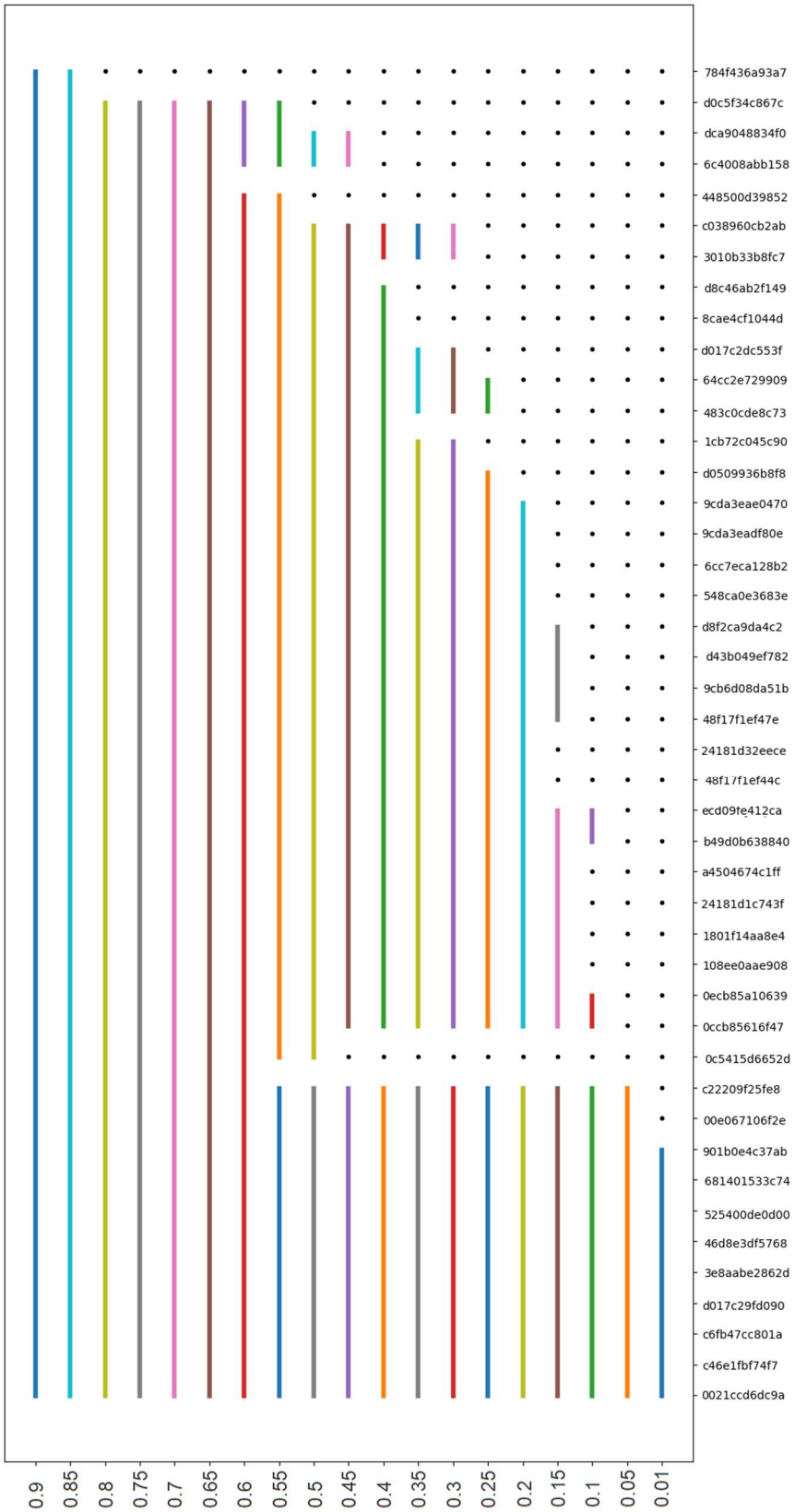

Figure 18 - Graph obtained by combining different configurations through the selection of different threshold values. Each row represents the configuration corresponding to the chosen threshold value. Each point identifies a host with a certain MAC address.

4. Implementation

After the explanation of the model architecture and the rationale behind data collecting and feature engineering phases, in this chapter, we are going to see the actual implementation of the most important steps of each phase.

The project was entirely developed utilising Python programming language, because nowadays it is the Data Analysis standard, having a lot of libraries dedicated to the subject in addition to statistics, Machine Learning and Neural Networks and, last but not least, a surprisingly active community.

During the project development, it was largely exploited Pandas, an open source library providing high-performance, easy to use data structures and Data Analysis tools for the Python programming language. Pandas is based on NumPy, the fundamental library package for computing with Python.

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data by means of its object type ndarray.

From the official NumPy user API documentation [7], on ndarray: "*An array object represents a multi-dimensional, homogeneous array of fixed-size items. An associated data-type object describes the format of each element in the array (its byte-order, how many bytes it occupies in memory, whether it is an integer, a floating-point number, or something else, etc.)*".

This kind of array is important for its performances and the amount of operations and methods it supports, considering also the fact that indexing and slicing on a ndarray is really trivial.

The exploitation of Pandas and NumPy, whom back-end code is often written in C language for performance reasons, has accelerated in a considerable way both development time and algorithm execution time compared to using pure Python.

The data structures introduced in Pandas and widely utilised on this project development are objects of type Series and DataFrame.

Series is a one-dimensional labelled array capable of holding any data types (integers, strings, floating-point numbers, Python objects, …). Series acts very similarly to a ndarray and is a valid argument to most NumPy functions.

DataFrame is a 2-dimensional labelled data structure with columns of potentially different types. From Pandas official user API documentation [8], "*You can think of it like a spreadsheet or SQL table, or a dictionary of Series objects. It is generally the most commonly used Pandas object. Like Series, DataFrame accepts many different kinds of input*".

4.1. Data Collecting

Let's start analysing the data collecting phase, where, starting from the traffic sniffed on the edge router, it is obtained a dataset organised in a .csv files, one for each day of capture, containing all the flows with their fundamental properties.

Unfortunately, for space reasons, it is not possible to show all the developed source code, therefore only the most important and complex steps are reported.

Let's skip the Ethernet frames extraction from the original .pcap file (as mentioned in section 3.1, for this task it was employed tshark network tool) and let's analyse the logic of merging Ethernet frames into the flows they belong to.

```
# List of fields used for the flow extraction; careful when deleting fields, because some of them
# are used in the calculation and if missing the interpreter will throw a runtime error
FIELDS = np.array(['frame.number', 'frame.time_epoch', 'frame.len', 'eth.src', 'eth.dst',
          'ip.len', 'ip.src', 'ip.dst', 'ip.proto',
          'tcp.srcport', 'tcp.dstport', 'tcp.stream', 'tcp.len', 'tcp.seq', 'tcp.ack',
          'tcp.connection.syn', 'tcp.connection.sack', 'tcp.connection.fin', 'tcp.connection.rst',
          'udp.srcport', 'udp.dstport', 'udp.stream', 'tcp.analysis.ack_rtt', '_ws.col.Protocol'])

# read the dataset in .csv format
data = pandas.read_csv(source_file, names=FIELDS)

# due to the fact that a tcp frame has the udp ports to Nan and a udp frame has
# tcp ports to Nan, copy on the Nan couple the other couple, so that just
# tcp.srcport/dstport or udp.srcport/dstport can be used for all the frames.
# For example a tcp frame will become from:
# IP.PROTO  IP.SRC          IP.DST          TCP.SRC TCP.DST UDP.SRC UDP.DST
# 6         192.168.1.100   130.0.200.20    43468   443     nan     nan
# to:
# IP.PROTO  IP.SRC          IP.DST          TCP.SRC TCP.DST UDP.SRC UDP.DST
# 6         192.168.1.100   130.0.200.20    43468   443     43468   443
data.loc[data['udp.srcport'].isnull(), 'udp.srcport'] = data['tcp.srcport']
data.loc[data['udp.dstport'].isnull(), 'udp.dstport'] = data['tcp.dstport']
data.loc[data['tcp.srcport'].isnull(), 'tcp.srcport'] = data['udp.srcport']
data.loc[data['tcp.dstport'].isnull(), 'tcp.dstport'] = data['udp.dstport']
# now I can delete the columns udp.src and udp.dst because their info are carried
# in the columns tcp.src and tcp.dst
data = data.drop(columns=['udp.srcport','udp.dstport'])
# now rename tcp.src and tcp.dst to port.src and port.dst
data.rename(columns = {'tcp.srcport':'port.src' , 'tcp.dstport':'port.dst'}, inplace=True)
```

As depicted in the code snippet below, a list of Ethernet frames is read from a .csv file by means of Pandas' read_csv function, which returns a DataFrame object.

Afterwards, because of the need of having the transport-layer port information of each frame in just one field (while tshark stores this information in two different fields whether the frame is carrying TCP streams or UDP datagrams), the values stored in tcp.srcport and tcp.dstport are copied in udp.srcport and udp.dstport for TCP flows (and vice-versa for UDP ones). In this way we have the port information on the same fields and it is possible to drop the other couple, allowing to rename them in port.src and port.dst.

Therefore, as explained in section 3.1, the frames stored in the DataFrame are grouped by means of Pandas' groupby function twice in waterfall.

```
# create two additional columns for converted ip addresses on their int representations
data['ip.src.int'] = data['ip.src'].apply(ipaddress.ip_address).apply(int)
data['ip.dst.int'] = data['ip.dst'].apply(ipaddress.ip_address).apply(int)
# calculate the sum of ip address src/dst and the sum of port src/dst and store it on
# two new columns
data['hash_addr'] = data['ip.src.int'] + data['ip.dst.int']
data['hash_ports'] = data['port.src'] + data['port.dst']

# columns on to perform the fisrt groupby
grouping_columns = ['ip.proto', 'hash_addr', 'hash_ports']
# columns on to perform the second groupby
fivetuple_columns = ['ip.proto', 'ip.src', 'ip.dst', 'port.src', 'port.dst']

# group all the data on address and port hash in order to collapse all frmaes that
# belong to the same flow (and being independent to the frame direction)
grouped_data = data.groupby(grouping_columns)
```

The first GROUPBY, performed on the transport-layer protocol, the sum of the IP address and the sum of ports, allows to detect the frames belonging to the same flow, regardless of their travel direction.

If we would have grouped directly by the 5tuple, we could not have in the same group a flow's incoming and outgoing traffic.

```
df_data = pandas.DataFrame()

# loop through the frames grouped by flows (ip+port hash)
# n is the (ip.proto, hash_addr, hash_port) value for the group
# s is the list of frames that belongs to n
for n, s in grouped_data:

   # calculate for each flow the amount of traffic exchanged and its per frame avg
   tot_len = s['frame.len'].sum()
   avg_len = s['frame.len'].mean()
   # duration of each flow, calculated subtracting the timestamp of the first frame
   # to the timestamp of the last one
   flow_duration = s['frame.time_epoch'].max() - s['frame.time_epoch'].min()
   # save on 'filtered' the frames that have a public source ip address, because, in order
   # to calculate the rtt, we are not interested in the rtt from inside the lan to the border router
   filtered = s[ s.apply(lambda row: ipaddress.ip_address(row['ip.src']).is_global, axis=1) ]
   # gets the flow's rtt by taking the minimum non-zero rtt for each frames
   rtt = filtered[ filtered['tcp.analysis.ack_rtt'] > 0]['tcp.analysis.ack_rtt'].min()
   incoming_traffic_size = filtered['frame.len'].sum()

   # now group by the 5 tuple every group found; this should lead to just one
   # group in udp flows and 2 groups (the traffic directions) in tcp flows
   # also filters out all the frames with a not private ipsrc, aka from wan to lan
   group_ft = s[ s.apply(lambda row: ipaddress.ip_address(row['ip.src']).is_private, axis=1)
   ].groupby(fivetuple_columns)

   # loop through the flows obtained by the second group by
   # k is the (ip.proto, ip.src, ip.dst, port.src, port.dst) value for the group
   # v is the list of frames that belongs to k
   for k, v in group_ft:

      # interframe time is the time that occurs between two frames sent on the same flow by the same
      # host (in this case the one in the lan)
      v['interframe'] =  v['frame.time_epoch'].diff()
      # get the mac address of the user in the lan
      macsrc = v['eth.src'].iloc[0]

      # create a dictionary with the following keys:
      dic_stat = {
          '1_proto': k[fivetuple_columns.index('ip.proto')],
          '1_zmac': macsrc,
          '2_ipsrc': k[fivetuple_columns.index('ip.src')],
          '3_ipdst': k[fivetuple_columns.index('ip.dst')],
          '4_portsrc': k[fivetuple_columns.index('port.src')],
          '5_portdst': k[fivetuple_columns.index('port.dst')],
          'time': v['granularity'].min(),
          'totsize': tot_len,
          'avgsize': avg_len,
          'duration': flow_duration,
          'rtt': rtt,
          'inlan': incoming_traffic_size,
          '6_tproto': v['_ws.col.Protocol'].values[0],
          'zinterframe': v['interframe'].mean()}
      # append the dictionary in this dataframe; it will contain a dictionary
      # for each flow (and direction)
      df_data = df_data.append(dic_stat, ignore_index=True)

# after the nested loops
df_data.to_csv(dest_file + '.csv', header=['proto','macsrc','ipsrc','ipdst','portsrc','portdst',
'tproto','avgsize','duration','inlan','rtt','time','totsize','iframe'], index=None)
```

The second GROUPBY, which is actually performed on the 5tuple values, allows to discriminate incoming and outgoing traffic in order to easily extract some flow's properties; moreover, it is able to detect the rare flows that were erroneously merged together by the first GROUPBY operation.

On a following iteration the second GROUPBY was employed just on the outgoing traffic because:

1. It was noticed that no frames were added to the wrong flows.
2. All the flows information could be easily obtained from the frames of that direction.

As the reader can see in the code snippet above, the employment of a DataFrame object, makes the whole process trivial thanks to its offered methods and its accessing policies.

In particular, it is easy to use indexing in order to get a column and apply a function to all the values, like, for example, getting the field frame.len of all the grouped frames and then sum all of them to get that flow's total amount of data transferred.

After having done this job of merging all the frames into their respective flows, for the sake of organization, it was chosen to collapse in a single file all the .csv that carried traffic that belongs to the same day.

```
# get a list of filenames that match the first argument
arg = ( sys.argv[1] if ('*' in sys.argv[1]) else (sys.argv[1]+'*') )
sourcefiles_list = glob.glob(arg)

data = []
for flow_filename in sourcefiles_list:
   df = pandas.read_csv(flow_filename)
   data = np.append(data, df)

# ---------------
# create the flows merged file
# ---------------
data = data.flatten()
data = np.reshape(data, (int(data.size/17), 17))

allflows_df = pandas.DataFrame(data, columns=['proto','macsrc','ipsrc','ipdst','portsrc',
'portdst','tproto','avgsize','duration','inlan','rtt','time','totsize','iframe','cat','service',
'app'])
allflows_grouped = allflows_df.groupby(['proto','ipsrc','ipdst','portsrc','portdst'])
```

In order to achieve this result, the files that need to be merged are iteratively read and all their frames are stored in a single DataFrame object.

Then, a GROUPBY is performed on the 5tuple in order to group together the same flows that exist in more than one .csv file. For each group, the properties that need to be updated are recalculated (like flow's duration and total amount of transferred bytes) as shown in the code below.

Finally, for each of the detected groups, a row with the correct flow's properties is stored in a final DataFrame and, in the end, saved as a new .csv file.

```
# build interesting fields
table = []
for ftuple, flows in allflows_grouped:

   inlan = flows['inlan'].sum()
   totsize = flows['totsize'].sum()
   row = [
      ftuple[0],
      flows['macsrc'].values[0],
      ftuple[1],ftuple[2],ftuple[3],ftuple[4],
      flows['tproto'].values[0],
      flows['avgsize'].mean(),
      flows['duration'].sum(),
      inlan / totsize if (totsize > 0) else None,
      flows['rtt'].min(),
      flows['time'].min(),
      totsize,
      flows['iframe'].mean(),
      flows['cat'].values[0] if flows['service'].values[0] is not None else "unknown",
      flows['service'].values[0] if flows['service'].values[0] is not None else "unknown",
      flows['app'].values[0] if flows['service'].values[0] is not None else "unknown", ]
   table.append(row)

collapsed_flows = pandas.DataFrame(table,
columns=['proto','macsrc','ipsrc','ipdst','portsrc','portdst','tproto','avgsize','duration','inlan',
'rtt','time','totsize','iframe','cat','service','app'],)
# output on .csv
collapsed_flows.to_csv(dest_file + '.csv', index=None)
```

4.2. Feature Engineering

In this section we are going to show the methodology that brought us to obtain the different sample representations used to feed the Deep Neural Network.

We will start with the algorithm that reads from the .csv traffic file and obtains the different users IP destination matrices, saving the result in hdf5 format.

Then the Autoencoder implementation is shown, along with a brief overview of the MACAddressGroups class, which will come in handy also in the Deep Neural Network development.

Finally, we will take a look at the process of getting, from the .csv traffic, the service distribution vectors, presenting also the Traffic class.

4.2.1 IP destination matrix

As explained in section 3.2.1, an IP destination matrix represents a list of IP endpoints contacted by a user during a certain time interval. More precisely, for each day of capture (remember that the daily traffic is stored, after the data collecting phase, in different .csv files), the algorithm loops through the dataset in 5-minutes time increments.

As it is possible to notice in the following code snippet, DataFrame indexing is used in order to filter out from the original DataFrame dataset all the flows not falling in that 5-minutes timespan.

For each 5-minutes interval detected in this way, the algorithm builds a new DataFrame object, which depicts a table where a row represents a host identified by a MAC address while a column an IP destination endpoint; a single cell is then interpreted as the number of times a certain user generated a flow towards a specific IP destination in that time interval.

Then, for each user in that DataFrame, the actual IP matrix is built, detecting the coordinates in the red, green and blue channel (and their values) like it was shown in 3.2.1.

Finally, the image is saved in an hdf5 file together with all the other images from other users and time intervals.

```
#[...] Open a new hdf5 file by means of Python's h5py library
# Read the flows from a .csv file and take just the needed columns
flows_data = pandas.read_csv(filename, low_memory=False)
#[...] Define capture_start_time and capture_end_time on top of info in flows_data
# How big the time iinterval is
time_interval = 60 * minutes

#============================================================
# Loop through all the time intervals
#============================================================
#[...]
time_user_data = pandas.DataFrame()

# i represents the start of the time interval
i = capture_start_time
# loop until in this nnetwork traffic capture time range
while i < capture_end_time:

   # interval end is the end of this time interval
   interval_end = i + time_interval
   # put in this time interval the flows that began in it (so with a start_time
   # included in the interval) and/or ended in it (so with an end_time
   # included in the interval)
   begin_after_start = flows_data['start_time'] >= time.localtime(i)
   begin_before_end = flows_data['start_time'] <= time.localtime(interval_end)
   stop_after_start = flows_data['end_time'] >= time.localtime(i)
   stop_before_end = flows_data['end_time'] <= time.localtime(interval_end)

   # apply the filters
   tfiltered_flows = flows_data[(begin_after_start & begin_before_end) |
                           (stop_after_start & stop_before_end)]
   #[...]
   # Obtain user_ipdst_matrix, a data structure that stores the number of times a user generated a
   # flow towards a certain IP destination during this time interval.
   # (rows = user, columns = ipdst, cell = occurrencies)
   #[...]
   # backup that dataframe
   user_ipdst_stats = user_ipdst_matrix.copy()
```

```
#==================================
# For each user finally build its IP image for this time interval
#==================================
for mac, column in user_ipdst_stats.iteritems():
   # Initialize the channel matrices
   red_channel = np.zeros(shape = (256, 256), dtype = np.uint8)
   green_channel = np.zeros(shape = (256, 256), dtype = np.uint8)
   blue_channel = np.zeros(shape = (256, 256), dtype = np.uint8)
   # Loop through all the IP of this user
   for ipdst, occurrencies in column.items():
      if (occurrencies == 0):
         continue
       # Detect the coordinates in the rgb channels
      try:
         octects = ipdst.split('.')
         rx = int(octects[0])
         ry = int(octects[1])
         gx = int(octects[1])
         gy = int(octects[2])
         bx = int(octects[2])
         by = int(octects[3])
      except:
         continue
   # Write in the cells found by means of the coordinates the number of occurrencies
   red_channel[rx][ry] = occurrencies
   green_channel[gx][gy] = occurrencies
   blue_channel[bx][by] = occurrencies
   # Format the matrix as a 256*256*3 = 196.608 elements array
   rgb_matrix = np.asarray([red_channel, green_channel, blue_channel])
   rgb_matrix = np.reshape(rgb_matrix, (256, 256, 3))
   rgb_matrix = rgb_matrix.ravel()

   #[...] Save the resulting IP matrix inside the hdf5 file

# make i be the next interval
i = interval_end
```

4.2.2 The MACAddressGroups class

The MACAddressGroups class was developed in order to easily pick arbitrary sample IP images from hdf5 files and directly feed the Deep Neural Network.

We introduce this class here because it was employed also for the Autoencoder training phase.

In order to understand the logic behind this class, it is important to understand how the images are organized inside each daily hdf5 file.

As briefly mentioned earlier in chapter 3, an hdf5 is organized in a directory-like fashion, like a file system; we can say that an hdf5 directory is called group.

The IP images were saved inside such a file divided by time intervals: each group identifies a precise 5-minutes interval and inside of it, called as the MAC address of the user who generated it, are stored the actual IP matrices relative to that interval.

The Deep Neural Network is trained one batch of data at a times; this means that we will not just feed it with the whole dataset.

The task of this class it is to organize and select, according to certain policies, samples from the dataset in order to fill the batches with actual traffic data.

It is true that Keras, the Python library that was employed to develop the Neural Networks, supplies the developers with a train() method which takes care of all the data selection process for the batches, making sure that no duplicates are picked and that the data is as heterogenous as possible; but, in order to be able to fully customize the training process, it was chosen to train by hand the Networks and hence it was necessary to develop some methodology to select IP images from the hdf5 file and create a batch.

According to this premises, the MACAddressGroups class, which is relative to just one MAC address, stores the dataset as a list of 2 elements tuples; these tuples are in the form (hdf5 dataset name, group name) and are exploited as a kind of pointers to an IP image in the hdf5.

Therefore, this class, through the __build() method, provides a set of tuples that identifies part of the data for the batch that will feed the Neural Network. It is the Network's task, as we will see in the following sections, to get the actual image from the dataset following the tuple's directions. This is important because we will not load in RAM all the actual IP images but only pointers to them, avoiding memory saturation.

```
class MACAddressGroups:
    '''
    Represents a list of groups of data for a mac address. Given the mac address and a list
    of sample references (represented as a tuple in the format: (dataset.hdf5, groupname)), it
    splits all the samples in groups of the same size. Each time an object of this kind is
    created, the data is shuffled; hence two MACAddressGroups with the same sample data will
    not have the same data grouped in the same way.
    Constructor arguments:
    @mac: the mac address relative to all the samples in this object.
    @data: a list of tuples, representing the samples references; a simple reference is built
    like this (datasetname, groupname).
    @num_of_groups: The desired amount of groups this object should contain. Every group will
    contain the same amount of sample references (except the last one that can have less than
    the others).
    @group_size: the amount of samples for each group. If group_size is zero, the system will
    calculate the group size by dividing the amount of data by the number of groups provided.
    Otherwise, if a fixed amount of data per group is asked for, every group will contain that
    number of samples; IMPORTANT: if a mac address has less than group_size * num_of_groups
    samples, some of them will be duplicated (though no duplicates will be introduced in the
    same group). Default = 0.
    '''

    def __init__(self, mac, data, num_of_groups, group_size = 0, allow_duplicates = False):

        # initialize properties from constructor parameters
        self.__mac = mac
        self.__data = data
        self.__num_of_groups = num_of_groups
        self.__allow_duplicates = allow_duplicates

        # If group_size is zero, the system will calculate the group size by dividing
        # the amount of data by the number of groups provided. Otherwise, if a fixed
        # amount of data per group is asked for, every group will contain that number of
        # samples; important: if a mac address has less than group_size * num_of_groups
        # samples, some of them will be duplicated (though no duplicates will be
        # introduced in the same group)
        if (group_size == 0):
            # calculate the amount of samples in each group (but the last)
            self.__group_size = int( len(self.__data) / self.__num_of_groups)
            # calculate the amount of samples in the last group
            c = len(self.__data) - (self.__num_of_groups * self.__group_size)
            self.__last_group_size = c if c <= self.__group_size else self.__group_size
        else:
            self.__group_size = group_size
            self.__last_group_size = group_size

        # creates a list of indexes used for shuffling purposes
        if (self.__allow_duplicates):
            self.__samples_indexes = self.__get_indices_when_duplicates_allowed()
        else:
            self.__samples_indexes = list(range(len(self.__data)))
        # build the list of groups data structure, being an array of array.
        # the outermost array represent a single group, the innermost represents all
        # the samples in the group. Each sample is a tuple in the same format at the
        # one accepted by @data.
        self.__samples_groups = self.__build()
```

```
def __build(self):
   if (not self.__allow_duplicates):
      # shuffle the indexes list. This because is faster and more performing having
      # a list of "pointers" to the data and shuffle them instead of shuffle all the data.
      shuffle(self.__samples_indexes)

   # initialize the list of groups
   sample_groups = np.zeros([self.__num_of_groups, self.__group_size], dtype=tuple)
   counter = 0
   # for each group and for each sample that should be in the group
   for group in range(self.__num_of_groups):
      for position in range(self.__group_size):
         try:
            # access the current index from the shuffled list
            index = self.__samples_indexes[counter]
            # populate this element in this group with the tuple pointed by the index
            sample_groups[group][position] = self.__data[index]
         # When this exception is raised it means we are in the last group and the
         # samples are over, but the algorithm is trying to add another element to the
         # group. We just break out of the loop, keeping in mind that the last group
         # has the free position set at 0.
         # This exception should not appera when allowing duplicates.
         except IndexError:
            break
         counter += 1

   # return to the caller the list of groups
   return sample_groups

def __get_indices_when_duplicates_allowed(self):

   indices = np.empty([self.__num_of_groups, self.__group_size], dtype=int)
   for c in range(self.__num_of_groups):
      group_indices = []
      # loop until the group does not contain unique indices (in order to avoid
      # duplicates in the same group)
      while(True):
         group_indices = np.random.randint(0, len(self.__data), self.__group_size)
         if (len(np.unique(group_indices)) == len(group_indices)):
            break
      indices[c] = group_indices
   indices = indices.flatten()
   return indices

def shuffle(self):
   '''
   Shuffle the data arranged in groups. This operation is in place and shuffle
   all the sample references in this object in different groups.
   '''
   self.__samples_groups = self.__build()

def get_groups(self):
   '''Get the list of groups contained in this object'''
   return self.__samples_groups

def get_mac(self):
   '''Get the mac that owns the samples in this object'''
   return self.__mac
```

4.2.3 Autoencoder

Like all the Deep Neural Networks developed in this project, the Autoencoder exploits Keras Python library, a high-level Neural Networks API.

It is possible to notice, like it was explained before in Chapter 3, in the AutoEncoder class constructor, how such a Neural Network is composed by two distinct components: an encoder and a decoder.

Thanks to Keras, it was possible to define the architecture of the two objects separately (by means of the methods __build_encoder() and __build_decoder()) and then link the output of the first to the input of the second.

Also, the library allows to define the architecture of the Neural Network in a really easy way, implementing the objects of type Sequential and Model.

The Sequential model is a linear stack of layers while exploiting Model objects permits the developer to define complex architectures, such as multi-output models, directed acyclic graphs or models with shared layers. For more information the reader should refer to the official Keras API documentation [9] and [10].

Once the AutoEncoder object is instantiated with the correct parameters, the train() method should be called.

This method, at first creates a MACAddressGroups object for each user from the full dataset, making sure that each MAC address has the same number of samples to put into the Network; in order to achieve this result, the MACAddressGroups objects were allowed to duplicate some samples if their relative user does not have enough images.

```
class AutoEncoder:

    def __init__(self, input_shape, x_train, y_train, x_test, y_test,
        batch_size, steps, group_size):

        self.input_shape = input_shape
        self.x = x_train
        self.y = y_train
        self.xtest = x_test
        self.ytest = y_test
        self.batch_size = batch_size
        self.steps = steps
        self.group_size = group_size

        self.__encoder = self.__build_encoder()
        self.__decoder = self.__build_decoder()

        z = Input(shape=self.input_shape)
        new_feature = self.__encoder(z)
        validity = self.__decoder(new_feature)

        self.__model = Model(z, validity)
        self.__model.compile(loss='mean_squared_error', optimizer = RMSprop(), metrics=['accuracy'])

    def __build_encoder(self):
        model = Sequential()
        model.add(Conv2D(8, (3, 3), activation='relu', padding='same'))
        model.add(MaxPooling2D(pool_size=(4, 4)))
        model.add(Conv2D(8, (3, 3), activation='relu', padding='same'))
        model.add(MaxPooling2D(pool_size=(4, 4)))
        model.add(Conv2D(3, (3, 3), activation='relu', padding='same'))

        input_tensor = Input(shape=self.input_shape)
        output_tensor = model(input_tensor)
        return Model(input_tensor, output_tensor)

    def __build_decoder(self):
        model = Sequential()
        model.add(Conv2D(3, (3, 3), activation='relu', padding='same'))
        model.add(UpSampling2D((4, 4)))
        model.add(Conv2D(8, (3, 3), activation='relu', padding='same'))
        model.add(UpSampling2D((4, 4)))
        model.add(Conv2D(3, (3, 3), activation='relu', padding='same'))

        encoder_out_shape = self.__encoder.get_output_shape_at(-1)[1:]
        input_tensor = Input(shape=encoder_out_shape)
        output_tensor = model(input_tensor)
        return Model(input_tensor, output_tensor)

    def __get_batch_from_all_references(self, xref):
        xref = [item for sublist in xref for item in sublist]
        batch = np.empty(shape=(len(xref),)+self.input_shape, dtype=int)

        for c, reference in enumerate(xref):
            dset = h5py.File(reference[0], 'r')
            groupname = reference[1]
            filename = reference[2]
            flatten_matrix = np.asarray( dset[groupname][filename][:] )
            matrix = np.reshape(flatten_matrix, self.input_shape)
            batch[c] = matrix
        return batch
```

```
def train(self, N = 5, threshold=0.995):

   # [...] Initialize variables
   labels = np.unique(self.y)
   macs_groups = np.empty(len(labels), dtype=object)

   print("Training with:")
   print(len(np.unique(self.y)), "different macs")
   print(self.group_size, "samples for each mac in each batch")
   print("Output shape of encoder:", self.__encoder.get_output_shape_at(-1))

   for c, mac in enumerate(labels):

      thismac_x = []
      for i, y in enumerate(self.y):
         if y == mac:
            thismac_x.append(self.x[i])

      mac_group = MACAddressGroups(mac=mac, data=thismac_x, num_of_groups=1,
                                   group_size=self.group_size, allow_duplicates=True)
      macs_groups[c] = mac_group

   for step in range(self.steps):

      all_macs_sample_references = []
      for this_mac_g in macs_groups:

         refs = this_mac_g.get_groups()[0, :]
         all_macs_sample_references.append(refs)

      train_batch = self.__get_batch_from_all_references(all_macs_sample_references)
      test_batch = self.__get_batch(self.xtest)

      loss = self.__model.train_on_batch(train_batch, train_batch)
      scores = self.__model.test_on_batch(test_batch, test_batch)

      print("step", step+1, "[loss: ", loss[0], " acc: ", 100*loss[1], "] -", scores)

      # record train and test set losses and accuracies for each training epoch
      train_score_evol.append(loss)
      test_score_evol.append(scores)

      # updates the max accuracy obtained on the test set, used for early stopping
      if (loss[0] <= min_training_loss):
         min_training_loss = loss[0]
         epochs_not_improving_acc = 0
      else:
         epochs_not_improving_acc += 1

      # early stopping strategy
      good_test_score = scores[1] > threshold
      not_improving = epochs_not_improving_acc > N
      if ( not_improving and good_test_score ):
         print("Early stopping, with min train loss being", min_training_loss)
         break
```

Then, for a prefixed number of iterations (self.steps in the code), the Autoencoder is trained batch by batch.

This is achieved by feeding the Neural Network, each iteration, with groups of samples from each user; the MACAddressGroups objects ensure that the number of samples for each MAC is the same (however accepting some duplicates for the less sample-rich users).

In order to get the actual IP images from the MACAddressGroups tuples, the __get_batch() and __get_batch_from_all_references() methods are exploited.

Their implementation provides a way to read the image tuple pointers and detect the actual IP matrix from the correct hdf5 dataset and group.

Finally, the actual instantiation and training of the Autoencoder is shown in the snippet below.

```
# Split the dataset in training and test set 75/25
x_train, x_test, y_train, y_test = train_test_split(dataset_300, labels_300, test_size=0.25,
                                                    random_state=0)

ae = AutoEncoder(input_shape=(256,256,3),x_train=x_train, y_train=y_train, x_test=x_test,
                 y_test=y_test, batch_size=200, steps=1000, group_size=5)
ae.train()

ae.save_encoder("encoder.h5")
ae.save_decoder("decoder.h5")
```

After this step, the trained Autoencoder is saved on disk in order to avoid the training phase each times the Autoencoder is needed.

The h5 save file, that is obtained thanks to save_encoder() and save_decoder() methods (not shown in the code above for sake of briefness) exploiting Keras API, includes the model architecture and weights allowing to load it and use it in future.

4.2.4 Service distribution vectors

After the description of some of the most interesting parts about the achievement of IP destination matrices and the Autoencoder to get their compressed representations, in this section we are going to present the implementation of the service distribution vectors and how they were obtained.

Let's start with presenting the Traffic class. This class serves as an envelope of the captured traffic currently stored in the daily .csv files.

This type of object was designed with the aim of reading traffic from .csv in a comfortable and easy way, other than perform common tasks on them, like, as we will see later, calculating the user's traffic distribution on certain attributes.

When a Traffic object is created, it requires the .csv filename along with a MAC list as constructor arguments. Internally, the constructor uses pandas.read_csv() method to save the traffic in its rawData property; moreover, if a not empty list of MAC addresses was provided as second argument, the constructor will filter out from rawData all the flows belonging to users not present in that list.

What makes a Traffic object really useful, is the possibility to combine it with an arbitrary number of other Traffic instances: in this way we can end up with just one Traffic object containing all the captured flows, ready to be the base for our operations.

In this case, Traffic objects were exploited only for getting service distribution vectors, while the algorithm to obtain IP matrices does not use this abstraction. This happened because the Traffic class was developed after having obtained the IP images, as it was used, as shown before, a different method of collecting the data from the daily .csv datasets.

In a future iteration of this work, Traffic could surely be employed in order to create a bridge from .csv files to IP matrices, for example by creating a Traffic method that, acting on its rawData property, writes the images in the usual hdf5 files.

As a matter of fact, in order to obtain the service distribution vectors, a Traffic object offers the calculate_permac_stat() method which automatically saves distribution vectors in new .csv files.

```
class Traffic:

   def __init__(self, data, mac_filter=[]):
      # [...] Put in variable date the capture day obtained from the .csv filename
      self.date = date.groups()[0]
      self.rawData = pandas.read_csv(data)

      if (len(mac_filter)>0):
         self.set_mac_filter(mac_filter)
      else:
         self.macAddresses = self.rawData['macsrc'].unique()

   def set_mac_filter(self, mac_list):
      '''
      Updates the properties rawData and macAddresses according to the mac_list filter
      '''
      for mac in self.rawData['macsrc']:
         # [...] Create a boolean mask (mac_filter) that will tell which mac address should be kept
      self.rawData = self.rawData[mac_filter]
      self.macAddresses = self.rawData['macsrc'].unique()


   def join(self, traffic):
      '''
      Join this Traffic object with another one passed as argument
      '''
      if not isinstance(traffic, Traffic):
         print("Can join only object of type Traffic")
         return
      else:
         # [...] Updates also self.date property
         self.rawData = self.rawData.append(traffic.rawData)
         self.macAddresses = self.rawData['macsrc'].unique()


   def joinn(self, traffics):
      '''
      Join this Traffic object with a list of Traffic objects passed as argument
      '''
      if not isinstance(traffics, list):
         print("Use joinn method only to append more than one Traffic objects")
         return
      else:
         for t in traffics:
            self.join(t)
```

```
def calculate_permac_stat(self, column, interval):
    '''
    Calculate, for each MAC, the traffic statistical distribution among all the values that
    the parameter column can assume; then save this information in a .csv file.
    '''
    #============================================================
    # Preparing data
    #============================================================
    # get all the different values in the dataset for the selected columns; append then two extra
    columns
    # 'mac' and 'time' that will be used in the final dataframe containing user's distributions
    if isinstance(column, list):
        column_values = self.rawData[column].drop_duplicates()
        d = { v:'mac' for v in column}
        t = { v:'time' for v in column}
        column_values = column_values.append(d, ignore_index=True)
        column_values = column_values.append(t, ignore_index=True)
    else:
        column_values = self.rawData[column].unique().tolist()
        column_values.insert(0, 'time')
        column_values.insert(0, 'mac')
    # initialize the data structure that will store the distributions of the current time interval
    user_matrix = pandas.DataFrame(columns=column_values)

    # [...] define start and end time of the capture and interval_time

    #============================================================
    # Loop through all traffic capture days
    #============================================================
    dates = self.date if isinstance(self.date, set) else set([self.date])
    for current_date in dates:

        # [...] define capture_start_time and capture_end_time
        #============================================================
        # Loop through all the time intervals
        #============================================================
        # i represents the start of the time interval
        i = capture_start_time
        # loop until in this nnetwork traffic capture time range
        while i < capture_end_time:

            # interval end is the end of this time interval
            interval_end = i + interval_time

            # [...] Obtain filtered_flows: just the flows that falls in this time interval
            # execute the actual calculations in this time interval
            user_matrix = self.__permac_distr(column=column, data=filtered_flows,
            distr_data=user_matrix, time=label)
            # make i be the next interval
            i = interval_end

        # Here it means all the flows from the current date have been analyzed
        # [...] Save in .csv the results of this day
        user_matrix.to_csv(outfilename)
        # clear this day information and go to the next one
        user_matrix = user_matrix[0:0]
```

```
def __permac_distr(self, column, data, distr_data, time, verbose=False):
    '''
    Calculate, for each MAC, the traffic statistical distribution among all the values that
    the parameter column can assume; distr_data the result will be returned by this function and
    needs to be a DataFrame with columns all the values that the param column che assume. Time is
    the starting time of this interval.
    '''
    # user is the macsrc of thr current user
    # flows is the list of flows belonging to the current user
    user_groups = data.groupby('macsrc')
    for user, flows in user_groups:
        count = {}
        count['mac'] = user
        count['time'] = time
        # custom_column: the current value of the 'column' generated by the current user
        # user_flows is the dataframe that lists all flows belonging to the current 'column' value
        custom_groups = flows.groupby(column)
        for custom_column, user_flows in custom_groups:
            count[custom_column]  = len(user_flows)

        distr_data = distr_data.append(count, ignore_index=True)
    return distr_data
```

The calculate_permac_stat() method accepts two arguments: a list of string, representing the rawData column names to calculate the per-MAC distributions on, and an integer representing a time interval; the interval, like in the IP matrices case, it is needed in order to understand how to group flows to calculate the distributions; as already seen in the IP images algorithm, the distributions are calculated every 5 minutes of capture.

The first argument, the string list called column, is interesting because allows to calculate the distributions on arbitrary flow properties; in this case this method was executed three times passing as first argument respectively ‘cat’, [‘cat’, ‘app’] and [‘cat’, ‘app’, ‘service’].

The implementation of this method groups all the flows (for each 5-minutes interval) by the MAC address who generated it and counts, for each MAC, which values of the interested properties (our cat, app and service) are present in the flows and how much times; after each day of capture, a new .csv file containing the distribution vectors is created and saved on disk.

Each row of these .csv files, which is nothing else than a distribution vector, will be employed as sample for the Deep Neural Network; clearly, as already explained in Chapter 3, three different Networks will be trained, each one taking a different properties configuration.

In the following snippet of code, we can see the creation of a Traffic object containing all the captured flows and the call to calculate_permac_stat() method to obtain cat distribution vectors.

```
# Create a Traffic object with all the flows from all the .csv read,
# filtering the mac addresses we don't want
tlist = []
for filename in filelist:
   t = Traffic(filename, mac_filter=macs)
   tlist.append(t)
traffic = tlist.pop(0)
traffic.joinn(tlist)

# Calculate the mac distributions on the selected columns. One column can be
# a string, more columns are passed as list of strings
traffic.calculate_permac_stat('cat', interval=5)
```

4.3. Model

In this section we are going to present the implementation of the Deep Neural Network and the following clustering algorithm.

Both these logics are enveloped and handled by the MDModel class, which has the task of building and training the Network in order to minimize the degree of similarity between users in its predictions (for more information about this step, check section 3.3) and finally applying the actual clustering algorithm.

The code that will be shown is, for sake of conciseness, relative only to the IP matrices case. The model implementation of other types of input is analogue to this one; the only different parts are how the data is gathered from the dataset and the Network's layers structure itself.

4.3.1 The MDModel class

Like AutoEncoder, shown in section 4.2.3, also this class exploits Keras library in order to build and train the Deep Neural Network.

This class shares many similarities with AutoEncoder, especially because in this case, their enveloped Neural Networks are trained on the same type of input; this means that also MDModel offers a __get_batch() method that allows to get actual IP images (in this case, of course the method returns the appropriate Network input typology) from a list of tuples.

```
class MDModel:
    '''
    Constructor arguments:
    ---
    @epochs: number of epochs to run for the training phase
    @num_of_groups: the number of groups each mac address samples list should be split into. Hence
    each group (from the same mac) will have the same amount of sample (except the last one). A
    single training batch will be a list composed by one group for each mac address. Groups from
    different mac addresses will have different amount of samples. This number of groups will be also
    the number of steps an epoch will be composed by.
    @mac_addresses: a list of all the mac address present in the system.
    @dataset_by_mac: a dictionary where its keys are the mac addresses and the related values are a
    list of sample references, represented as tuples (datasetname, groupname).
    @lr: this model's learning rate, default = 0.0001.
    @decay: this model's decay, default = 1e-6.
    '''
    def __init__(self, epochs, num_of_groups, y_train_set, trainset_by_mac,
        y_test_set, testset_by_mac, input_shape, lr=0.0001, decay=1e-6):

        # initialize properties from constructor parameters
        if (isinstance(trainset_by_mac, dict) and isinstance(testset_by_mac, dict)):
            self.x = trainset_by_mac
            self.xtest = testset_by_mac
        else:
            raise TypeError("trainset_by_mac and testset_by_mac should be a dictionary of array-likes")


        self.y = y_train_set
        self.ytest = y_test_set
        self.epochs = epochs
        self.input_shape = input_shape
        self.num_of_groups = num_of_groups
        self.num_of_groups_test = int(num_of_groups/4)
        self.lr = lr
        self.decay = decay
        self.distributions = pandas.DataFrame()
        # build the neural network model
        self.__model = self.__build_model()

    def __get_batch(self, input_map, is_trainingset=True):
        # [...] Obtain nn_input_x and nn_input_y_cat
        # nn_input_x is the list of IP matrices obtained "dereferencing" input_map
        # (use the tuples to read the actual data on the hdf5 files).
        # nn_input_y_cat is a categorical labels representation; it has the same row of
        # the nn_input_x elements identifying its MAC address.
        # This is needed because Keras wants to work with categorical labels.


        # return the list of rgb matrices and its categorical labels
        return (nn_input_x, nn_input_y_cat)
```

As it is possible to see from the class constructor, this class requires the training and test dataset represented as a dictionary, each key being a MAC address and each value being a list of tuples (the 'pointers' to the actual images), similarly to what happens in the Autoencoder case.

This means that also this class exploits the previously presents MACAddressGroups objects, which allow to create tuples data batches that can be easily converted in actual samples data batches by __get_batch() method.

As we will explain in the next sections, this class provides all the mandatory steps to get the clustering results; MDModel in fact, implements methods to train the Network, to calculate the objective function's fitness (more on section 3.3), to get the dissimilarity tables and to run the custom clustering algorithm in order to get the threshold driven clustering graph.

In the code below it is shown the architecture of the Convolutional Neural Network that is going to be trained by means of our custom MDModel's train() method.

```
def __build_model(self):
    self.num_classes = len(np.unique(self.y))
    model = Sequential()

    model.add(Conv2D(8, (3, 3), padding='same', input_shape=self.input_shape))
    model.add(BatchNormalization())
    model.add(LeakyReLU())
    model.add(Conv2D(8, (3, 3), padding='same'))
    model.add(BatchNormalization())
    model.add(LeakyReLU())
    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Conv2D(8, (3, 3), padding='same'))
    model.add(BatchNormalization())
    model.add(LeakyReLU())
    model.add(Conv2D(8, (3, 3), padding='same'))
    model.add(BatchNormalization())
    model.add(LeakyReLU())
    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Flatten())
    model.add(Dense(128))
    model.add(LeakyReLU())
    model.add(Dropout(0.33))
    model.add(Dense(self.num_classes))
    model.add(Activation('softmax'))

    opt = rmsprop(lr =self.lr, decay=self.decay)
    model.compile(loss='categorical_crossentropy',optimizer=opt,metrics=['accuracy'])
    return model
```

4.3.2 The Deep Neural Network training phase

As already anticipated, we are not going to exploit Keras built-in function fit() for Sequential and Model objects, instead we are going to implement our custom train() function.

Likewise the AutoEncoder case, this choice was due to the possibility of fully customizing the training process; namely, following this path, it was possible to calculate the objective function's fitness after each training batch and store its results, along with the relative Neural Network's weights. It was also possible to save the state of the Network in the middle of the training to revert it back whether the fitness didn't substantially improve.

In order to understand the implementation of the train() method it is important to understand the difference between a *training step* and an epoch, which was explained in section 3.3.

For convenience's sake, let's summarise again these concepts.

A *training step* consists in getting a batch of samples from each MAC address (by means of __get_batch()) and calling train_on_batch() on the Neural Network, hence updating its weights. In this context we also get the resulting Network prediction in order to build the custom confusion matrix and to calculate the objective function's fitness; all these results are recorded, including the Network's weights snapshot.

This procedure, the so-called *training step*, is repeated until all the data from the dataset is picked and put into a batch; MACAddressGroups objects ensure that this phase is correctly executed.

Instead, an epoch is composed by all the *training steps* needed to feed the Network with the whole dataset. Hence, after the last *training step* fed the model with the last batch, we can declare the epoch settled.

We can see from the following code showing the train() method that the algorithm loops this training phase on a prefixed number of epochs; usually a Neural Network is trained over the epochs until its accuracy (or other scores depending on the model's objective) touch an asymptote.

In this case an optimization algorithm was developed in order to avoid the objective function being stuck in a local minimum and not improving over the epochs. The idea is, for each epoch, to look for the *training step* that scored the best objective function's fitness. When all the epochs are over, we will end up having a number of *training steps* record, the best of each epoch.

From each one of this record, get their relative Neural Network's weights and calculate a new weights configuration obtained from their arithmetical average, injecting them in the model. Then, the algorithm recalculates the objective function's fitness with the new weights; if it is better, the weights are kept, otherwise the last *training steps*' weights are restored. Anyway, the training phase is restarted with the usual number of epochs.

In the source code, this whole phase of going through a full training phase and then restarting from the beginning with updated weights, is called gstep. The amount of gsteps to be executed is established at the beginning as well, as a train() argument.

It is possible to see in the method's body these three big loops on gsteps, epochs and *training steps*. It is also worth noticing how, at the beginning of each epoch, the MACAddressGroups object are instantiated and prepare the data batches for all the following training steps. From their constructor arguments it is possible to appreciate the different policies of batch creation that are being used (the reader may have noticed that these policies are different also from the ones used in AutoEncoder, not allowing duplicated samples).

Finally, after all the gsteps, the train() method returns the resulting fitness along with a DataFrame storing all the MAC address pairs info and dissimilarities (this is our dissimilarities table).

```
def train(self, encoded, genetic_steps=1, save=True):
   ''' Train the neural network model'''
   # [...] Initialize variables
   # [...] Build a dict of all the sample references (train+test set) and save it as all_samples
   # this references will be used in order to calculate the custom loss
   #----------
   # For the specified amount of gsteps do a full model training. Then, for each epoch, look at
   # the training step with the lowest FIT and get the respective model weights; ending up with
   # <self.epochs> number of weights, average them and inject the result in the model, then
   # restart the training. Repeat the process <genetic_steps> times.
   for gstep in range(genetic_steps):

      #================
      # For each epoch
      #================
      for epoch in range(self.epochs):

         # ------------------------------------------------------------
         # Initialize the structure that will carry the list of mac groups.
         # Each mac address has a groups list, each group contains samples references.
         # This structure is a list of all mac addresses groups list.
         macs_groups = np.empty(len(labels), dtype=object)
         # Populate the list of mac groups list
         for c, mac in enumerate(labels):
            # MACAddressGroups envelops the list of list of groups along with other parameters.
            # It handles data shuffling (for now is useless because for each epoch a new instance
            # is created)
            mac_group = MACAddressGroups(mac=mac, data=self.x[mac],
            num_of_groups=self.num_of_groups)
            macs_groups[c] = mac_group
         # Do the exact same thing with the test set but with different settings:
         # Just one group with 300 samples for each mac;
         # these 300 samples do not contain duplicates.
         macs_groups_test = np.empty(len(labels_test), dtype=object)
         for c, mac in enumerate(labels_test):
            mac_group = MACAddressGroups(mac=mac, data=all_samples[mac],
            num_of_groups=1, group_size=300)
            macs_groups_test[c] = mac_group

         #-------------------------------------------------------------
         # For each step in this epoch, feed the neural network with a batch of data composed by
         # all the mac address of the system.
         for step in range(self.num_of_groups):

            # create a dictionary of: mac => group at this step
            nn_input_map = { g.get_mac(): g.get_groups()[step,:] for g in macs_groups }
            # given the previous data structure, read in the datasets and get from there the
            # actual data as RGB matrices. The function returns the input data to feed the net as
            # a list of matrices and a list of categorical labels.
            x, y = self.__get_batch(nn_input_map)

            # --- Train the net with this batch of data and update its weights.
            loss = self.__model.train_on_batch(x, y)
            fit, df_pair = self.__custom_loss(macs_groups_test)
            # --- Stores the model weights after this step of training [...]
            weights = ModelWeights(layers = len(self.__model.layers))
            for layer in self.__model.layers:
                # [.. .] Add actual weights to ModelWeights abstraction. w is weights of layer
                weights.add_weights(w)
            # --- Update this step's information [...]
```

```
        # End for each epochs loop.
        # From here the model is trained for the specified number of epochs.

        #------------------
        # Average the best model weights obtained by the best training step for each epoch
        # first, save the last step weights, in order to recover it if needed
        # df_loss_evolution is a DataFrame that stores each training step information
        # (the epoch, its resulting weights…)
        last_genetic = df_loss_evolution.gstep == df_loss_evolution.gstep.max()
        last_epoch = df_loss_evolution.epoch == (self.epochs - 1)
        last_step = df_loss_evolution.step == df_loss_evolution.step.max()
        df_last_step = df_loss_evolution[last_genetic & last_epoch & last_step]
        last_weights = df_last_step.weights[0]

        # get each epoch's best weights
        df_loss_evol_last = df_loss_evolution[last_genetic]
        best_weights = np.empty(shape=(self.epochs,), dtype=object)
        for curr_epoch, groups in df_loss_evol_last.groupby('epoch'):
            gdf = pandas.DataFrame(groups, columns=['gstep','epoch','step','fit','weights'])
            maxfit_group = gdf[ gdf.fit == gdf.fit.min() ]
            maxfit_weights = maxfit_group.weights[0]
            best_weights[curr_epoch] = maxfit_weights

        # average these weights
        first = best_weights[0]
        others = best_weights[1:]
        first = first.average_weights(others)

        # inject the resulting weights in the model and restart the training
        for l, layer in enumerate(self.__model.layers):
            layer.set_weights(first.raw_weights[l])

        # Calculate the custom fit on this new weighted model and store its info
        f, pair = self.__custom_loss(macs_groups_test)
        # [...] Store this result in df_loss_evolution
        # If the fit calculated on the avergaed weights is lower than the one obtained by
        # the last step of training, restore the weights
        if (f > fit):
            f = fit
            for l, layer in enumerate(self.__model.layers):
                layer.set_weights(last_weights.raw_weights[l])
            print("Restore last training step's weights")
    # ------------------------
    # END OF TRAINING PHASE
    # ------------------------
    # [...] self.distributions stores all the info about the MAC pairs. It is the data
    # sructure used to run the clustering algorithm. In order to create the
    # self.distribution DataFrame it was necessary reading the info on df_pair, obtained
    # through the __custom_loss() method.
    return f, self.distributions
```

In the following snippet of code is reported the __custom_loss() method implementation which, following the procedure shown in section 3.3, calculates the objective function's fitness and the dissimilarities table for this training step.

```
def __prepare_misdistributions(self, macs_groups):
    ''' Calculate and return the custom confusion matrix as a DataFrame'''
    mac_miss_distributions_avg = []
    macs = []
    for mac_g in macs_groups:
        mac = mac_g.get_mac()
        group = mac_g.get_groups()[0]
        mac_batch = np.empty(shape= (len(group),) + self.input_shape )
        # <group> is the list of tuples of this training step for the current MAC
        for c,element in enumerate(group):
            dsetname = element[0]
            groupname = element[1]
            dset = h5py.File(dsetname, 'r')
            plain_matrix = np.asarray(dset[groupname][mac][:])
            rgb_matrix = np.reshape(plain_matrix, self.input_shape)
            mac_batch[c] = rgb_matrix
            dset.close()
        # for each MAC obtain its row in the custom confusion matrix from the model prediction
        prediction = self.__model.predict(mac_batch)
        prediction = prediction.sum(axis=0)
        mac_miss_distributions_avg.append(prediction/sum(prediction))
        macs.append(mac)
    miss_distributions_df = pandas.DataFrame({'distr': mac_miss_distributions_avg,'mac': macs })
    return miss_distributions_df

def __custom_loss(self, macs_groups):
    ''' Calculate fitness and return the DataFrame representing the dissimilarities table'''
    misdistributions = self.__prepare_misdistributions(macs_groups)
    # <pairs> is a list of all the possible MAC pairs
    pairs = list(itertools.combinations(misdistributions['mac'], 2))
    fit = len(pairs)
    labels = np.unique(self.y)
    # [...] obtain labels_map, that list the MAC addresses represented as their id
    # [...] init pair_0, pair_1, id_0, id_1 and sub as empty lists.

    # for each MAC pair
    for pair in pairs:
        pair_0.append(pair[0])
        pair_1.append(pair[1])
        id_0.append(labels_map[pair[0]])
        id_1.append(labels_map[pair[1]])

        pair_df = misdistributions[misdistributions['mac'].isin(pair)]
        pair_distr = [distr for distr in pair_df['distr']]
        # subtract the two MACs' row from the custom confusion matrix.
        difference = np.subtract(pair_distr[0], pair_distr[1])

        # update the global fitness with this pair's contribute
        score = 0
        for val in difference:
            score += abs(val)
        sub.append(score / 2)
        fit -= (score / 2)

    # [...] Calculate df_pair, a DataFrame that stores info about a MAC pair and its difference
    return fit, df_pair
```

4.3.3 The Clustering Algorithm

After the training phase is done, we are sure that with the selected configuration we cannot reach a lower objective function's fitness, meaning that we minimised the global similarity degree among all the users.

Our custom train() method returned a dissimilarities table represented as a DataFrame: now it is time to exploit this data structure as input for the clustering algorithm.

As better explained in section 3.3, in order to proceed with the clustering phase, we need a value that will act as a threshold in order to establish whether two MAC addresses should be placed in the same cluster.

MDModel's save_clusters() method, can be used, after the specification of a threshold value, to retrieve a clustering configuration; in order to be executed, it needs also a dissimilarities table formatted like the DataFrame returned by the train() method. However, if save_clusters() is executed after train(), there is no need of this second parameter because the dissimilarities table is already saved in self.distributions property.

In order to get the clustering configurations plots shown in Chapter 3, we just need to perform this step sequentially, each time specifying a different threshold value.

Let's now analyse the actual implementation of the save_clusters() method; the idea is to group together all the users with a dissimilarity score less than the selected threshold.

It was chosen to represent this kind of information as a dictionary, being the key-value pair represented as MAC-set(MAC) pair; all the MAC addresses form its keys, while the data structure's values are the set of relative cluster companions.

```
    def __get_clusters(self, threshold, df_pair_evol_last=pandas.DataFrame()):

        # <df_pair_evol_last> represents the dissimilarities table
        if (df_pair_evol_last.empty):
            df_pair_evol_last = self.distributions
        labels = np.unique(self.y)

        # Create a dict of MAC -> set of all its cluster companioins
        sims = { k : set() for k in range(len(labels))}

        # --- Cluster all similar labels
        for idx, subs in enumerate(df_pair_evol_last.subtracion):
            id_0 = int(df_pair_evol_last.id_0[idx])
            id_1 = int(df_pair_evol_last.id_1[idx])
            # If the MAC pair dissimilarity score < threshold, put MAC1 in MAC0 set
            if (subs <= threshold):
                sims[id_0].add(id_1)
        # add itself in its own set
        for key in sims:
            sims[key].add(key)

        # count how many elements has been added (for duplicate detection)
        count = 0
        for key in sims:
            if (sims[key] is not None):
                count += len(sims[key])

        # --- Compare each set with each other, looking for inconsistencies and duplicates
        while count != len(labels):
            for key1 in sims:
                if (sims[key1] is None):
                    continue
                for key2 in sims:
                    if (key1 == key2 or sims[key2] is None):
                        continue

                    # check whether the two sets have elements in common
                    intersec = sims[key1].intersection(sims[key2])
                    if (len(intersec) > 0):
                        # if so merge them and empty the second set
                        sims[key1] = sims[key1].union(sims[key2])
                        sims[key2] = None

            # [...] recalculate <count> like above
        # do it until each labels appear just once


        # [...] recalculate <count> like above
        assert count == len(labels)
        return sims

def save_clusters(self, threshold, save=True, filename ="", df_pair_evol_last=pandas.DataFrame()):

    if (df_pair_evol_last.empty):
        df_pair_evol_last = self.distributions

    cluster_map = self.__get_clusters(threshold, df_pair_evol_last)
    return cluster_map
```

The dictionary, however, is built in order not to carry duplicated MACs in all the sets, forcing each user to be referenced just one among the dictionary values.

For instance, in a situation where just 6 MAC addresses, denoted with an index from 0 to 5, are present, this could be one possible legal dictionary format:

0. : { 0, 1, 5 }
1. : { }
2. : { 2, 3 }
3. : { }
4. : { 4 }
5. : { }

From the above representation it is easy to infer that an empty set for a MAC address, means that its user is already in another cluster, while a one-member set denotes a one-MAC cluster.

Being the dictionary formatted in this way, it is really easy to extract the detected amount of clusters and their members. As a matter of fact, it is enough to analyse just the not empty sets to notice that they correspond to the actual clusters.

Following, the code of the Neural Network training and the clustering through different thresholds.

```
shape = (16, 16, 3) if ENCODED else (256, 256, 3)

# Init the neural net
nn = MDModel(epochs=5, num_of_groups=20, y_train_set=y_train, trainset_by_mac=dset_train,
            y_test_set=y_test, testset_by_mac=dset_test, input_shape=shape)
# Train the neural net
fit, distr = nn.train(encoded=ENCODED, genetic_steps=5)

clusterings_result = {}
for counter in range(90, 0, -5):
   threshold = float(counter) / 100
   # Get the result of the clustering for the currently selected threshold
   clusters_dict = nn.save_clusters(threshold=threshold, save=False)
   # Store it
   key = "threshold_" + str(threshold)
   clusterings_result[key] = clusters_dict
# [...]
with open(fname, 'w') as outfile:
   json.dump(clusterings_result, outfile)
```

5. Results

In the previous chapters, it was shown how the developed model allows to obtain, starting from traffic data captured inside a LAN, different clustering configurations exploiting a novel methodology based on Deep Neural Networks.

As explained in Chapter 3, a use-case example of this model is to train the Deep Neural Network on an offline dataset traffic (that means previously captured and stored on disk) and then obtain an appropriate clustering configuration. After this step, into the LAN's edge router, can be inserted a software module with the task of performing an online mapping on the crossing packets between their MAC address and their behavioural class obtained from the model.

By pairing some resource sharing rules to each behavioural class, it is possible to handle online each packet that passes through the edge router.

Of course, following this approach, it is needed to take into account that the traffic training data must not be too old: it is not certain that it is possible to consider a host's behaviour constant on the long period, therefore it is advisable to periodically update the dataset and train again the Deep Neural Network (and the Autoencoder if it was chosen the configuration with encoded IP matrices as input).

Bearing in mind that the analyst's job is also choosing the most suitable configuration for his LAN, different results obtained through the 5 different Deep Neural Network training will now be shown.

We will start comparing the configurations obtained exploiting the IP matrices (encoded and not) as input for the Network; remember that the difference between the two options is that the dataset samples are always matrices representing the visited IP constellation, but in a case the matrices have a dimension of 256x256x256, in the other one the matrices are encoded through an Autoencoder up to a 16x16x16 format.

Let's start with the premise that, by using encoded IP images as input, it was not possible to reach a fitness as good as the one obtained through the original input. This fact is probably due to a loss of information that occurred during the encoding process even though the Autoencoder showed very high accuracy; in particular, a fitness score of 133 was obtained utilising the original IP matrices, while 298 exploiting the reduced ones.

This disparity on fitness values is underlined in the two graphs, figure 19 and 20, that show the resulting configurations. Having a very low average intra-host similarity, in the first solution's plot it is really easy to notice how, yet at very high thresholds, the MACs become distinguishable and pop out of their clusters.

At a dissimilarity threshold of 0.7, half of the hosts have already formed a one-MAC cluster, while, in order to get the same number of lonely users, the encoded solution needs to drive the threshold down to 0.25.

Although at first sight the results of these two solutions may seem very different, if studied with more attention they show many similarities in how the users are grouped.

Many of the clusters detected by the first solution are present also in the second one, of course at different threshold values because of the fitness disparity, like:

- The cluster containing the MAC addresses 3e:8a:ab:e2:86:2d and 46:d8:e3:df:57:68 is suddenly detected at threshold 0.9 in both cases.
- The cluster with the hosts 68:14:01:53:3c:7a, 90:1b:0e:4c:37:ab and d0:17:c2:9f:d0:90.
- The users 0c:cb:85:61:6f:47 and 0e:cb:85:a1:06:39 that at higher thresholds join a cluster together with the previous 3 hosts.
- The host c4:6e:1f:bf:74:f7 that from threshold 0.9 is always alone in a one-MAC cluster in both solutions.

It is logical that these two clustering results are similar, being the Deep Neural Networks both trained on the same information.

Another comparison that it is possible to perform, is the one between the three application-layer solutions; the ones that consider the flow fields cat, (cat, app) and (cat, app, service).

Differently from the previous case, all the three Neural Networks have reached a comparable fitness, fluctuating between 300 and 380; as a matter of fact, as it is possible to see from their plots (figure 21, 22 and 23), there is less difference in the 'shape' of their configurations.

Many hosts are in fact clustered in the same way in all the three solutions; some examples are:

- The MAC addresses b4:9d:0b:63:88:40 and ec:d0:9f:e4:12:ca.
- The lonely host 78:4f:43:6a:93:a7.
- Users 48:3c:0c:de:8c:73, 64:cc:2e:72:99:09 and d0:17:c2:dc:55:3f.

Also, in this case it is natural for these three results to be very similar, being that their Deep Neural Networks are all trained on the information from the user flows application-layer, even though each solution adds more information on top of the previous one.

If instead we compare the results from the visited IP solutions with the ones from the application-layer information, we can notice how much they are different.

The best way to do this is comparing the graph relative to encoded IP matrices (figure 20) with the ones relative to cat distribution vectors (figure 22),because they have a fitness value very similar: 298 and 301 respectively.

We can immediately notice how deep are the differences by looking at how the user c4:6e:1f:bf:74:f7 is treated in the two approaches; in all IP cases this host is always and immediately recognised and put inside a one-MAC cluster at high threshold values. To the contrary, in all the

application-layer based solutions, this host is always placed in the most numerous cluster, even at the lowest thresholds.

Clearly it is possible to find some affinities in the two approaches, however they are weaker and fewer with respect to the ones found comparing solutions of the same kind.

It is exactly for this reason that these approaches are so dissimilar, because they based their Deep Neural Networks training on two deeply different features of a user's traffic. The first takes into account only the visited IP addresses and their neighbourhood, not caring of anything else; the second one instead neglects the host's peers and focuses on the kind of information being transmitted.

The fact that these two approach typologies show dissimilar results gives the opportunity to the analyst of obtaining information on hosts' behaviour from different perspectives.

If instead it is desired to compromise between the two ideas, it is possible to calculate which clustering configuration is most similar to another configuration of the opposite approach.

Namely, by defining two set:

- IP set, composed by the solutions taking as input IP matrices and encoded IP matrices,
- Service set, composed by the solutions taking as input service distribution vectors,

it is possible to calculate which solution, threshold included, of a set is the most similar to another solution and threshold of the opposite set.

The two most similar configuration are encoded IP matrices one at threshold 0.1 (figure x) and the cat distribution vectors one with at threshold 0.1 (figure y). This because the number of one-MAC clusters is alike and some users are 'cluster companion' in both configurations, like:

- MAC addresses 3e:8a:ab:e2:86:2d and 46:d8:e3:df:57:68.

- MAC addresses 24:18:1d:1c:74:3f and ec:d0:9f:e4:12:ca.

Once the clusters are obtained, it is possible, even to help the analyst to understand which configuration is the most desirable, to analyse the dataset in order to get more insights on why some users belong to the same group and thus what that group represents.

This thing allows to predict the LAN hosts demand, in terms of network resources, on more levels; even if a clustering configuration is chosen, it does not mean that it is forbidden to exploit the information got from higher or lower thresholds in order to better understand the users' demand.

This kind of analysis can be done in two ways: from the top to the bottom and vice-versa.

Studying a clustering result from the configuration with highest threshold and going to the bottom, allows to notice how some hosts detach themselves from their cluster at certain thresholds; investigating more deeply it is possible to understand which is the reason that permitted the Neural Network to distinguish that user from the others while instead, for example, a companion of him remains in the cluster.

Analysing in the opposite sense, it is possible instead to notice 'lonely' hosts that merge themselves into groups and, analogously to before, to understand why the Deep Neural Network started to consider that host traffic similar to those in the cluster.

One thing to take into account is that a host that pops out from a cluster at very low thresholds, may be because of the presence of an anomalous IP address (or a small percentage in a service distribution vector) in his traffic; this outlier can make him suddenly look different at the Network's eyes as soon as the threshold decreases.

An example of this phenomenon may be a user that erroneously, while he is surfing the web in his usual sites, clicks on an advertisement banner generating thus a flow toward an extraneous IP

address; the Network may consider this host different only for this feature when actually he has a behaviour comparable to the other hosts' ones.

This argument is not true for hosts that pop put from their cluster at high threshold values: few outliers are not enough to 'trick' the Neural Network; in order to be left out from a cluster so high in the thresholds plot, the behaviour of such a user needs to be really dissimilar to the others.

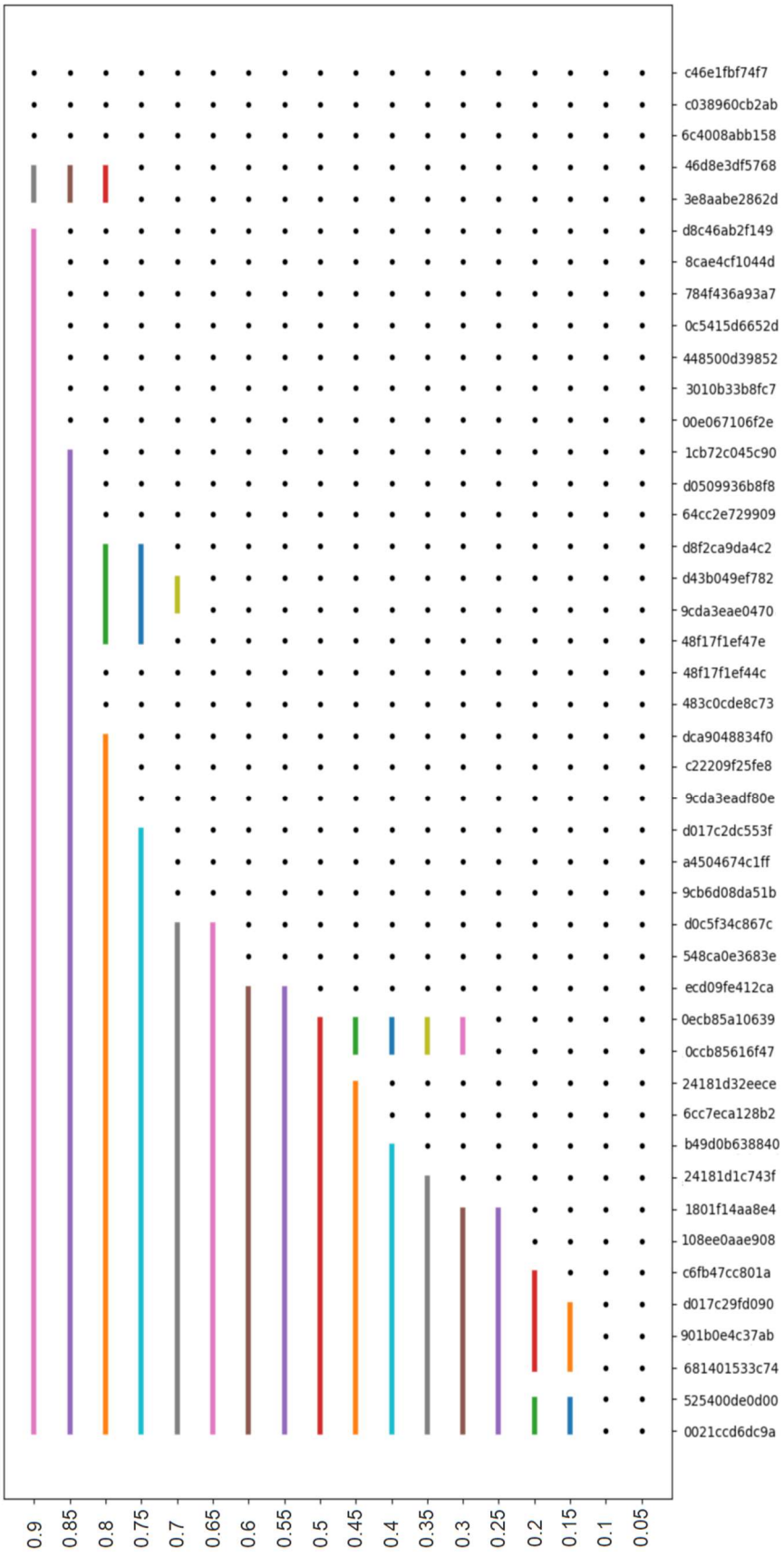

Figure 19 - Plot that shows different clustering configurations obtained training the Deep Neural Network with IP matrices.

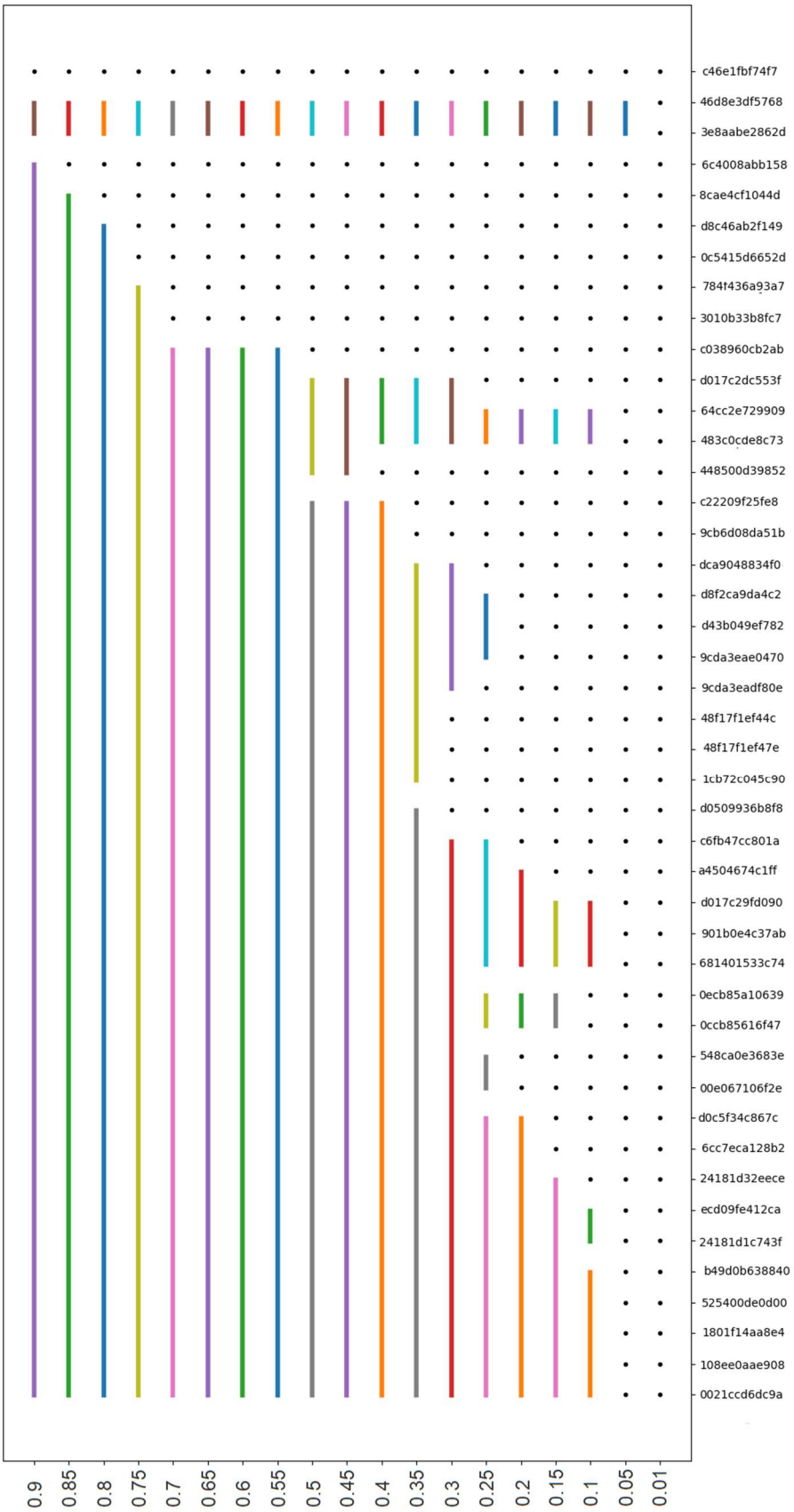

Figure 20 - Plot that shows different clustering configurations obtained training the Deep Neural Network with encoded IP matrices.

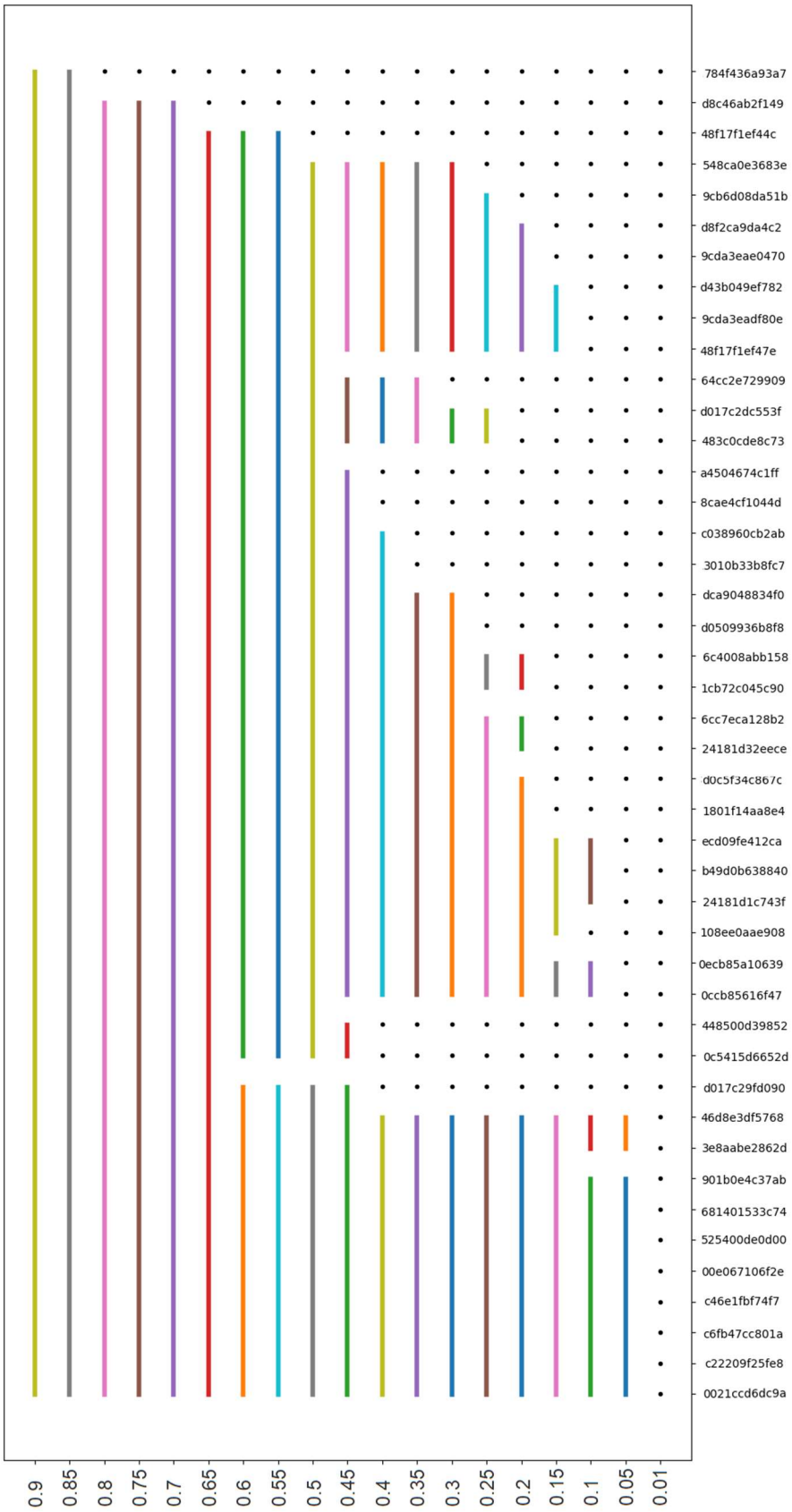

Figure 21- Plot that shows different clustering configurations obtained training the Deep Neural Network with cat distribution vectors.

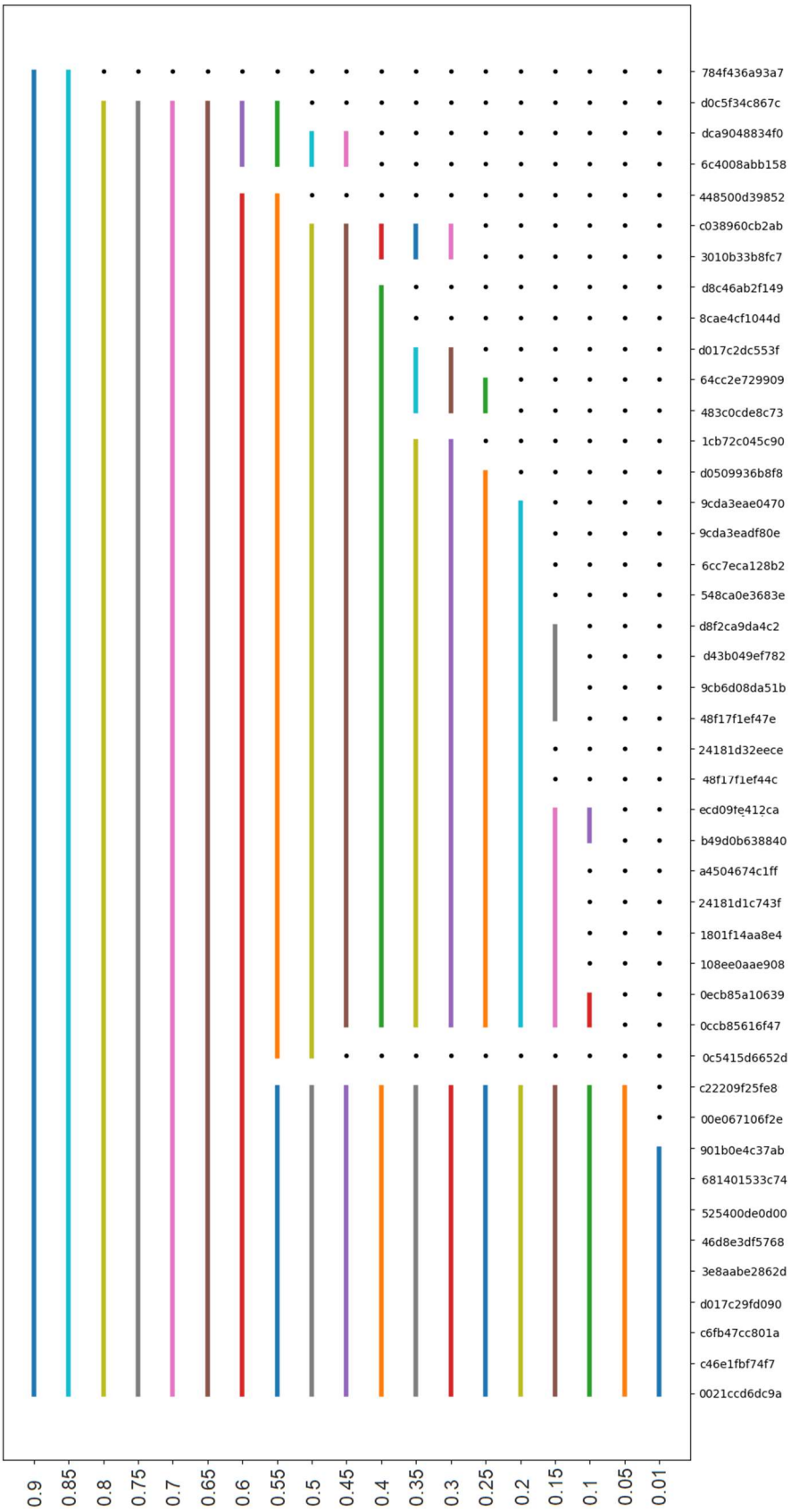

Figure 22 - Plot that shows different clustering configurations obtained training the Deep Neural Network with (cat, app) distribution vectors.

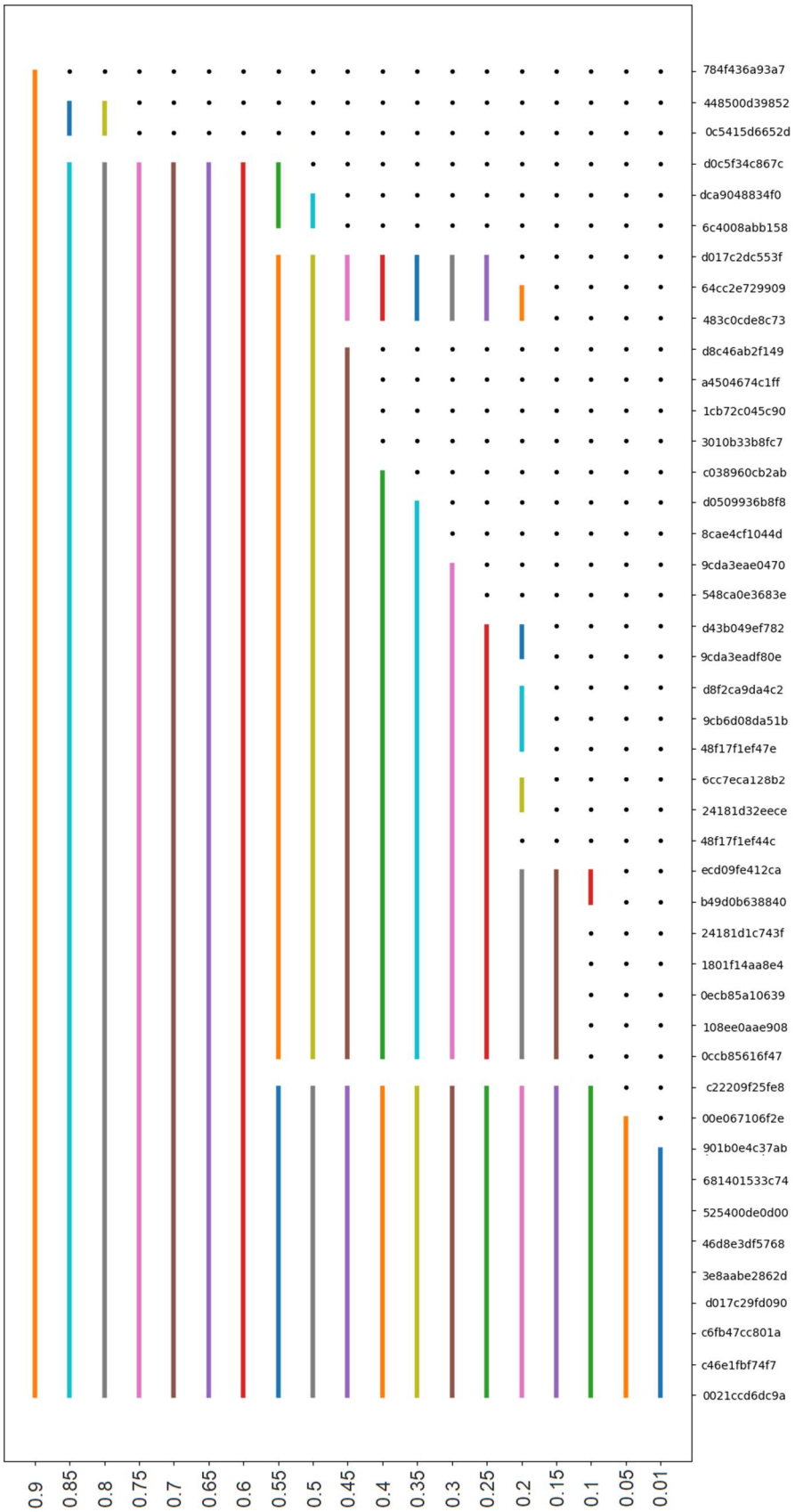

Figure 23- Plot that shows different clustering configurations obtained training the Deep Neural Network with (cat, app, service) distribution vectors.

6. Conclusions

The model obtained at the end of this project allows, through a novel methodology specifically developed, to classify users inside a LAN without a priori knowledge on their class membership.

Although a Deep Neural Network is employed in a supervised manner, the followed approach is an unsupervised one in the entirety because the classification performed by the Network is just a mean to detect features for the final clustering algorithm.

Once the clustering algorithm is run on the Neural Network's output, and therefore, after having obtained the clustering configurations graph, it is the analyst's task discerning the various possible solutions and establishing a criterion for choosing the best threshold value through the interpretation of the results.

These motivations, which caused the Deep Neural Network, trained to accent dissimilarities among users in its predictions, to obtain such a weights configuration, can be researched analysing and comparing the training data and the clustering results.

As already outlined in this document, it is sincerely advised to re-train the Network (and hence obtain new clustering configurations) with updated data traffic, especially in the case of new users, that did not previously take part to the analysis, joining the LAN.

Even in lack of this event, it is good keeping as much as possible the System updated, not so much for a model defect, as for the inability to state whether and for how long a user behaviour can be considered constant in time.

For the future of this project, it may be a good idea to investigate different user traffic training features. In this context, the analysis was limited to studying two characterizing properties of a data flow, namely the endpoint IP address and the service typology offered by the flow itself.

The investigation on other attributes like the endpoint geographical position or the amount of data moved in upload and download are surely interesting cues to consider.

Another possible path may be continuing to exploit the IP address peer information but changing its RGB matrix format to a different representation, monitoring how the clustering results change.

Finally, it may be worth putting some effort in the evolution of the clustering algorithm: the actual implementation considers subtracting the two users' distributions in order to calculate their dissimilarity score; one strategy could be using a different method to compute the probability distributions difference, like the *Kullback-Leibler divergence* with respect to both members.

Bibliography

Boutaba, R., Salahuddin, M. A., Limam, N., Ayoubi, S., Shahriar, N., Estrada-Solano, F., Caicedo, O. M. (2018). A comprehensive survey on machine learning for networking: evolution, applications and research opportunities. *Journal of Internet Services and Applications,* 16.

Mowei, W., Yong, C., Xin, W., Shihan, X., Junchen, J. (2017). Machine Learning for Networking: Workflow, Advances and Opportunities. *IEEE Network*, 32(2).

Raschka, S. (2015). Python Machine Learning, *PACKT Publishing*.

Murphy, K. P. (2012). Machine Learning, A Probabilistic Perspective, *The MIT Press*.

Trevino, A. (2016). Introduction to K-means clustering. Retrieved from https://www.datascience.com/blog/k-means-clustering.

Bock, T. What is a Dendrogram? Retrieved from https://www.displayr.com/what-is-dendrogram/.

Moise, I., Pournaras, E., Helbing, D. Density-Based Clustering. Retrieved from https://ethz.ch/content/dam/ethz/special-interest/gess/computational-social-science-dam/documents/education/Spring2015/datascience/clustering2.pdf.

Nandi, M. (2015). Density-Based Clustering. Retrieved from https://blog.dominodatalab.com/topology-and-density-based-clustering/.

Ntop, nDPI webpage, https://www.ntop.org/products/deep-packet-inspection/ndpi/.

Pogančić, M. V. (2019). On the Curse of Dimensionality. Retrieved from https://towardsdatascience.com/on-the-curse-of-dimensionality-b91a3a51268.

[1] Leng, B., Liu, J., Pan, H., Zhou, S. (2015). Topic Model Based Behaviour Modeling and Clustering Analysis for Wireless Network Users, *arXiv:1511.05618v1 [cs.SI].*

[2] De Oliveira, M. R., Valadas, R., Pacheco, A., Salvador, P. (2003). Cluster Analysis of Internet Users Based on Hourly Traffic Utilization. *First International Working Conference on Performance Modelling and Evaluation of Heterogeneous Networks*.

[3] Nogueira, A., De Oliveira, M. R., Salvador, P., Valadas, R., Pacheco, A. (2005). Classification of Internet Users using Discriminant Analysis and Neural Networks. *Next Generation Internet Networks.*

[4] Kumpulainen, P., Hätönen, K., Knuuti, O., Alapaholuoma, T. (2011). Internet Traffic Clustering Using Packet Header Information. *Joint International IMEKO Symposium*.

[5] Gu, G., Perdisci, R., Zhang, J., Lee, W. (2008). BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection. *17th USENIX Security Symposium*.

[6] Wireshark Display Filter Reference, https://www.wireshark.org/docs/dfref/.

[7] NumPy official documentation, https://numpy.org/doc/1.17/.

[8] pandas: powerful Python data analysis toolkit – pandas documentation, https://pandas.pydata.org/pandas-docs/stable/.

[9] Keras documentation – The Sequential model API, https://keras.io/models/sequential/

[10] Keras documentation – Guide to the Functional API, https://keras.io/getting-started/functional-api-guide/