Delivrable Algorithms Description: Traffic pattern evolution and unsupervised network anomaly detection ONTIC D4.2

Juliette Dromard, Philippe Owezarski, Velasco Mozo, Alberto Ordozgoiti, Bruno Vakaruk

Online Network Traffic Characterization

Deliverable
Algorithms Description

Traffic pattern evolution and unsupervised network anomaly detection

ONTIC Project
(GA number 619633)

Deliverable D4.2
Dissemination Level: PUBLIC

Authors

Dromard, Juliette
Owezarski, Philippe
CNRS

Mozo Velasco, Alberto
Ordozgoiti, Bruno
Vakaruk, Stanislav
UPM

Version

ONTIC_D4.2.2016.01.04.0.5

Version History

Previous version	Modification date	Modified by	Summary
0.1	2015.12.10	CNRS	1st draft version
0.2	2015.12.26	CNRS	Included contribution from UPM
0.3	2016.01.04	CNRS	Included second contribution of UPM
0.4	2016.01.20	CNRS	Adaptit's review
0.5	2016.01.21	CNRS	Polito's review
0.6	2016.01.28	CNRS	Included UPM's modification

Quality Assurance:

	Name
Quality Assurance Manager	**Miguel-Angel Monjas**
Reviewer #1	**Daniele Apiletti**
Reviewer #2	**Panos Georgatsos**

Table of Contents

List of figures

List of tables

1 Executive Summary

Deliverable 4.2 describes the algorithms produced to characterize online large network traffic in terms of traffic pattern evolution and unsupervised network anomaly detection. Experimental results of these algorithms are presented and discussed. These algorithms are designed to be run on the Big Data analytics system and the provisioning subsystem specified in the WP2.

These algorithms leverage large network traces and data mining techniques for helping the network administrator in its task. They offer him an insight on its network which could not have been possible with small network traces. This insight can help him improve the network's Quality of Service (QoS) and security. We can also imagine that, thanks to the analysis provided by these algorithms, automatic decisions could be taken to protect and optimize the network.

This deliverable follows deliverable 4.1 which has introduced, among others, an Unsupervised Network Anomaly Detection Algorithm (UNADA) capable of detecting network anomalies without relying on signatures training or labeled instances.

The first part of the deliverable deals with the problem of leveraging traffic patterns in order to make reliable forecasts. Driven by the goals of WP5 UC #2, which aims to build a distributed system for bandwidth assignment to proactively control network congestion, a Network Traffic Forecasting Framework (NTFF) has been developed. We have designed a forecasting procedure that is shown to reliably predict the number of open sessions crossing a network link, which can be particularly useful for computing approximate bandwidth assignments effectively. An exploratory analysis of the ONTS dataset has also been carried out, drawing useful insights on the behavior of the traffic it represents. We have conducted experiments on a sizeable sample of the ONTS dataset to validate the effectiveness of the forecasting procedure integrated in the NTFF, showing its ability to make 4-step forecasts within 2% of mean error. The conducted research and the obtained results reveal key directions to be explored in the future.

The second part of the deliverable deals with the problem of identifying, in near real-time and on large network traffic, anomalies in an unsupervised way, i.e., without previous knowledge on the anomalies. First, a Parallel version of UNADA, called PUNADA, is proposed. The obtained results show that this solution improves UNADA execution time; however it displays a limit in scalability. To overcome this issue, an Online and new Real-time Unsupervised Network Anomaly Detection Algorithm (ORUNADA) is then proposed. It is an incremental version of UNADA relying on an incremental grid clustering algorithm and a sliding-time window. This solution speeds the execution time by a factor of 100 and allows a near real-time detection while preserving the quality of anomaly identification. This part ends with an extensive study of existing unsupervised network anomaly detectors which points out some new directions to explore.

Last, the deliverable concludes with a discussion on the main challenges encountered during algorithm development and the topics of future work.

2 Acronyms

Acronym	Defined as
ARIMA	Autoregressive Integrated Moving Average
AUC	Area Under the Curve
CWT	Continuous Wavelet Transform
DBSCAN	Density-Based Spatial Clustering of Applications with Noise
DoS	Denial of Service
DWT	Discrete Wavelet Transform
EA	Evidence Accumulation
LOF	Local Outlier Factor
FP	False Positive
FPR	False Positive Rate
IGCA	Grid density-based Clustering Algorithm
IGCA	Incremental Grid density-based Clustering Algorithm
MAE	Mean Absolute Error
MAPE	Mean Absolute Percentage Error
OFAT	One Factor At a Time (OFAT).
ORUNADA	Online and Real-time Unsupervised Network Anomaly Detection Algorithm
PC	Principal Component
PCA	Principal Component Analysis
PUNADA	Parallel and Unsupervised Network Anomaly Detection Algorithm
R2L	Remote To User
ROC	Receiver Operating Characteristic
SOD	Subspace Outlier Detection
TP	True positive
TPR	True Positive Rate
U2R	User to Root Attacks

UNADA	Unsupervised Network Intrusion Detection Algorithm
WP	Working Package

3 Intended Audience

The intended audience for this deliverable includes all the members of the ONTIC project and specifically those involved in:

- WP4, as they devise the online algorithms.
- WP2, as they design the provisioning subsystem on top of which the algorithms have to run.
- WP5, as they propose use cases which take benefit of the algorithms presented in this deliverable.

Furthermore, this report could be of interest to any person working in the field of traffic pattern evolution and network anomaly detection.

4 Algorithms for Traffic Pattern Evolution

Network traffic analysis today is especially challenging because of the huge volumes of data involved. In online settings, where the available data cannot be permanently stored and must be processed and discarded straight away, an additional challenge arises. Network traffic data often fit what is known as a time series model, i.e. the data samples represent events that are consecutive. This means that the underlying distribution can change over time, even in a relatively small data set, making it especially difficult to build predictive models. Driven by the goals of WP5 UC #2, the ONTIC consortium have carried out a research process to determine whether or not it is possible to identify useful trends and patterns from the ONTS dataset, as well as to build a system that can reliably and adaptively predict them. This research has led to the development of a Network Traffic Forecasting Framework (NTFF), a system that integrates network traffic data preprocessing and modular forecasting model design. We propose a forecasting procedure to predict short and medium term traffic behavior. Up to our knowledge, there exists no open-source solution integrating these functionalities. The NTFF is available online at a git repository.

The NTFF is designed so that network managers and engineers can use these models without deep knowledge of the field. The first released version has been designed with the goal of forecasting the number of open TCP sessions crossing an ISP's core network at a given time period, which can be particularly useful in the congestion control protocols being developed in WP5. In order to proactively control network congestion, we are interested in forecasting the number of flows crossing a router link, which can be used when computing the bandwidth allocation for each session. Details on this process can be found in section 8.3 of deliverable D5.2. The techniques described below, however, can be used to forecast many different variables and could therefore be useful to tackle a variety of problems that arise in network management and engineering.

In order to successfully develop the NTFF, we addressed the problem of forecasting network-related variables. We posit that the patterns underlying network traffic present regularities that can be exploited for making reliable forecasts in the short term. We have carried out a costly preprocessing procedure, followed by visualization, revealing meaningful qualities of the captured network data that can be useful for employing the NTFF in practice. In particular, we have observed a clear periodicity in the traffic patterns and an absence of drift that can be leveraged to build long-lasting models. Afterwards, we have trained and evaluated different forecasting models in combination with judiciously chosen discrete wavelet transforms, an approach that up to our knowledge has not been sufficiently explored in the domain of network traffic. Since WP4 activities revolve around algorithms for streaming data, we have focused our efforts on building models that can be feasibly applied in the online setting. For instance, we have applied the SBLLM algorithm for training the neural network models, which is much more efficient than backpropagation, and we have combined our models with the stationary wavelet transform, which can be updated efficiently as new data arrive.

The rest of this section describes the NTFF and the proposed procedure. It also shows the results of the exploratory analysis, as well as a description of the experimental results obtained from the application of the resulting models on the ONTS data set. Our results suggest that the employed data set does indeed present patterns that can be exploited for reliable short-term forecasting. Specifically, we have found a set of techniques, which have been integrated into NTFF and will be used in the congestion control subsystem to be

developed in WP5 UC#2. We finalize this section by summarizing our main conclusions and the further research directions motivated by our results in Traffic Pattern Evolution.

4.1 NTFF: Network Traffic Forecasting Framework

The NTFF is comprised of three modules: Data Preprocessing (DP), Wavelet Decomposition (WD) and Forecasting Model Design (FMD). These modules are connected sequentially in the specified order, with the Wavelet Decomposition being optional.

The DP module can ingest large volumes of traffic traces in the pcap format, coming from either permanent storage or data streams, which are filtered and aggregated into the adequate format. The WD module decomposes the resulting time series data into the specified number of frequency components. Finally, the FMD offers the possibility of training and testing different forecasting models (ARIMA, Ridge Regression and Artificial Feedforward Neural Networks).

4.1.1 Data preprocessing (DP)

The DP module converts compressed pcap data into an adequate time-series format. First, the pcap files are aggregated into 5-tuple flows using the Tstat tool[1].

The feature filtering module developed in WP2 D2.2 is used to parse the timestamp out of the Tstat output. The flows are aggregated into 1-second slots by means of an ad-hoc Apache Spark job that we wrote to speed up this part of the process.

An overview of the whole procedure is shown in Figure 1.

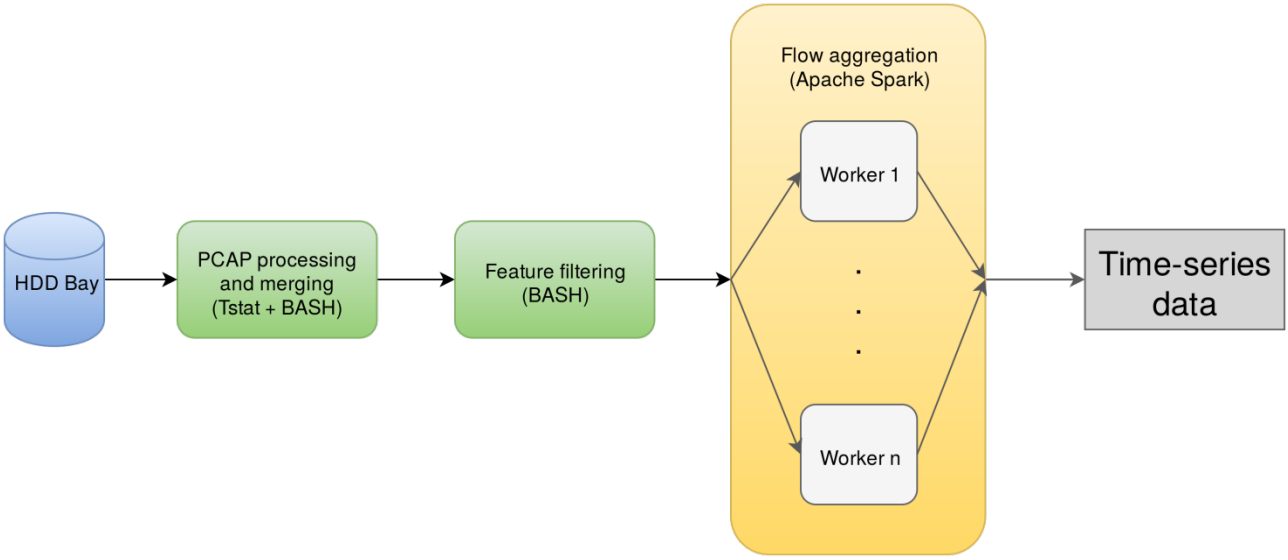

Figure 1: Data preprocessing overview

Experimentally, we have observed that the whole preprocessing takes about three days in our available hardware for one month of traffic. For that reason, we plan to leverage the availability of the enterprise-grade cloud environment during the third year to process the entirety of the dataset. The data retrieval from the HDD bays remains a bottleneck, however, which constitutes a hindrance that must be overcome.

4.1.2 Wavelet Decomposition (WD)

Wavelet transforms are a widely used tool for time series analysis and signal processing. As opposed to the Fourier transform, the wavelet transforms have the ability to build a time-frequency representation of a signal that offers very good time and frequency

[1] http://tstat.polito.it/

localization. The WD module offers an interface for decomposing an input signal using the stationary wavelet transform, in either packet or non-packet form. The levels of decomposition can be specified by the user. A detailed description of wavelets and how we have integrated them into the NTFF can be found in section 4.1.4.1.

4.1.3 Forecasting Model Design (FMD)

The FMD features an interface for training and validating different forecasting modules. We have included some of the models that generally perform best in other domains (energy consumption, gas and electricity prices, wind speed, stock exchange prices). Specifically, the NTFF offers ARIMA, Ridge Regression and Artificial Feedforward Neural networks, and methods for training all of them.

4.1.4 Proposed Forecasting Procedure

We apply the SWT along with the forecasting models in two phases: during the training phase, the time series is decomposed into *n* components[2] with the same bandwidth each. If the packet decomposition is used, then the transform is equivalent to the one described in [1]. We then train a different model for each component separately. The forecasts are also made on each component individually, which are then aggregated to obtain the final prediction. The different resulting models are shown in figures Figure 2Figure 3Figure 4 andFigure 5.

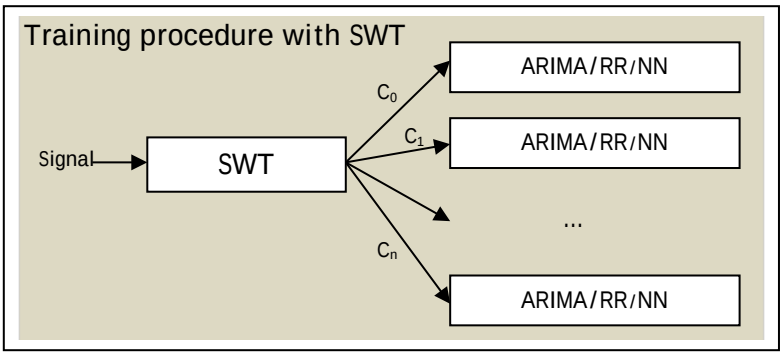

Figure 2: Training procedure with SWT

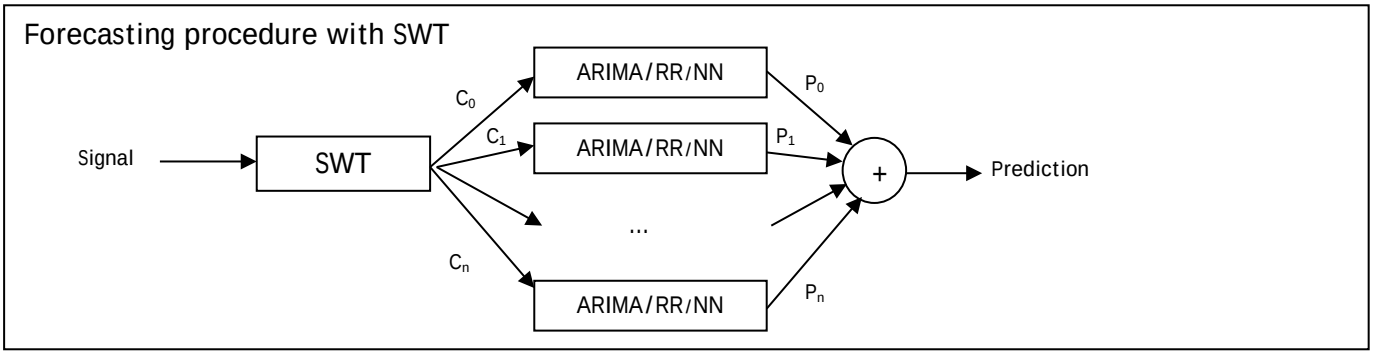

Figure 3: Forecasting procedure with SWT

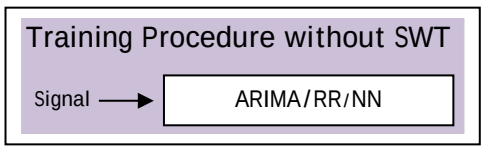

Figure 4: Training procedure without SWT

[2] In our experiments we have tested 16 components.

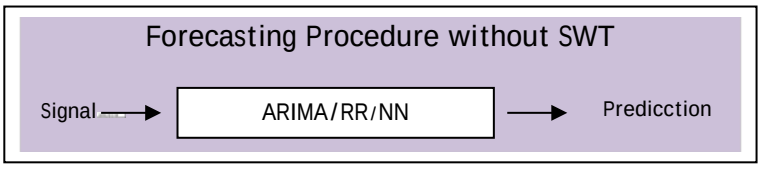

Figure 5: Forecasting procedure without SWT

For each of the available models, we have chosen the training method that best suits the problem at hand. Ridge Regression is trained with the LevenbergÌ-Marquardt method, which combines the advantages of the Newton method and gradient descent. In the case of Neural Networks, we use the SBLLM algorithm, which significantly outperforms backpropagation in terms of convergence speed. Finally, ARIMA is trained following the usual procedure.

The NTFF is open source and is available online at https://gitlab.com/ontic-wp4/NTFF. Details on how to use it are provided in Annex A.

4.1.4.1 Used Techniques

In this section, we provide a brief overview of the employed forecasting methods, namely ARIMA, Artificial Feedforward Neural Networks and Ridge Regression, which we have used in combination with the Discrete Wavelet Transform.

Discrete Wavelet Transform

The discrete wavelet transform (DWT) is a widely used tool with applications in signal processing and time series analysis. The DWT decomposes a discrete signal into various components representing different frequencies in the time domain.

Since the early 2000s, wavelet transforms have been combined with forecasting models as a preprocessing step. The continuous wavelet transform (CWT) is used to divide continuous-time functions into frequency components. As opposed to the Fourier transform, the CWT has the ability to build a time-frequency representation of a signal that offers very good time and frequency localization. The DWT can be understood as a sampled CWT, yielding sufficient information about the signal with significantly decreased computation time. By applying high-pass and low-pass filters, the DWT decomposes the input signal into different signals that correspond to subsets of the original frequency bandwidth.

There exist many different methods for performing a DWT. Among them, we have chosen the stationary wavelet transform because it can perform a transformation at time t without resorting to posterior values, which is of course a desirable property in forecasting scenarios.

Since the different frequency components behave in a "cleaner" way than the original data, the DWT constitutes an interesting technique for improving forecasting results. In order to use the DWT for making forecasts, we have conducted the following process:

1. The time series data z is decomposed into n frequency components using the DWT, that is $DWT(z) = (f_1, f_2, \dots, f_n)$
2. A signal is retrieved from each resulting component using the inverse DWT: $z_k = DWT^{-1}(0,0,\dots,f_k,\dots,0)$
3. Using the forecasting model of choice F, we make a forecast for each of the n retrieved signals: $z'_k = F(z_k)$
4. We rebuild the original signal with the forecast data point by adding up all the components: $z' = \sum_{i=1}^{n} z'_i$

The DWT presents certain issues for its efficient application to time series forecasting. Specifically, the transformation must be carried out on a time window, which is

computationally demanding. To circumvent this we use the SWT, which can be efficiently updated as each new data point arrives, a key characteristic for its application to online scenarios. This technique has been widely used for time series analysis and forecasting in different domains [2] [3].

The second issue is the choice of mother wavelet. We studied two that are perhaps the most popular ones: the Haar and the Daubechies wavelets. The former presents a problem that hindered its applicability to our domain. In particular, the bandwidth corresponding to each component is not clearly delimited, and includes components from frequencies outside the correct region. The Daubechies wavelet helps overcome this issue, providing a better isolation of each frequency component in its associated bandwidth, but presents shortcomings of its own. When the inverse transform is applied to individual components, the results present artifacts at the borders of the timeline. This is not a problem for signal reconstruction, since these artifacts cancel each other out in the transform inversion process. For forecasting, though, they do constitute an obstacle, because forecasts are of course heavily dependent on the last values of the series.

Figure 7 and Figure 8 illustrate the issue described above. We performed a SWT of three overlapping samples of our data with a Daubechies wavelet of order 20, spanning a 1000-second window each, separated by 50 seconds from the next (the first window starts at time 0, the second starts at time 50 and the third at time 100, as shown in Figure 6 Region of the original data corresponding to the wavelet components depicted in figures 7,8,9 and 10), We then applied the inverse SWT to the first individual component of each signal, as in step 2 of the procedure described above, resulting in three signals comprised of an individual frequency component each. Figure 7 shows the last 100 values of the first component of each transformed sample respectively (the time interval in the black rectangle region shown in Figure 6). The inconsistencies between the three plots at the last values of the blue and the red lines are evidence of the problem in question: they correspond to the same time period but are different. Figure 8 is equivalent, but shows the results for the third frequency component instead of the first. Figures 9 and 10 show the wavelet transform components corresponding to the same signals and period, but using the Haar wavelet instead. The artifacts introduced by the Daubechies wavelet are not present.

This problem led us to choose the Haar wavelet for our preliminary models. Tests with Daubechies wavelets of different orders and other forecasting schemes remain a part of our intended future work.

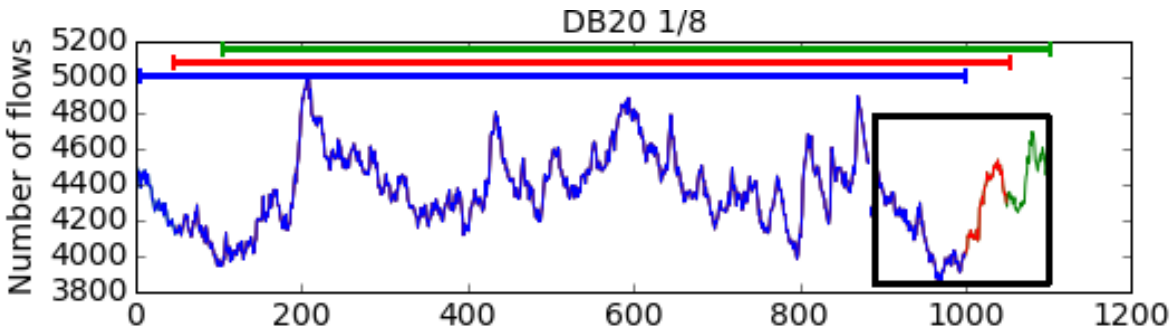

Figure 6 Region of the original data corresponding to the wavelet components depicted in figures 7, 8, 9 and 10

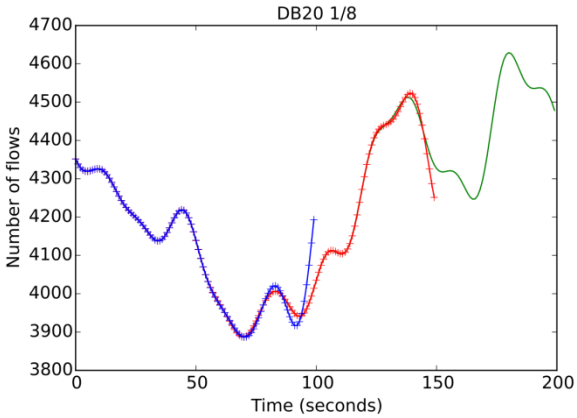

Figure 7 Inverse DWT of the first component of a Daubechies SWT for three overlapping time windows

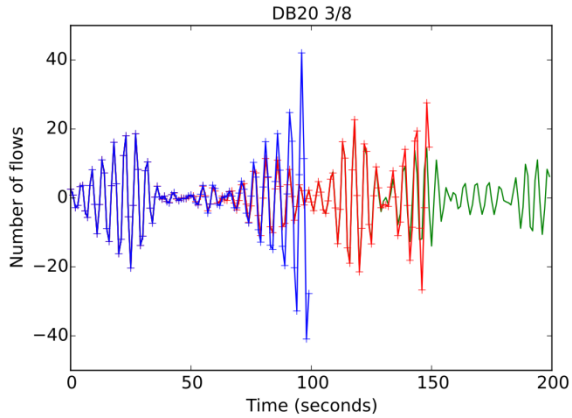

Figure 8 Inverse DWT of the third component of a Daubechies SWT for three overlapping time windows

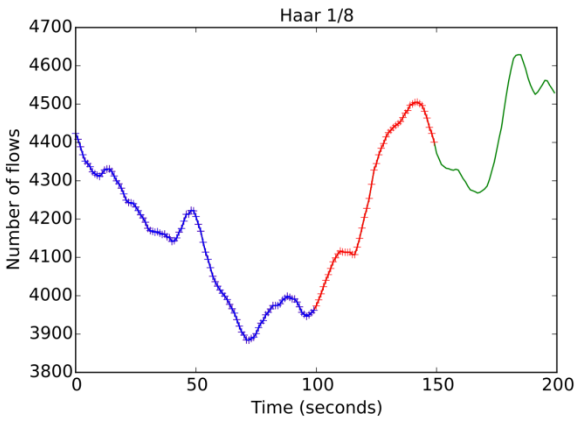

Figure 9 Inverse DWT of the first component of a Haar SWT for three overlapping time windows

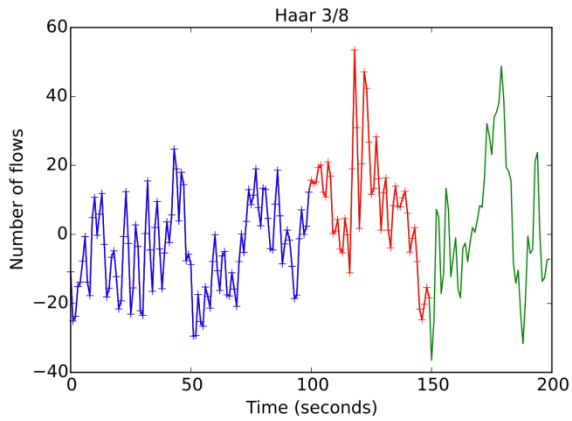

Figure 10 Inverse DWT of the third component of a Haar SWT for three overlapping time windows

Linear Regression

Linear regression is one of the most basic tools for statistical learning [2]. Given a set of n-dimensional data points, it fits the linear function that best predicts the target value. As any other regression model, linear regression can be employed as a component of forecasting systems.

Linear regression can be trained using objective functions with regularization penalties in order to confer the model with robustness in the face of outliers and noise. Since we do not aim for a sparse model, we have chosen L2-norm regularization, often referred to as Tikhonov regularization or simply ridge regression. For training the model, we chose the Levenberg–Marquardt method [4], which resembles either the Gauss-Newton algorithm or gradient descent, depending on the approximation error from the last iteration.

ARIMA

Autoregressive integrated moving average (ARIMA) models are a widely used statistical tool for time series analysis and forecasting. These models combine an autoregressive model that represents a linear dependency between the current and lagged data, according to a certain difference, values of the time series, and a moving average model that represents the errors of current and previous values from the moving mean of the series. They are suitable for capturing non-stationary and seasonal behaviors, features that are typically exhibited in telecom environments, but at larger time-scales and with heteroscedasticity. ARIMA models

and variants have recently been applied for making forecasts in various fields including network management [5] [3].

Artificial Neural Networks

Artificial neural networks surfaced in the decade of the 1980s as a powerful model for machine learning, given their ability to learn any type of function. The computational complexity of the available training algorithms, the difficulty of designing an adequate architecture and the shortcomings of the training process meant a decrease in their popularity during the 1990s. However, recent advances have brought them back to the spotlight and are now applied with success to a variety of problems, including forecasting [[6]. Neural networks: was selected to compare in order to contrast with the ARIMA as a non-linear model as was done in [3] [2]. But here, we used SBLLM [7], as a faster method for Neural Networks training.

4.1.5 Experiments

We have performed an extensive set of experiments on the ONTS dataset in order to validate the different components of the NTFF. We have first carried out an exploratory analysis of the ONTS dataset, in order to reveal patterns, trends and regularities that could be useful for properly training the models, as well and to gain insights on the behavior of the traffic. Below we provide details on said analysis, as well as the experimental setup and results.

4.1.5.1 Exploratory analysis

The success of forecasting models is heavily dependent on the nature of the processed data. For instance, a sequence that takes values uniformly at random in a fixed interval is best estimated by a constant function, which is certainly of little use. Therefore, for a forecasting system to be valuable, the target dataset must have certain qualities. In order to determine whether the ONTS dataset presents said qualities or not, we processed a sizeable sample of it to obtain a time-series representation of the number of open TCP sessions at consecutive time intervals, which is a key value for the goals of use case #2.

The required preprocessing (which is described in detail below) takes around three days on our available hardware for a one-month-long sample, which is why we employed a portion this long for our first experiments. Specifically, we processed the captures corresponding to months of April and May 2015.

We analyzed the resulting data with two purposes. First, we wanted to assess the applicability of the envisaged models. Secondly, we carried out an exploratory analysis process to gain insights on the ONTS dataset. Our most significant conclusions are explained below.

Figure 11 and Figure 12 provide a clear overview of how the processed traffic behaves. The plots represent the time series data (i.e. the number of open TCP sessions at one-second intervals) for the months of April and May. The regularity of the data across different days is clear, with weekends and holidays showing much more moderate activity in general.

There are some irregularities that are worth mentioning.

- The period from 2 April to 6 April is a holiday period in Spain. Therefore, these dates present weekend-like behavior.

- At some points in time, the data presents clearly visible activity peaks (e.g. 16 and 28 April). The cause of these anomalies is not clear, but they are likely to represent denial of service attacks.
- Some periods present sustained, high activity (e.g. 11 April). After consultation with the SATEC team, these probably correspond to tests conducted by Interhost-SATEC employees.
- There are missing data, which correspond to periods in which the capture system was down.

Figure 13 shows the period from Saturday the 23rd to Friday the 30th of May. Figures 14 and 15 show the data corresponding to a weekday (25 May) and a weekend day (23 May) respectively.

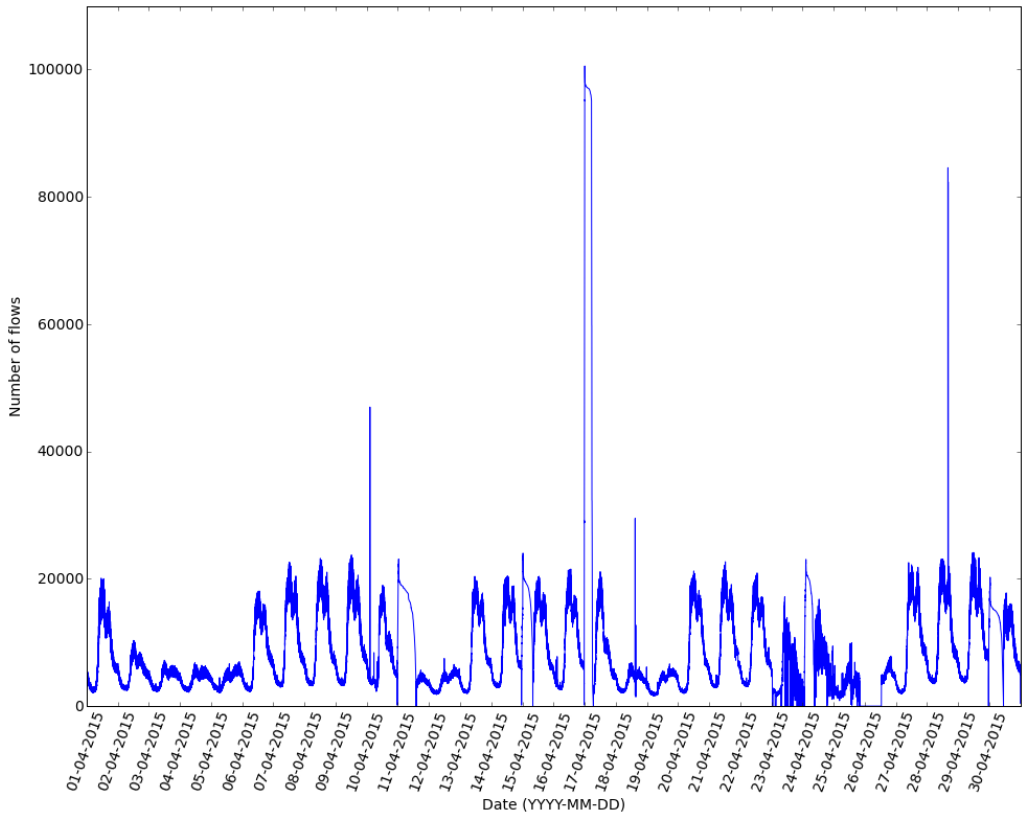

Figure 11: Time series of the processed data for the month of April

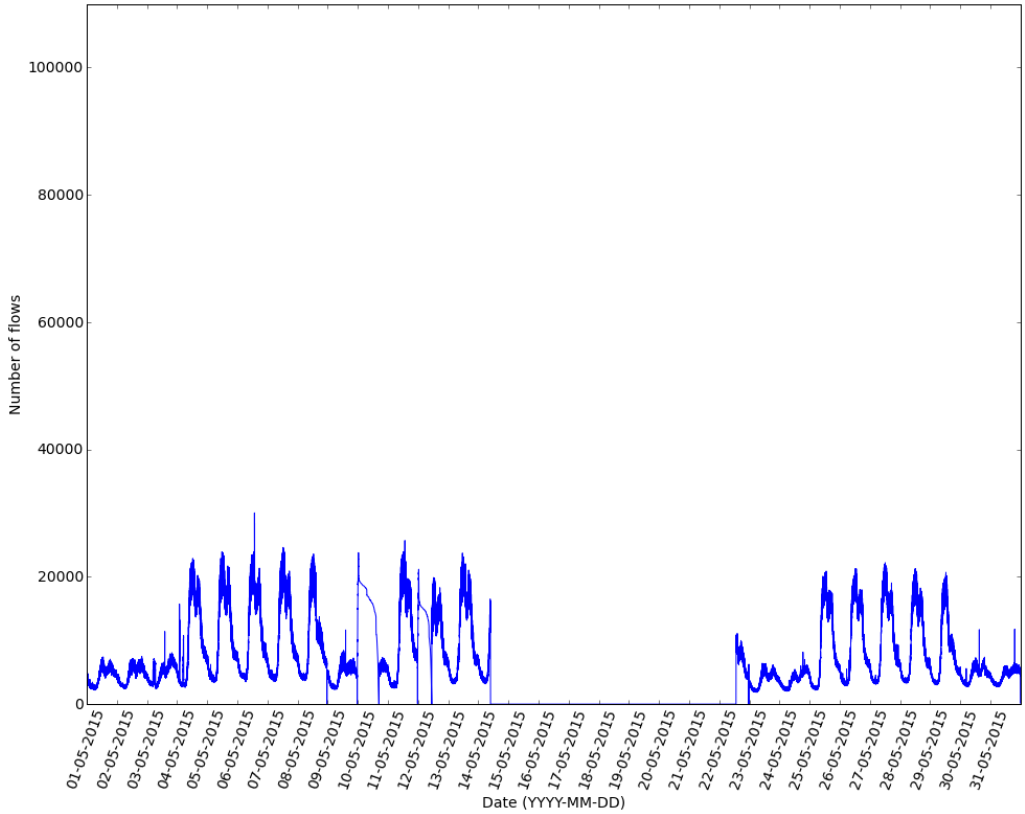

Figure 12: Time series of the processed data for the month of May

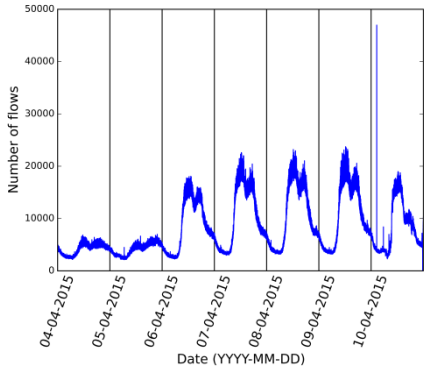

Figure 13 : Time series of the processed data from 4 to 11 April

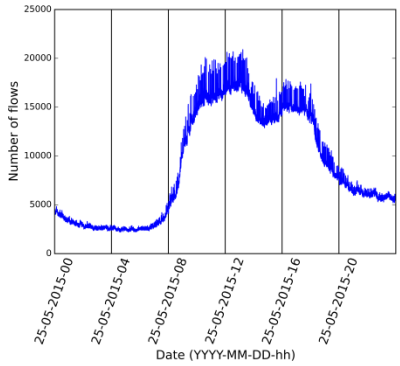

Figure 14: Time series of the processed data for 25 May

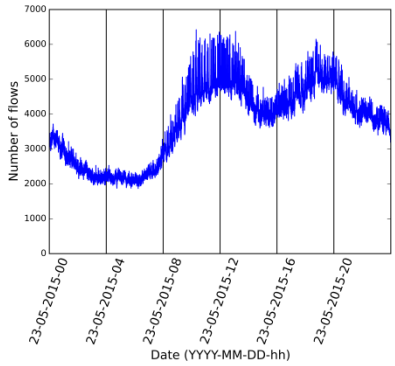

Figure 15: Time series of the processed data for 23 May

As previously stated, the data show clear long-term regularities. The challenges posed by WP5 UC #2, however, require models for short-term traffic behavior. In order to analyze regularities at smaller scales, we carried out a detrending process and then drew key statistics. Interestingly, the time-series becomes almost mean-stationary after first-differencing. Figure 16 shows the mean of the April data after first-differencing, for three different periods: morning (12pm-8am), day (8am-4pm) and evening (4pm-12pm). Its absolute value remains always below zero, while the first-differenced data normally varies in the order of several hundred. Figure 17 shows the standard deviation for the same data. This plot exhibits clear regularities as well. The first few days correspond to the holiday period mentioned above. The peaks in standard deviation correspond to the anomalous activity peaks shown above.

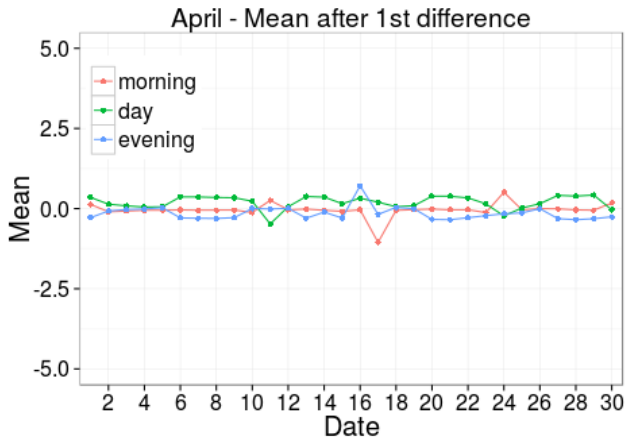

Figure 16 : Mean of the number of flows after computing the first difference

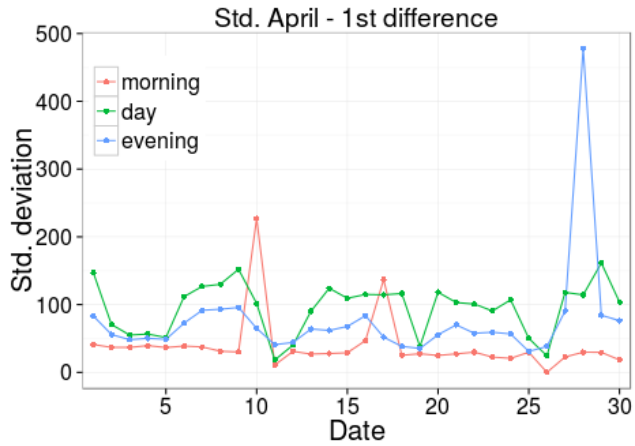

Figure 17 : Standard deviation of the number of flows after computing the first difference

This analysis process allows us to draw some key conclusions:

- The traffic presents clear regularities in the large scale, suggesting that models for long-term forecasting can be long-lasting.
- The data after first-differencing shows regular behavior for at least a one-month-long period, suggesting that models trained for short-term forecasts can be reliably employed across different weeks.

4.1.5.2 Data preprocessing

In order to convert the employed samples of the ONTS dataset into an adequate time-series format, we used the Data Preprocessing module of the NTFF. The flows were thus aggregated into 1-second slots, resulting in a time-series representation composed by 86400 data points per day.

As stated above, the procedure takes about three days in our available hardware for one month of traffic, which is why we could only afford to process a two-month period (i.e. 3TB of data) at first.

As stated above, the resulting time series consisted of the number of open TCP sessions per second, which will be a useful metric for the WP5 congestion control use case. The number of flows crossing a network link is a key variable to correctly allocate available resources (e.g. per-session bandwidth in a congestion control protocol). We chose one second intervals to

keep day-long datasets manageable in a first phase. However, we plan to increase the time granularity in order to stay consistent with the expected timescale for protocol packet RTT (dozens of milliseconds). We have computed forecasts 1, 2, 4 and 60 steps ahead in order to evaluate the performance of the models in both the short and the medium term.

4.1.5.3 Training datasets

We selected two days of April to act as training sets. Since the traffic patterns are very different during week days and weekends (see figure 5), we trained two different models on a Saturday (4th of April) and a Wednesday (8th of April) respectively. Each day consists exactly of 86400 data points (1-second slots).

4.1.5.4 Test datasets

We performed forecasts on the 5th and 7th of April (Sunday and Tuesday respectively) and the 2nd, 5th, 23rd and 24th of May (Saturday, Tuesday, Saturday and Sunday respectively). In order to test the generalization abilities of our models, we used a heterogeneous set of days, comprised of both weekdays and weekend days.
The behavior observed in the ONTS dataset shows a clear periodicity (high load during office hours, low during the night). It is reasonable to assume that similar regularities are found in different data centers, which could leverage this fact to improve their performance and energy savings. For CSP's whose infrastructure spans various time zones, for instance, this could be particularly beneficial. Many applications could benefit from being co-located with their client base, thus reducing consumption of network resources and improving response time. This can be made possible in a cost-effective manner via resource virtualization combined with reliable short-term forecasts to dynamically adapt the available resources at a specific location to the expected demand.

4.1.5.5 Model Training

Ridge Regression: To train this model, we used the Levenberg-Marquardt method previously described, as indicated in section 1.4 of [8]. The input for the model was a (N-50)x50 matrix (where N is the number of samples in the input data set). In this matrix, the first row represents a 50-second-long window from time t to time t-49. The rest of the rows are similar windows, each one corresponding to a period one second before the previous one. We set the initial value of lambda to 1.

ARIMA: We employed the implementation that is available as part of the "forecast" R package. We chose the values of p, q and d using the Hyndmanan-Khandakar algorithm [6]. The value of p was limited to a maximum of 10 and q to 5, because of the exponential RAM and CPU usage that this algorithm incurs. Finally, we estimated the weights (or coefficients) with the AICc criterion (H. Akaike, 1998).

Neural Networks: In order to train the neural network models, we used the SBLLM method [7]. The network architecture consisted of just one hidden layer, with the same number of neurons as the input layer. This layout yielded a good tradeoff between performance and training times. The input matrix was the same as the one used for Ridge regression, but in this case we applied a standardization procedure to the time series so as to have zero mean and unit variance. The training algorithm might fail to converge otherwise. With regard to the parameters, we set the initial step size to 1, the threshold error to 1e-5, previous Q and previous MSE to 1e6. The weights and the initial error were initialized to random values following a uniform distribution in the interval [-2/(100 * N) y +2/(100* N)]. These parameters were chosen after carrying out several experiments. Finally, for performing

predictions the logit function (the inverse of the sigmoid) is applied to the output, which is then multiplied by the variance and incremented by the mean of the data.

All of these training procedures are integrated in the NTFF.

4.1.5.6 Model Evaluation

In this section we show experimental results of the application of the NTFF to the prediction of the number of TCP flows per second –up to 60 seconds ahead- on the ONTS data set. Our results suggest that some of these methods are suitable for the goals of WP5 Proactive Congestion Control use case. Specifically, they show promising results for forecasting the number of sessions crossing a network link, which is key for computing an accurate approximation of the max-min fair bandwidth assignments to sessions in advance.

The models have been evaluated using a modified version of a widely used metric in the field of time-series forecasting, namely the Mean Absolute Percentage Error (MAPE). We computed this metric dividing the Mean Absolute Error by the absolute average of the data values and multiplying it by 100 in order to avoid divisions by zero. This metric is a modified version of the weekly absolute error, MAPEweek [9] [10], for all points of the dataset. From here on we will refer to the previously presented metric as MAPE.

We performed experiments on the previously described samples of the ONTS dataset.

Comparison of the different techniques

We compare the different three techniques described above, both with and without wavelet transforms, by computing the MAPE for the two trained models (Wednesday and Saturday), for 4 and 60 step forecasts. The models were tested on all the six days of the test set.

Figures 18 and 19 show the MAPE for 4-step forecasts, suggesting that neural networks and ARIMA are the best-performing methods for short-term forecasting, the former performing slightly better. It is also apparent that the use of the SWT yields a slightly higher error. This is a consequence of the error in high frequencies being significant, which is carried over to the final result in the summation process, even though the error in low frequencies might have been lower. In general, models trained on Saturday the 4th of April provide better results, which is likely due to the smaller variations that occur on weekends (see Figure 17).

Figures 21 and 22 show the MAPE for 60-step forecasts. The superiority of the models trained on a Saturday over those trained on a Wednesday is again visible. ARIMA yields better long-term results when used in combination with the SWT in the case of the Saturday model, but worse for the Wednesday model. This is consistent with the fact that modeling high frequencies is harder.

We conclude that neural networks provide the best results in general (only in the model trained on Saturday the 4th of April does ARIMA+SWT perform better). The use of the SWT does not show significant improvement, which suggests that they should not be employed in contexts where computation efficiency is key, as in some stream processing scenarios. Despite performing slightly worse than neural networks, ARIMA provides good results. Given that it is much more easily trained than the latter, the use of this model should be considered for practical scenarios. Since the results provided by the three techniques in the short term are fairly similar, we will consider all of them for their application in WP5 UC#2 (Proactive Congestion Control).

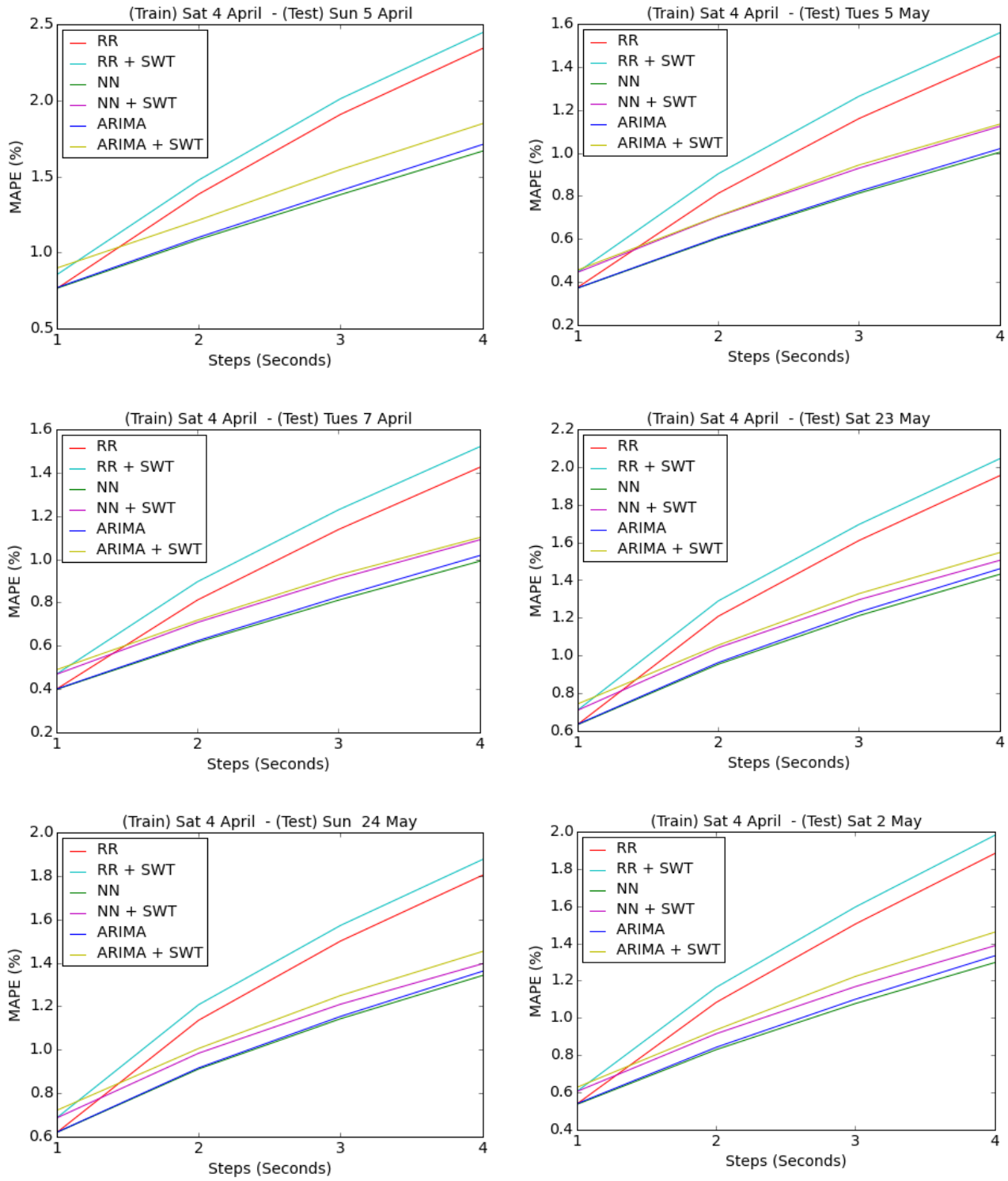

Figure 18: MAPE of 4 step forecasts for 6 techniques, 2 training datasets and 6 test datasets.

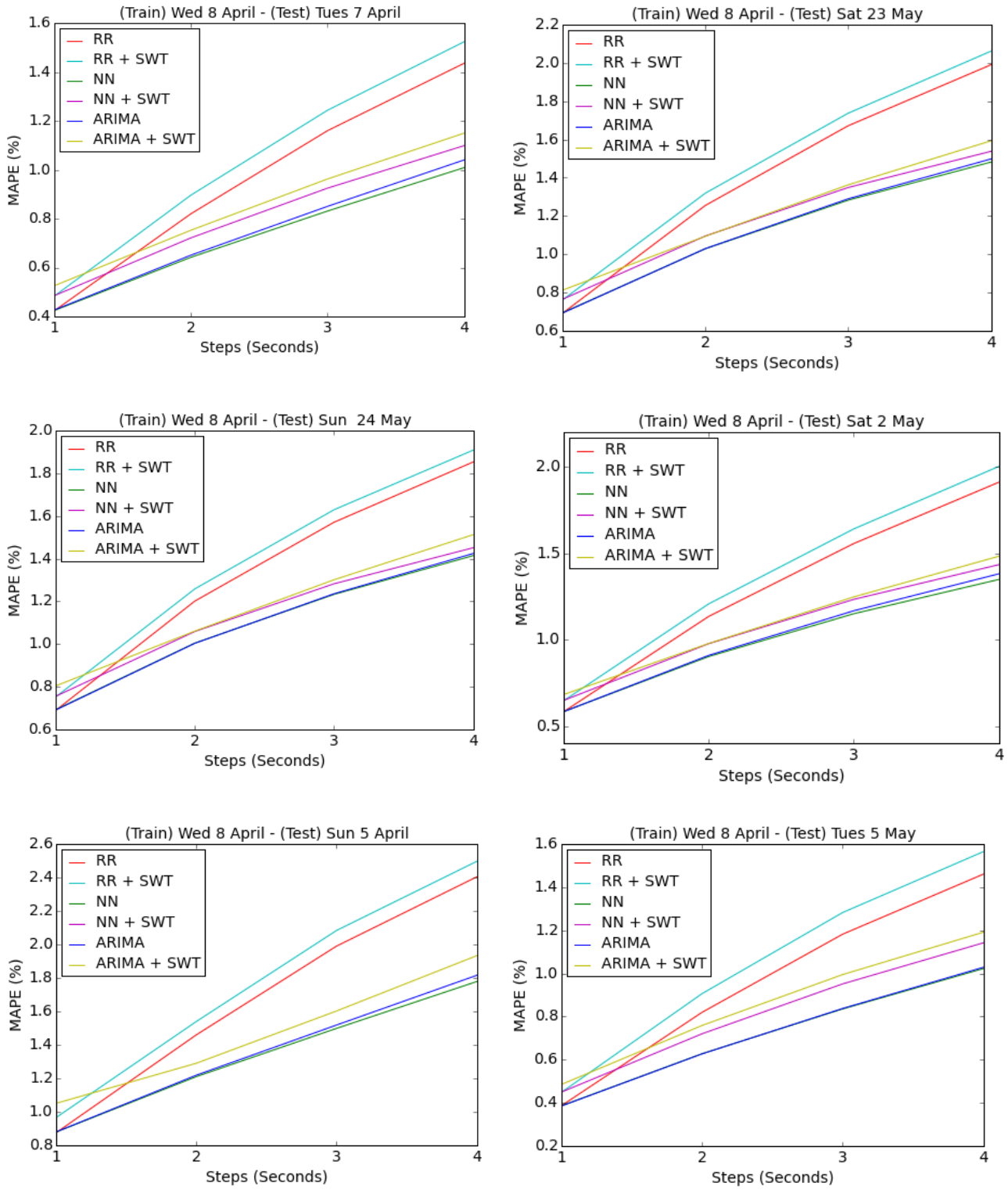

Figure 19: MAPE of 4 step forecasts for 6 techniques, 2 training datasets and 6 test datasets.

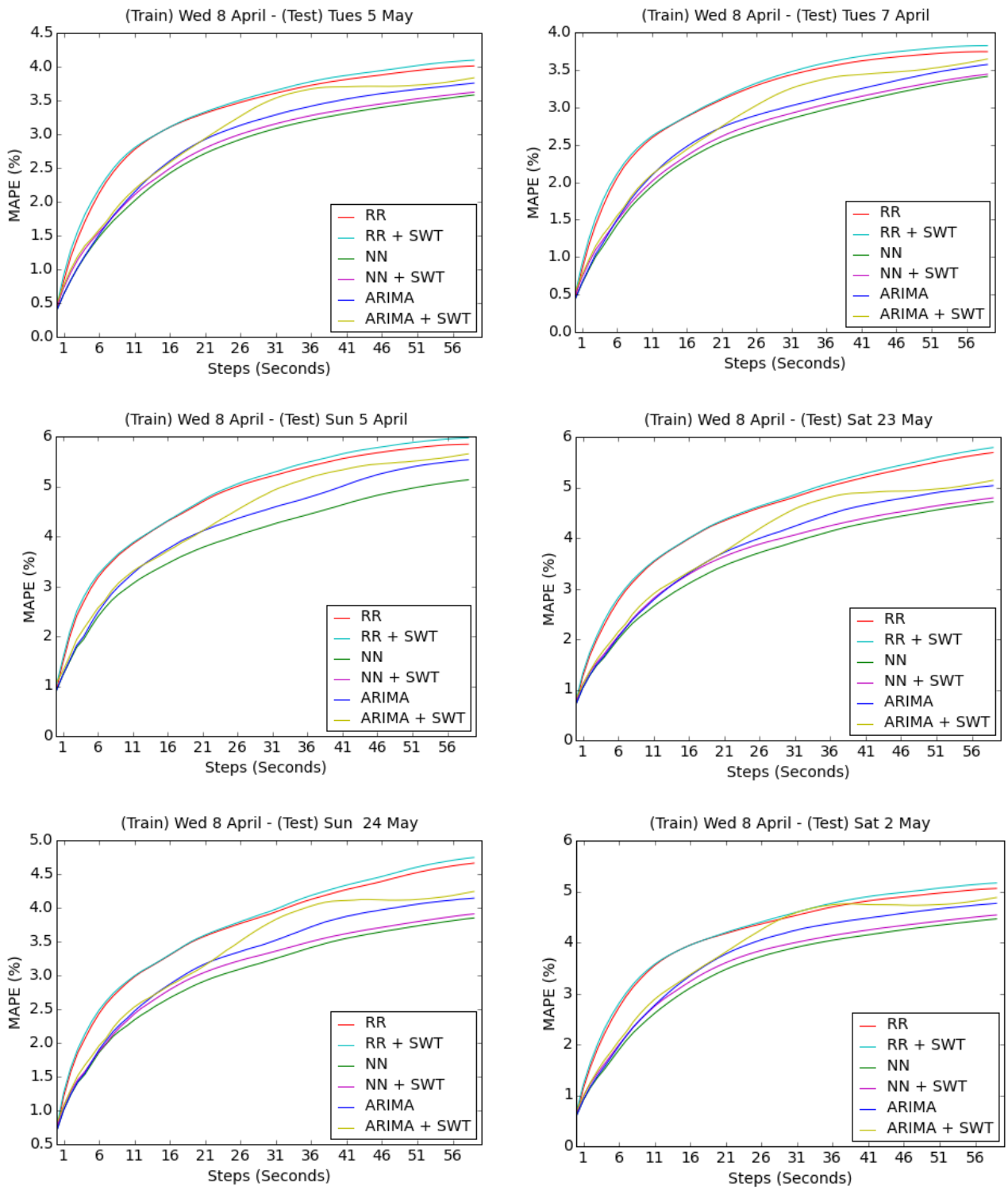

Figure 21: MAPE of 60 step forecasts for 6 techniques, 2 training datasets and 6 test datasets.

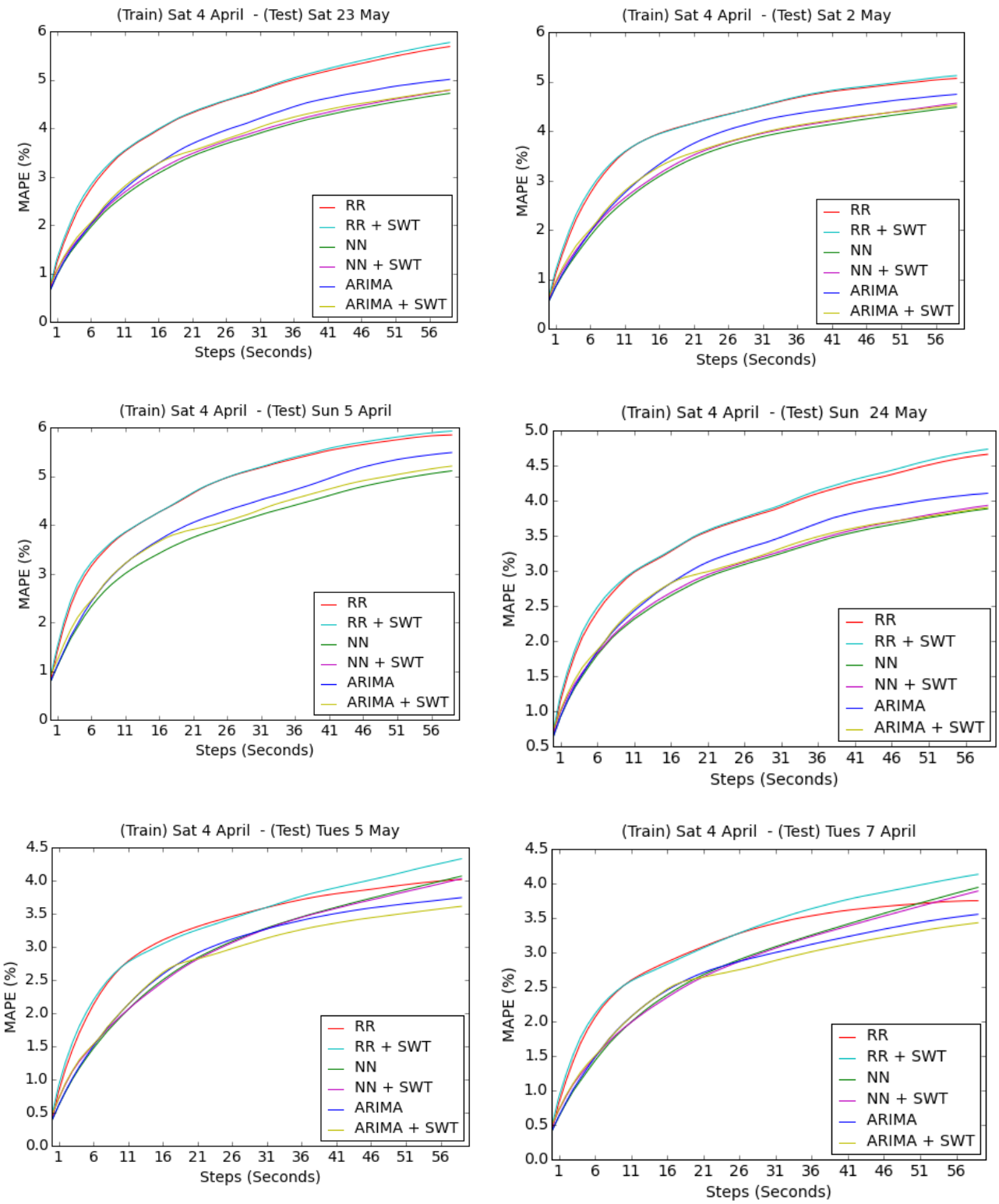

Figure 22: MAPE of 60 step forecasts for 6 techniques, 2 training datasets and 6 test datasets.

Short-term forecasts

As previously stated, our ultimate goal is to integrate the forecasting procedure into the congestion control system being developed for WP5 UC#2. This system works by having

router links notify sources of their max-min fair bandwidth assignment using probe packets, which traverse the network from source to destination and back. By taking into account the expected number of sessions crossing the link a few milliseconds in advance, a bandwidth assignment can be computed that remains valid once it reaches the source. Since the average round trip time for probe packets is generally in the order of milliseconds, we validate the ability of the NTFF to perform short-term forecasts.

In this section, we show 1, 2 and 4 step forecasts (figures 8 and 9) using neural networks (which showed the best performance in the previous experiments) on one-minute long periods taken at 7am and 5pm from a Saturday and a Tuesday (both from the test set).

Figures Figure 24 and Figure 25 show neural network forecasts of up to 4 steps along a time window, superimposed on the original data. It is apparent that the results are of an acceptable accuracy for the intended use case. The results from Figure 24 are consistent with the previous conclusions, that is, that models trained on a Saturday are generally more robust that those trained on a Wednesday.

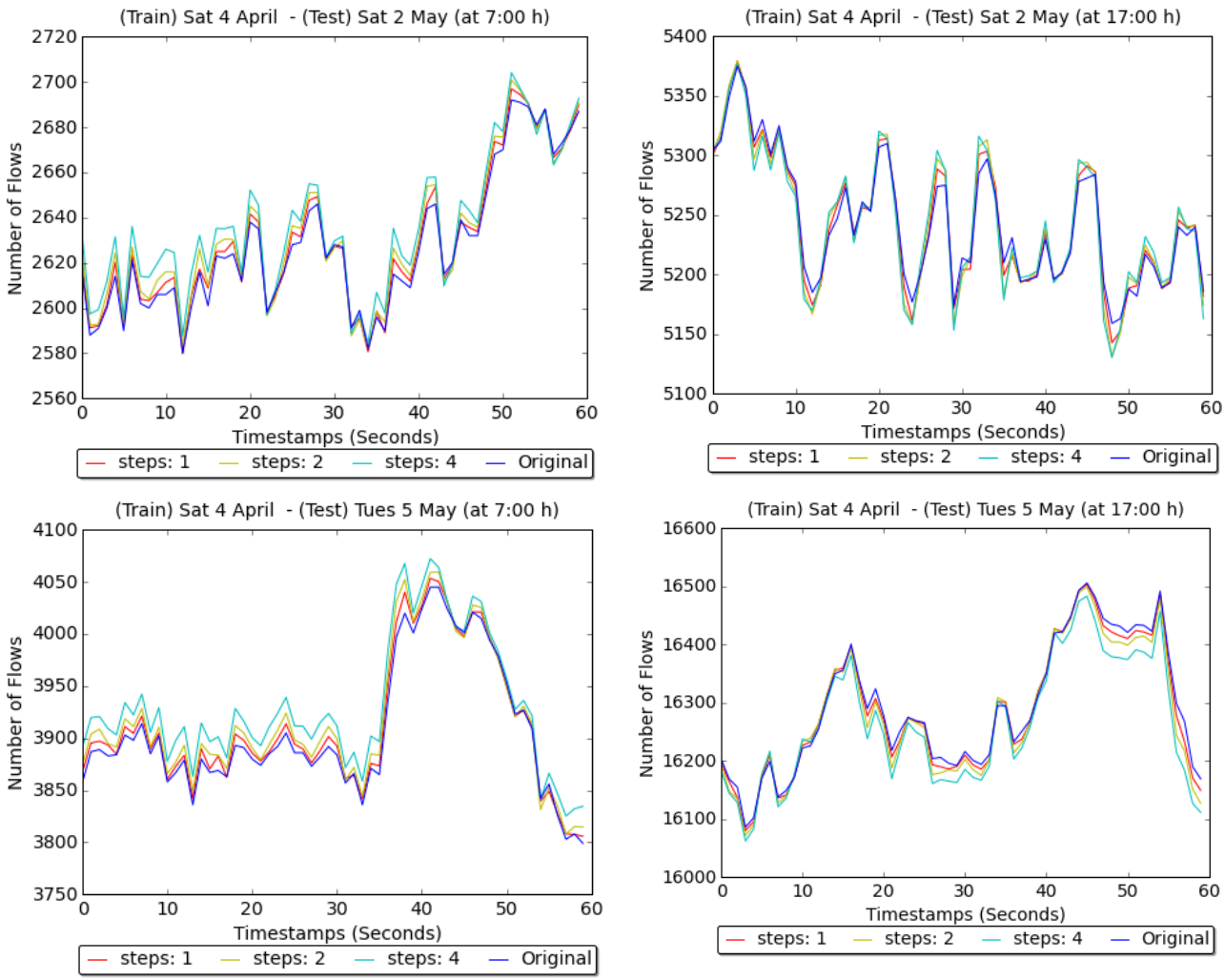

Figure 24: Forecasts on test data using neural networks, 1, 2 and 4 steps ahead, in one-minute periods at two different times per day (7am and 5pm), from a weekday (Tuesday) and a weekend day (Saturday), with the model trained on Saturday the 4th of April

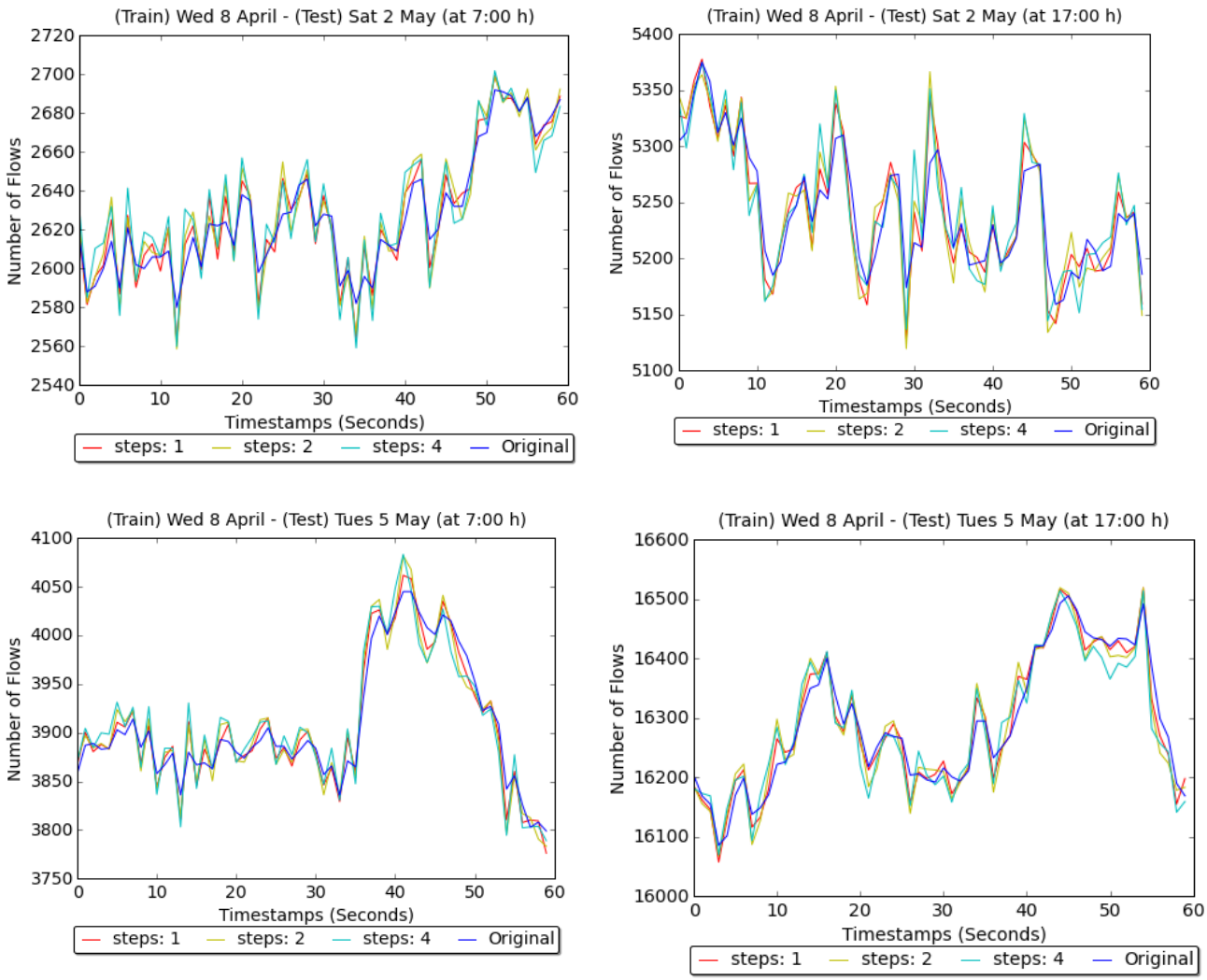

Figure 25: Forecasts on test data using neural networks, 1, 2 and 4 steps ahead, in one-minute periods at two different times per day (7am and 5pm), from a weekday (Tuesday) and a weekend day (Saturday), with the model trained on Wednesday the 8th of April.

The results obtained here are of crucial importance, since they help determine whether or not the forecasting system that we are going to integrate into the WP5 congestion control system is able to make reliable short-term forecasts. As shown by the results, the NTFF is capable of making forecasts up to 4 steps ahead within a very acceptable margin. The results are consistent across a variety of test sets, which correspond to different weekdays and weekend days. This suggests that we will be able to train a robust model for the congestion control protocol to be developed in UC#2.

Neural networks on different window sizes.

Since neural networks are the model that shows the best results in our experiments, we have tested the impact of the training window (i.e. the number of input samples used for training and forecasting) on the accuracy of the forecasts. The training time for neural networks is highly dependent on the dimensionality of the input. Therefore, in an online environment it is important to keep the window as small as possible, so that the model can be retrained and updated fast enough.

Figure 27 shows the MAPE for 4 step forecasts using two neural network models (which performed best) trained on the two training days (Wednesday and Saturday). We trained the

model using different window sizes (25, 50, 100, 200 and 400), and tested them on a Tuesday and a Saturday from the test set. These figures show that increasing the window does not yield significant improvements, and a size of 25 is therefore enough for short-term forecasts. However, more experiments should be conducted in order to determine the optimal window size with respect to a tolerated error threshold. This is part of the ongoing WP4 work and will be integrated into the NTFF in the next release.

Even though our main focus is on short term forecasts, which best suit the way in which bandwidth assignments will be predicted by the congestion control system being developed in UC#2, we have also tested the impact of window sizes on medium term forecasts. These can be useful in various network management and engineering scenarios, as shown for instance in [5]. Figure 28 shows the results of a set of experiments equivalent to those described above, but performing 60-step forecasts instead. Here we can see a much more significant impact of the window size. Among the tested settings, the best value (i.e. the point of diminishing returns) is clearly attained at 200. As expected, this result shows that increasing the number of steps requires a much larger window in order to keep the error from growing rapidly. However, the lower bound for the attainable error is significantly above the one observed for 4-step forecasts, regardless of the window size. This motivates new tests using different models –other than the ones tested here- for medium-term forecasts in order to decrease this lower bound. An additional result shown by these figures is that a window of size 25 remains sufficient up to about 10-step forecasts, with increases in window size showing little improvement.

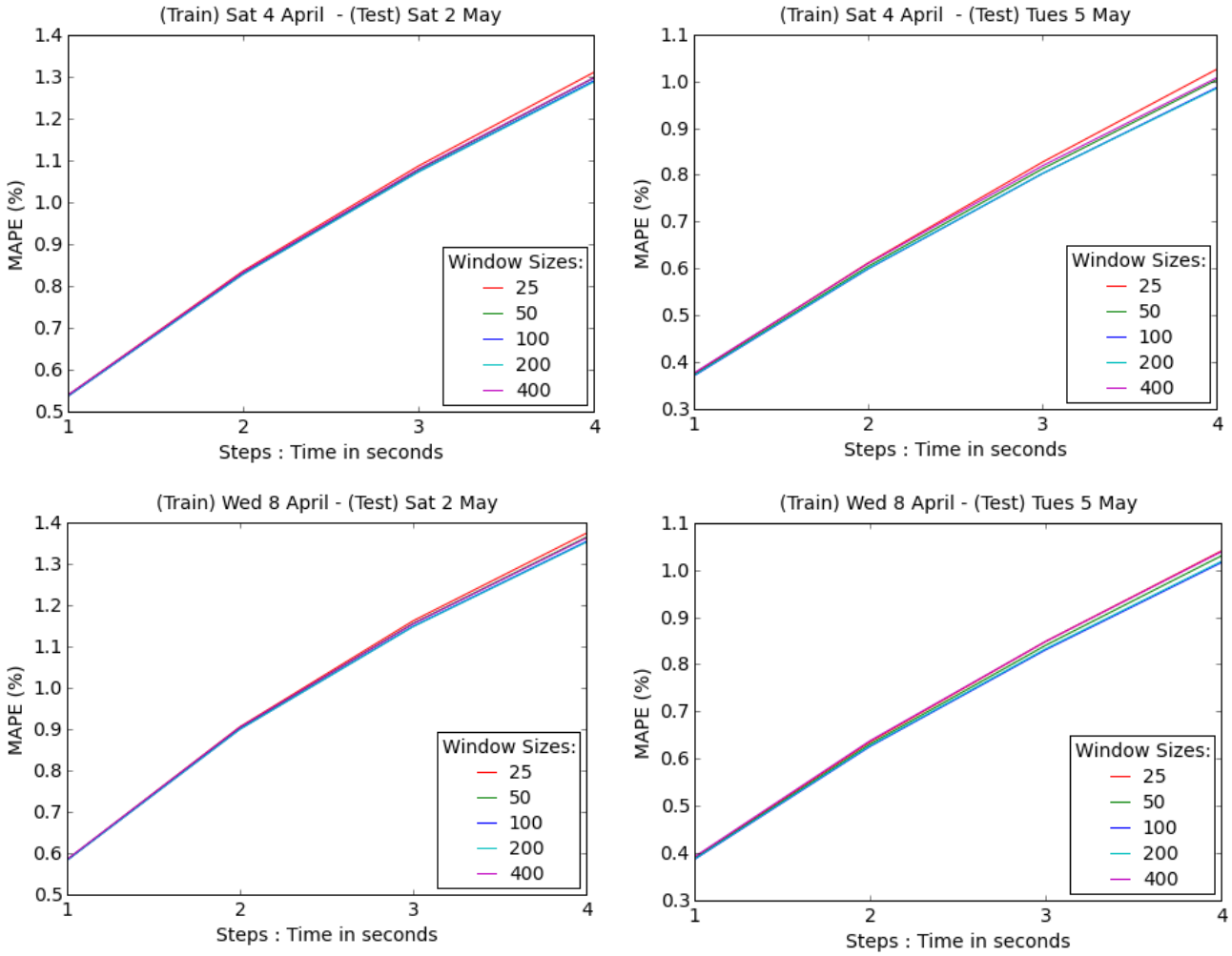

Figure 27: Forecasts using neural networks with different window sizes (25, 50, 100, 200 and 400), on a Tuesday and on a Saturday from the test set.

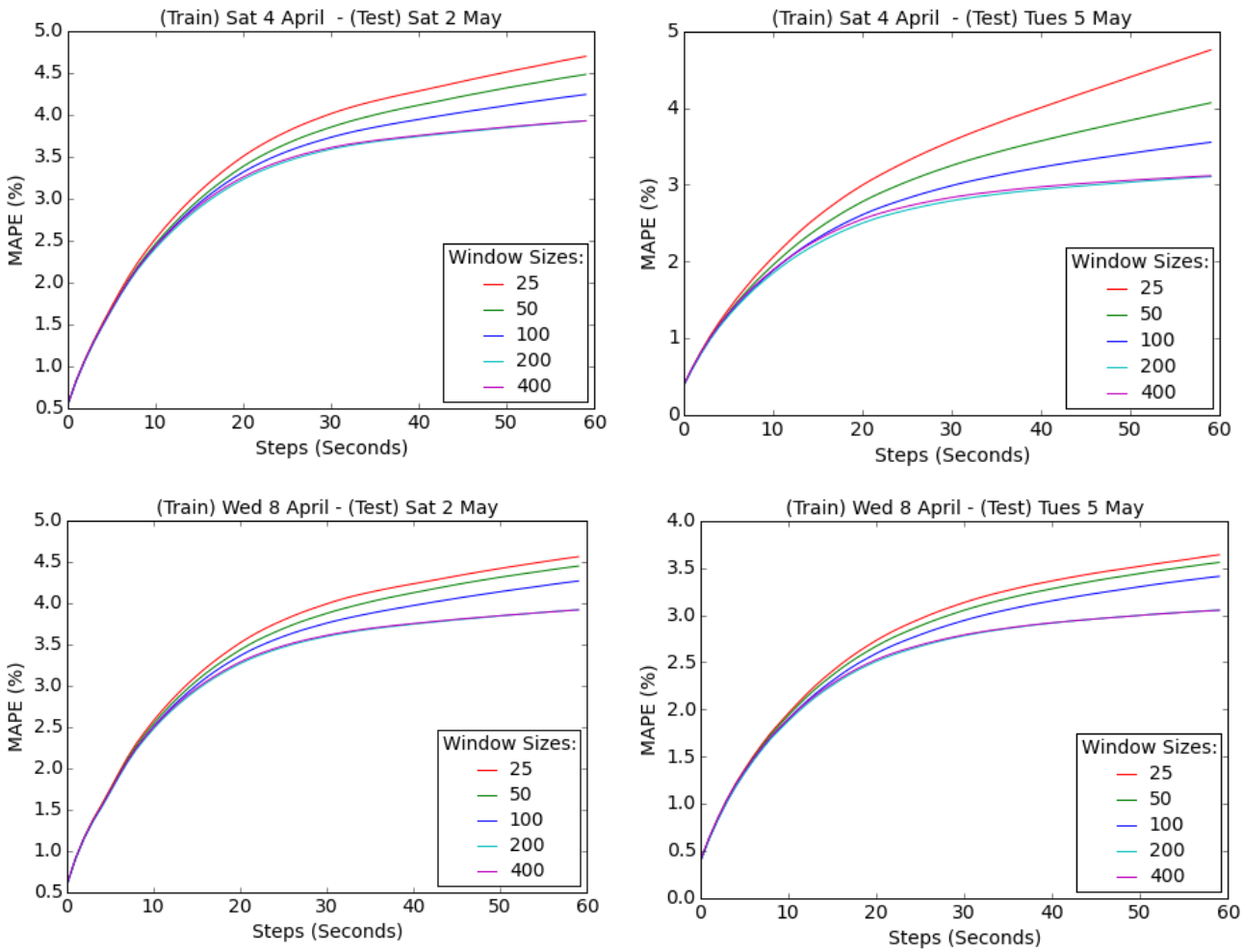

Figure 28: Forecasts using neural networks with different window sizes (25, 50, 100, 200 and 400), on a Tuesday and on a Saturday from the test set.

4.2 Conclusions and work in progress

We have addressed the problem of building robust forecasting models for network traffic analysis. We have shown that the patterns underlying the analyzed network traffic data present regularities that can be exploited for producing reliable, short-term forecasts.

We have proposed and developed a forecasting procedure specifically designed to predict short and medium term traffic behavior. This procedure has been integrated into a Network Traffic Forecasting Framework (NTFF), comprised of three modules: Data Preprocessing, Wavelet Decomposition and Forecasting Model Design. Up to our knowledge, there exists no open-source solution integrating these functionalities. The first released version of NTFF has been initially designed with the goal of forecasting the number of open TCP sessions crossing an ISP's core network at a given time period, which can be particularly useful in the congestion control protocols being developed in WP5 UC#2. The NTFF, however, can be used to forecast many different variables and could therefore be useful to tackle a variety of problems that arise in network management and engineering. The NTFF is available online at a git repository.

We have conducted a set of experiments using a sizeable sample of the ONTS dataset to validate the NTFF and test the considered models. Interestingly, models trained on a weekday or a weekend day perform equally well on weekdays and weekend days. This is consistent with the intuition that despite the observable changes in trend between weekdays and weekend days, small-scale dynamics remain similar. Regarding short-term forecasts (up to 4 steps ahead), neural networks are the model that performs best, although all of the analyzed models can make forecasts of an acceptable reliability (2% of mean error). Therefore, the simplest model can be used in practice if computational resources are limited. All models will therefore be taken into account for WP5 UC#2, where network nodes are expected to be running the forecasting algorithms and computational efficiency is an important requirement.

Even though our main focus is on short term forecasts, medium term forecasts can also be useful in various network management and engineering scenarios. In this regard, we have tested the ability of our procedure to make forecasts of up to 60 steps ahead, with neural networks and ARIMA performing best. Ridge regression, however, does not provide good results. The use of wavelet decompositions, forecasting each component separately, is not beneficial in general. We have only observed significant improvements when using wavelet transforms in the case of 60-step forecasts using ARIMA.

The literature is scarce with publications showing effective forecasting techniques on datasets of the nature and scale of ONTS. This, along with the effectiveness of our forecasting procedure, makes for a compelling publication. Therefore, our results will be compiled and submitted to a journal in the field of network management and engineering. In addition, recent publications show an interest of applying these forecasting methods to specific network-related problems [5]. Since our experiments are promising, the results from applying our forecasting procedure to novel congestion control protocols as part of UC#2 are also likely to be part of a future publication in this area.

The research activities that have been conducted reveal certain key challenges to be addressed in the field of time series forecasting, and specifically for network management and engineering. In the future, we plan to continue our research in order to improve our results and find solutions to these problems. One of these challenges is the choice of the window size to use for training the models and making forecasts when using neural networks. Up to our knowledge, this has not been sufficiently addressed in the forecasting literature, but it constitutes a key aspect of online model design. Another open issue in the application of neural networks is the design of network architecture, i.e. the number and layout of hidden neurons. The recent years have seen significant advances in this regard, but they have not yet been sufficiently explored in the field of forecasting. We plan to combine our forecasting procedure with these proposals in order to improve our results. We also plan to perform additional experiments using more sophisticated models, especially different neural networks that have proved effective in other domains, such as recurrent and convolutional neural networks.

Another key issue is the use of wavelet transforms in combination with our forecasting models. Contrary to our expectations and the results reported in other domains, our experiments do not show clear benefits to be gained from their application. We have observed specific issues when using both the Haar (poor frequency bandwidth precision) and the Daubechies (artifacts at the edges of individual components) mother wavelets. We plan to study the extent to which the Daubechies artifacts can be mitigated using transforms of a

different order. We also plan to employ different mother wavelets to evaluate their applicability and usefulness in this domain.

Finally, we plan to perform much more exhaustive experiments in order to answer certain key questions, such as: what is the expected lifetime of a forecasting model before retraining is necessary? Are these models sufficiently robust in the face of anomalous behavior? Is online training feasible and/or effective in practical scenarios? The availability of an enterprise-grade cluster will make it possible to perform the necessary experiments to answer those questions, as it will allow us to process all of the ONTS traces that are available at the moment. A large scale time series analysis will also be carried out in order to reveal long-term trends and dynamics in network traffic.

5 Network Anomaly and Intrusion Detection Algorithms

With the booming in the number of network attacks, the problem of network anomaly detection has received increasing attention over the last decades. However, current network anomaly detectors are still unable to deal with zero days attack or new network behaviors and consequently to protect efficiently a network. Indeed, existing solutions are mainly knowledge-based and this knowledge must be continuously updated to protect the network. However building signatures or new normal profiles to feed these detectors take time and money, As a result, current detectors often leave the network badly protected.

To overcome these issues, a new generation of detectors has emerged which takes benefit of intelligent techniques which automatically learns from data and allows bypassing the strenuous human input: unsupervised network anomaly detectors. These detectors aim at detecting network anomalies in an unsupervised way, i.e. without any previous knowledge on the anomalies. They mainly rely on one main assumption [11] [12]:

> *"Intrusive activities represent a minority of the whole traffic and possess different patterns from the majority of the network activities."*

Thus, unsupervised network anomaly detectors exploit data mining algorithms to identify flows which have rare patterns and are thus anomalous. A state of the art on network anomaly detection has already been presented in section 6.1 of the deliverable 4.1.

In this section, we start by describing two new algorithms which improves UNADA, an unsupervised network anomaly detector presented in the deliverable 4.1, in terms of scalability and execution time. The first one, PUNADA, is a parallel version of UNADA which allows distributing UNADA's computation over a large cluster of servers and takes benefit of new solutions from the Big Data world to speed up its execution. The second algorithm, ORUNADA, is an Online and near Real-time Unsupervised Network Anomaly Detection Algorithm, which relies on an incremental grid clustering algorithm and a time sliding window.

Finally this section provides comparative evaluation of existing unsupervised network anomaly detectors and underlines new challenges in the field.

The PUNADA is open source and is available online at https://gitlab.com/ontic-wp4/PUNADA. Details on how to use it are provided in Annex A.

5.1 PUNADA: A Parallel Unsupervised Network Anomaly Detection Algorithm

To uncover anomalies, unsupervised network anomaly detectors need to dive deeply into the network traffic to identify flows' patterns. They are often time-consuming and unable to meet real-time requirements. To solve this issue, existing detectors may process only sampled data which implies that harmful traffic may not be processed and so not detected [13].

To overcome this limitation, we propose PUNADA, a parallelizable version of UNADA which distributes the computing of UNADA over a cluster of servers. For sake of completeness, we

start by describing UNADA. Then, PUNADA is exposed and finally the obtained results are presented and analyzed.

5.1.1 UNADA: An Unsupervised Network Anomaly Detection Algorithm

Flows in network traffic are usually represented by a large set of dimensions or features which represents statistics on the flows. However, in high dimensions, the curse of dimensionality phenomena occurs: distance becomes meaningless and every point tends to become an outlier. Due to this curse, unsupervised network anomaly detectors tend, in high dimensions, to detect every flow as an outlier, i.e. as an anomaly. UNADA is a robust and efficient detector which addresses this issue by applying subspace clustering and evidence accumulation techniques. UNADA divides the whole space in subspaces and cluster each subspace independently. It then aggregates the partitions obtained to get a picture of the whole space and identify the anomalies. This algorithm can be divided in three parts: the preprocessing step, the subspace clustering step and finally the evidence accumulation (EA) step.

UNADA works on single-link packet-level traffic captured in consecutive time-slots of fixed length, ΔT. During the preprocessing step, packets are aggregated in flows using an aggregation flow key, which is usually the IP destination or the IP source associated with a mask (/32, /24, /16, /8). Numerous features can be computed over a flow such as: nDsts (# of different IPdst), nSrcs (# of different IPsrc), nPkts (# of pkts), nSYN/nPkts, nICMP/nPkts, etc. Each flow is described by a set of A features in a vector x_f. The set of vectors is denoted by a normalized matrix $X = (x_1, \dots, x_F)$ representing the features space.

In a second step, UNADA divides the feature space X in N subspaces $X_1, X_2, \dots X_N$ of two dimensions. It builds as many subspaces as there is combination of two dimensions, thus $N = m(m-1)/2$ with m the total number of dimensions (features). A clustering algorithm is then applied on each subspace X_i. It outputs a partition of the subspace where similar flows are grouped in clusters. Dissimilar flows are isolated and considered as outliers in the subspace. The dissimilarity between two flows is evaluated with a distance function like, for example, the Euclidian or the Mahalanobis distance function. Two flows which are close according to this distance function are considered as similar otherwise they are considered as dissimilar. UNADA is based on a density-based algorithm DBSCAN as it has the advantage to discover clusters of any shape in noisy data [14].

Finally, to combine the N obtained partitions, UNADA relies on an EA algorithm for Outliers identification (EA4O) which accumulates for each flow the level of abnormality it gets in each subspace. In a subspace, if a flow belongs to a cluster its level of abnormality is set to null, otherwise its level of abnormality is proportional to its distance with the centroid of the biggest cluster. A dissimilarity vector $D = (d_1, .., d_f \dots, d_F)$ is built where each element d_f reflects the accumulated level of abnormality of a flow f. To select the most pertinent anomalies, the dissimilarity vector is sorted and an anomaly detection threshold TH is defined. TH is set at the value for which the slope of the sorted dissimilarity presented a major change. Every flow with a dissimilarity score above this threshold is considered as anomalous.

5.1.2 Evaluation of UNADA

Initial evaluations of the detection performance of UNADA have already been performed by Pedro Casas during its postdoctoral work at LAAS-CNRS under the direction of Philippe Owezarski [15]. The evaluations were performed on a public ground truth: the MAWI dataset. This dataset consists of labeled 15 minutes network traces collected daily from a trans-Pacific link between Japan and the United-States since 2001 until now. The labels have been obtained by combining the results obtained by four different anomaly detectors [16].

Figure 29 depicts the ROC (Receiver Operating Characteristic) curves obtained with the MAWI data set with two different keys for flow aggregation; the IPsrc and the IPdst. The performances are compared with three previously used approaches for unsupervised network anomaly detection: DBSCAN, K-means and PCA. The first two consists in applying either DBSCAN or k-mean to the feature space matrix X to identify the largest cluster C_{max} and compute the Mahalanobis distance of all flows lying outside C_{max} to its centroid. The PCA based approach uses the subspaces method [17] applied to the X matrix. The results show clearly that UNADA outperforms the other methods. This difference can be explained by the fact that UNADA detects anomalies on subspaces while the other detectors process directly on the whole space and thus may suffer from the curse of dimensionality.

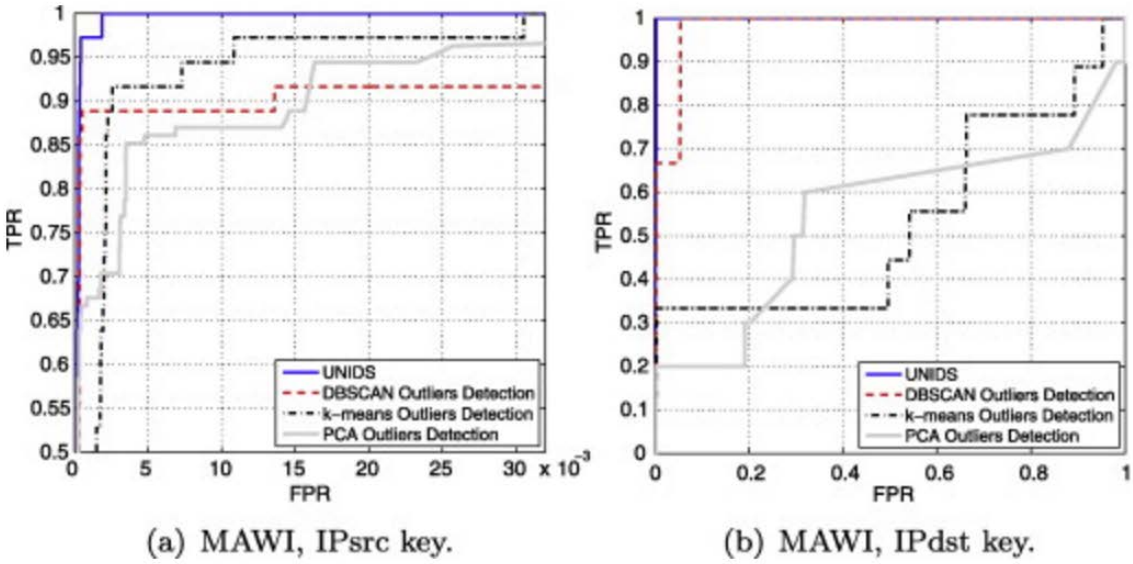

Figure 29 : ROC curve obtained with the MAWI dataset (UNADA is called UNIDs)

To evaluate UNADA in terms of execution time, ONTS traces have been used. These traces contain the 64 bytes of the header of each IP network packet that crosses the link of Interhost's core network, Interhost being a subsidiary of SATEC. More information about this collect can be found in deliverable D2.4. This link is crossed by around 300,000 packets per second and 1.2Gbit/s of data. As only headers are stored, UNADA, to be real time, should then be able to process 19.2 Mbit of data per second and thus 1.6 Terabytes per day.
For UNADA's evaluation, we analyze 60 slots of ONTS traffic, the aggregation key is set to IPsrc/16 and the time slot ΔT to 15 seconds. During a slot, 4,500,000 packets are collected. The evaluation does not consider the flows' features computation time; we assume that a dedicated hardware/process performs this task upstream, as there already exists powerful tool to complete this task like [18]. Figure 30 displays UNADA's mean execution time of one

slot obtained on a single machine with 16 Gbit of RAM and an Intel Core i5-4310U CPU 2.00GHz. The results show that UNADA's execution time is mainly due to the clustering step (DBSCAN) and that it increases nearly exponentially with the number of features. It can be noticed that the other steps' execution time (in black on the histograms) are nearly independent of the number of features. To be an online system, UNADA should process the data faster than they arrive, i.e.; in less than 15 seconds. However, to process 15 seconds of ONTS traces with 16 features per flows, UNADA takes nearly fifty seconds. By accumulating the additional time necessary to complete the detection, the interval of time between the reception of the 100th slot and its processing would be of approximately one hour. Therefore, UNADA is not fit for online detection as it is; the time between an anomaly's occurrence and its detection increases linearly and is in the order of hours after only a few hundreds of detection's cycles.

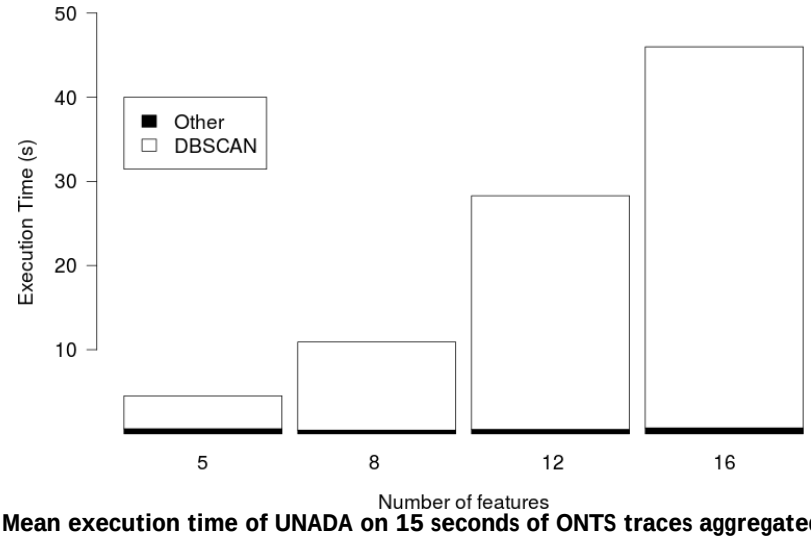

Figure 30 : Mean execution time of UNADA on 15 seconds of ONTS traces aggregated at the IPsrc

5.1.3 PUNADA's description

The idea of PUNADA is to process UNADA's subspaces in parallel; the subspaces are distributed over a cluster of servers which independently perform DBSCAN and the EA40 algorithm.

PUNADA is implemented using Spark 1.2.0. [19] to distribute its computations. Spark is an open source cluster computing framework developed by the Apache Software Foundation. Spark technology has been selected for realizing the analytics engine of the ONTIC Big Data architecture (see deliverable D2.3 for more details on the architecture and the Spark technology). PUNADA's implementation relies on two main Spark's operations (see Figure 31):

- A map operation which sends across the cores of the cluster the processing of the N subspaces. The clustering and the EA of each subspace are thus parallelized. The map function returns a dissimilarity vector for each subspace.
- A reduce operator which aggregates the dissimilarity vectors obtained in each subspace. It simply sums the dissimilarity vector of each subspace to obtain the global dissimilarity vector.

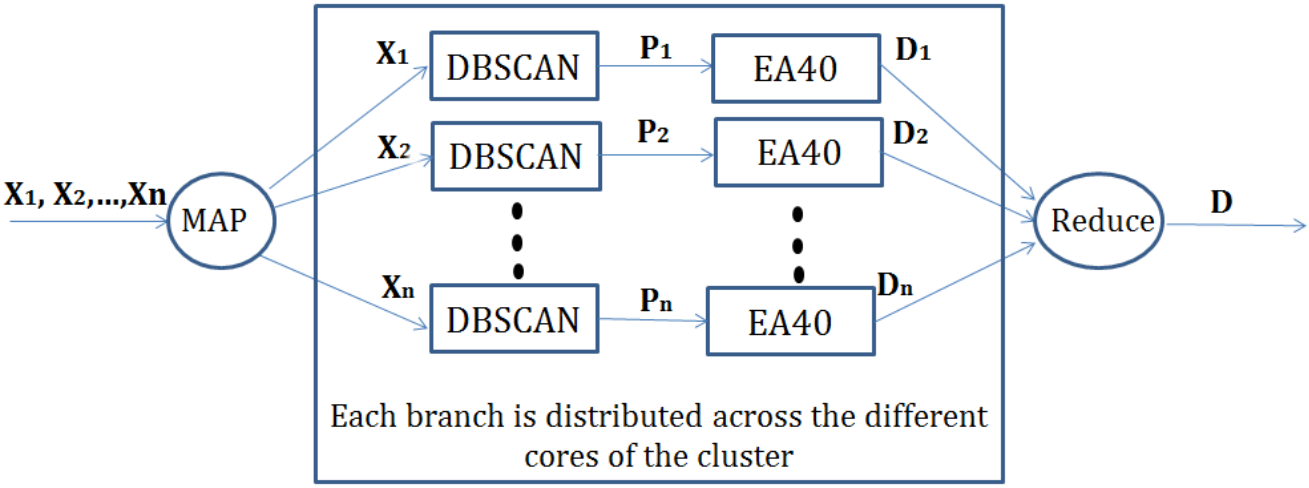

Figure 31 : UNADA's deployment over a cluster of servers with Spark

5.1.4 Evaluation of PUNADA

The validation has been performed on the Grid5000 platform [20], a large-scale and versatile testbed which provides access to a large amount of resources: 1000 nodes, 8000 cores, grouped in homogeneous clusters. We have used nodes with 8 GB of RAM, two CPUs at a frequency of 2.26GHz, each with 4 cores. PUNADA has been validated in terms of scalability and execution time.

Figure 32 displays PUNADA's execution time as a function of the number of features and cores considered. As it can be seen, PUNADA's execution time for each set of features decreases as the number of cores increases until reaching a threshold. Note that, this threshold is inferior to the number of subspaces generated for each set of features. Furthermore, the difference in execution time of PUNADA with different number of features tends to decrease while adding new cores. Indeed, with 40 cores, we can observe that the execution time of PUNADA is nearly the same, around 4ms, no matter you use 8, 12 or 16 features. It implies that a high number of features does not prevent any longer from using the detector.

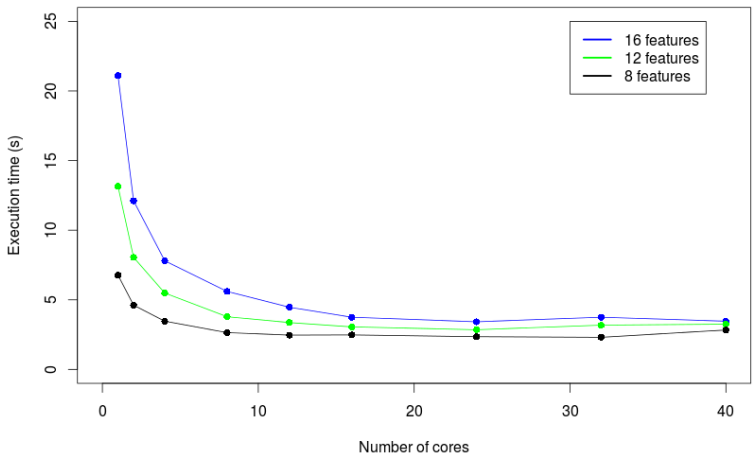

Figure 32 : Execution of PUNADA according to the number of cores and number of features

Figure 14 depicts the gain in execution time of PUNADA as a function of the number of cores and features. This gain increases with the number of cores till reaching a threshold. Furthermore, it can be noticed that the gain in execution time, with 8 and 12 features, from a

certain number of cores, decreases. These phenomena can be attributed to an over-dimensioned cluster for this amount of features. As a result, PUNADA's execution time is affected by the communication overhead of the distributed computing environment.

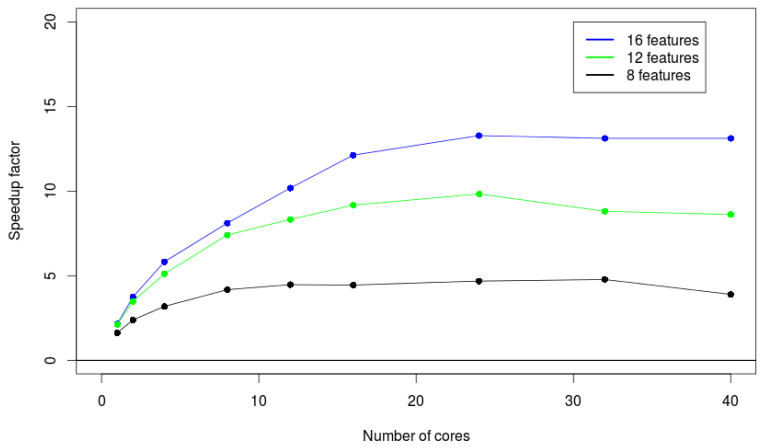

Figure 33 : Gain in time of PUNADA according to the number of cores and features

In both the figures above, a limit in the number of cores beyond which the execution time of PUNADA does not improve can be observed. One would expect this limit to be equal to the number of subspaces (which is in the order of the square of the number of features), however this limit is largely inferior. After an inspection of the Spark log, this limit can be attributed to the serializing time which becomes larger than the processing time.

5.1.5 Conclusion and Future Works

PUNADA is a parallel version of UNADA which distributes the subspaces to process to a cluster of servers. It benefits of the performance of an emerging and powerful tool from the big data world: Spark. Evaluation results show that the computation distribution (1) improves significantly processing time till reaching a limit and (2) allows execution time to be nearly independent of the number of features. Thus, the number of features can be largely increased, thereby improving the quality of the detection. PUNADA is a step forward to real-time detection. However, the evaluation also underlines a limit in the number of cores that can be added to enhance PUNADA execution time. Thus, to enhance the detection time more in depth changes of UNADA are required.

5.2 ORUNADA: An Online and Real-time Unsupervised Network Anomaly Detection Algorithm

As already stated, unsupervised network anomaly detectors have emerged to overcome the limitation of knowledge-based anomaly detection systems. These systems can detect anomalies in network traces without relying on signatures, training or labelled traffic of any kind. These new systems have to deal with two main issues for which few solutions have been proposed and which prevent them from being real-time and online:

- A high complexity. Indeed, an online detector must be able to process the network traffic as soon as it arrives which is rarely the case with existing unsupervised network anomaly detectors as they are often very slow due to their high complexity.
- The use of consecutive large time-bins of network traces. Indeed, to identify intrusions, detectors must be applied on large network traffic traces which must last dozens of seconds. Therefore, a long time may elapse between the occurrence and the detection of an anomaly, during which the network may be under attack.

Among existing detectors, UNADA, a system developed in the LAAS-CNRS's laboratory has shown to achieve good performance. However, UNADA suffers from high complexity and the use of consecutive large time-bins which prevent it from being an online and real-time detector. In the previous section, we have proposed PUNADA a parallel version of UNADA which allows distributing UNADA computation over a cluster of servers. It decreases UNADA's execution time by a factor of 10. However, above a certain number of cores the gain in time tends to stabilize and even decrease.

In this section, we propose ORUNADA, an Online and near Real-time version of UNADA which can detect in continuous network anomalies. This is made possible thanks to a sliding window and an incremental grid clustering algorithm. While usual clustering algorithms re-compute the whole space when few data change, an incremental clustering only updates the old partition. Thus, it can re-compute rapidly and efficiently the clusters when few data change. The time-sliding window of ORUNADA enables to update the feature space more frequently and thus handle, in a continuous fashion, the incoming traffic which can then be efficiently and rapidly processed with an incremental grid clustering to identify the anomalies. The results show that ORUNADA can indeed process in continuous the incoming traffic and can thus be applied online and reach near real-time performance.

ORUNADA is open source and is available online at https://gitlab.com/ontic-wp4/ORUNADA. Furthermore, UNADA is also open source and is available online at https://gitlab.com/ontic-wp4/UNADA. Details on how to use these packages are provided in Annex A.

5.2.1 The feature space update

Usually, network anomaly detectors perform detection on consecutive large time-bins of network traffic, which implies that a long time may elapse between the anomaly occurrence and its detection. To overcome this issue, we propose to use a time sliding window in association with an unsupervised network anomaly detector. The proposed method is generic and could be used with any sufficiently fast and efficient detector.

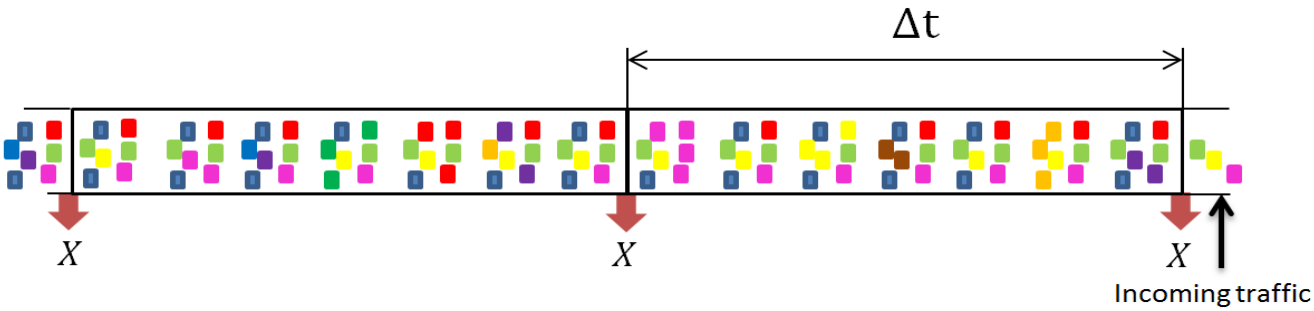

Figure 34 : Feature space computation at the end of each time-bin of Δt seconds

To detect anomalies, consecutive time-bins of incoming network traffic are preprocessed. The traffic is collected during a slot or a time-bin of Δt seconds which must be large enough to catch flows patterns. Some evaluations in [21] have shown that a slot of 15 seconds maximize the detection's performance. The traffic is then aggregated in flows with an aggregation key which can be for example the IPsrc or the IPdst with a mask /32, /24, /16, /8. Each flow is then represented by a set of A features in a vector x_f. The set of vectors of every flow is denoted by a normalized matrix $X = (x_1, \dots, x_F)$ representing the features space with F being the total number of flows. The detector is then applied on the matrix X to identify the anomalies. The process of consecutive time-bins is illustrated in Figure 34.

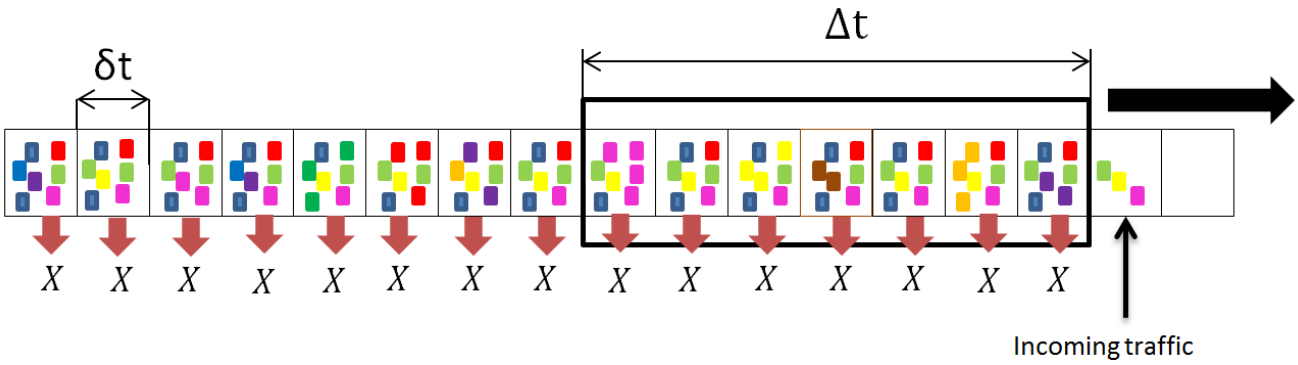

Figure 35 : Feature space's update at each micro-slot using the time sliding window

In order to detect the anomalies sooner and to avoid that attacks damage too deeply the network, we propose to update the feature space and launch the detector in a near continuous way, i.e. every micro-slot of δt ms. However, with a feature space computed on only the network traffic contained in a micro-slot, a detector may be unable to discriminate normal from anomalous patterns; the feature space may not contain enough information to identify flows' pattern.

To overcome this issue, a time sliding window of Δt s is set, which slides every δt ms. Each time the window slides, a new feature space X is computed with the network traffic contained in the current window (see Figure 35). To fasten the feature space computation, a micro feature space M_i is associated to each micro-slot of a window. This micro feature space is

computed with the packets contained in the micro-slot. The current window stores in a FIFO queue $Q = (M_1, \dots M_m)$ the micro feature spaces associated to each of its m micro-slot, M_m is the micro-feature space associated with the window's oldest micro-slot and M_1 with the newest. m is the number of micro-slots per window and is equal to $\Delta t/\delta t$. When the window slides, a new feature space X_{new} can be efficiently computed as follows:

$$X_{new} = X_{old} + M_{new} - M_m$$

where X_{old} is the previous features space, M_{new} is the micro feature space of the new micro-slot and M_m is the oldest micro feature space stored in Q. After the update of the matrix X, the queue M is updated (M_m is removed and M_{new} is inserted in the queue) and the detector is launched with the new feature space X_{new}.

This method can be applied only if the algorithm is sufficiently fast to process the feature space X in less than δt ms. To reach this goal, we devised a new incremental version of UNADA on top of the time sliding window, ORUNADA. The idea is to replace DBSCAN by an incremental grid clustering algorithm IGCA (Incremental Grid density-based Clustering Algorithm) described in [22] which exhibits a low complexity while it efficiently updates existing partitions.

5.2.2 An Incremental unsupervised network Anomaly Detection

UNADA suffers from high complexity mainly induced by the clustering step during which each subspace is partitioned with DBSCAN. Indeed, DBSCAN's complexity is in $O(n^2)$ where n the number of points to partition. This latter can be improved in $O(n.\log(n))$ by using a multi-dimensional index like a R-tree or a R*-tree [23]. These tree data structures are optimized for indexing spatial data: they can efficiently store and query spatial objects like points and rectangles.

To further improve the complexity of the clustering step, we focus our attention on grid clustering algorithms which complexity is often linear with the number of points. A grid clustering algorithm divides the space in cells which form a grid (hence its name). Instead of clustering directly the points as usual clustering algorithms do, a grid clustering algorithm partitions the cells where points are placed. As the number of cells is usually very inferior to the number of points to partition, these solutions significantly improve the clustering's execution time compared to standard approaches.

Among available grid clustering algorithms, GCA (Grid density-based Clustering Algorithm) [22] offers many advantages; it is a density based grid clustering, able to discover any shape of clusters and to identify noise. It takes as input two parameters, l the length of each cell and minDensePts, the minimum number of points a cell has to contain to be considered as dense enough to be partitioned. We had another parameter minPts which represents the minimum number of points that a group of dense cells must possess to be considered as a cluster. We advise to set minDensePts at 1, so that GCA has a behavior as close as possible to DBSCAN. Let c be the number of cells, c_{ne} the number of non-empty cells and c_d the number of cells which are dense enough to be partitioned, GCA's time complexity is $O(n + c_d.\log(c_{ne}))$. As usually $c_d < c_{ne} \ll c \ll n$ holds, GCDA is much faster than DBSCAN. Thus, GCDA should replace efficiently DBSCAN in UNADA to improve its execution time.

In the following, we make the assumption that, at each update of the feature space, i.e. at every micro-slot, the feature space changes slightly. This assumption is clearly realistic as few packets arrive and leave a sliding window of a dozen of seconds in a few milliseconds. As the

matrix X changes slightly from one micro-slot to another; it would make sense to update the partition of the feature space X instead of re-computing it entirely at each micro-slot.

Incremental clustering algorithms allow updating previous clusters when few data change. These solutions take benefit from the fact that deleting a point or adding a point affects the current partition of the data space only in the neighborhood of the point; it can then be efficiently updated by re-computing a few points. There exists an Incremental version of GCA (IGCA) which, thanks to its grid and incremental nature, can update efficiently an existing partition with a low complexity. It takes as input the same parameters of GCA. By using IGCA, the updated feature spaces could be efficiently processed and the anomalies rapidly identified.

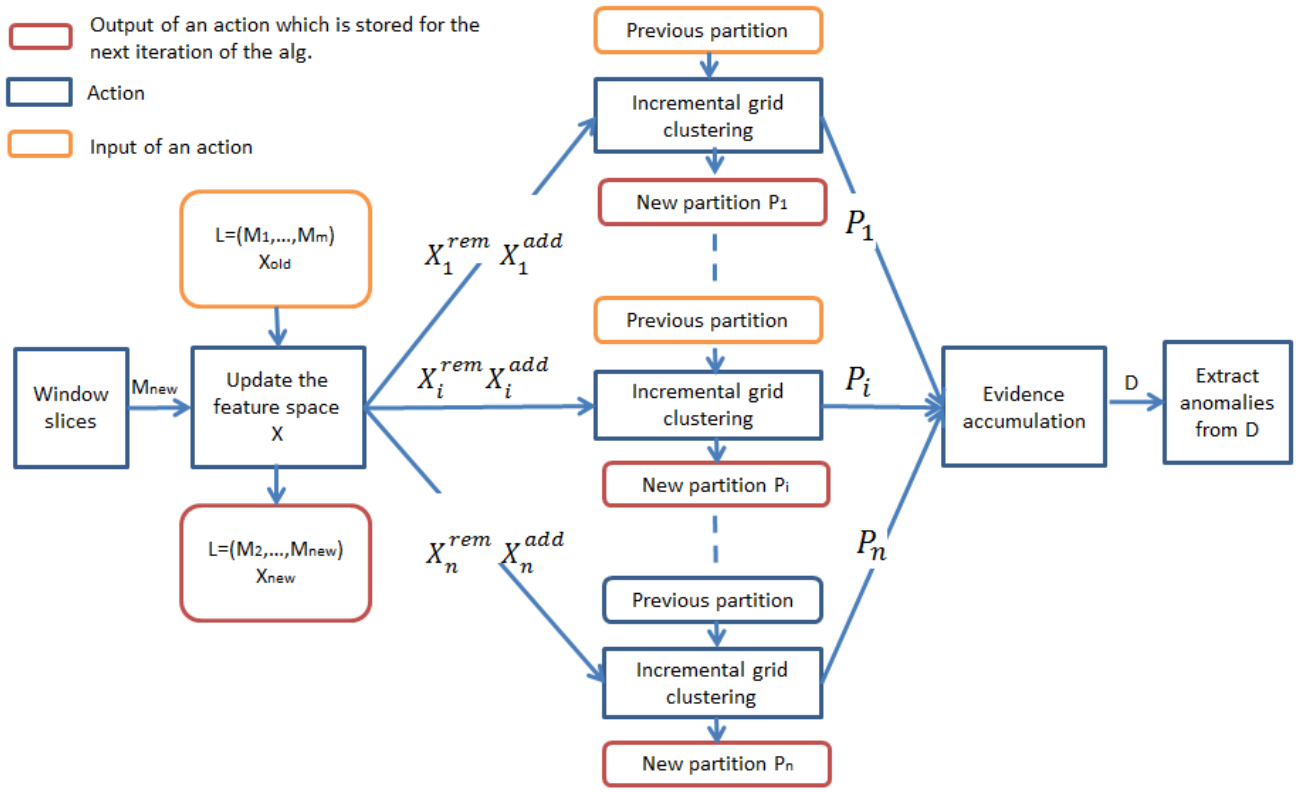

Figure 36: ORUNADA's operation

The online and incremental version of UNADA (ORUNADA) takes advantage of both the time sliding window and the incremental grid clustering algorithm IGCA. ORUNADA can be divided in three steps. The preprocessing step during which the feature space X is updated every micro-slot. The clustering step divides the feature space X in N subspaces of dimension 2 $(X_1, X_2 .. X_n)$ and updates the partition of each subspace to identify outliers. To update the partition of a subspace, IGCA needs to know the points to add and the points to remove from the previous partition. Thus, for each subspace i, two matrices are provided in order to update its partition: X_i^{rem} and X_i^{add} which describe respectively the points (or flows) to remove and to add. The updated feature space matrix X_i^{new} of a subspace i can be computed as follows:

$$X_i^{new} = X_i^{old} - X_i^{rem} + X_i^{add}$$

For each subspace i, IGCA outputs a new partition P_i. These partitions are then combined using an EA algorithm for Outliers identification (EA4O), which has been previously described

in section 5.1.1. and which outputs a dissimilarity vector D. This dissimilarity vector associates to each flow a score of abnormality. Flows which score is beyond a certain threshold are considered as anomalies, this threshold is fixed at the change in the slope of the sorted dissimilarity vector. Figure 36 gives an overview of ORUNADA's operation.

5.2.3 Evaluation of ORUNADA

We evaluate ORUNADA with the same network traces used by Pedro Cazas et al. [15] to study UNADA's performance in terms of detection: the MAWI network trace from the 30th of June 2006 (see Figure 29 for the results obtained by Pedro et al. [15]). This trace lasts fifteen minutes and is made up of 7.3M packet headers. As previously, the aggregation is done at the IP source /16 and the time-bin is set at 15 seconds. The evaluations have been performed on a single machine with 16 Gbit of RAM and an Intel Core i5-4310U CPU 2.00GHz. These evaluations aim at answering the three following questions?

1. By which factor UNADA can be speedup with a grid clustering like GCA?
2. Does grid clustering impact the detection performance of UNADA?
3. Can ORUNADA detect in near real-time the anomalies?

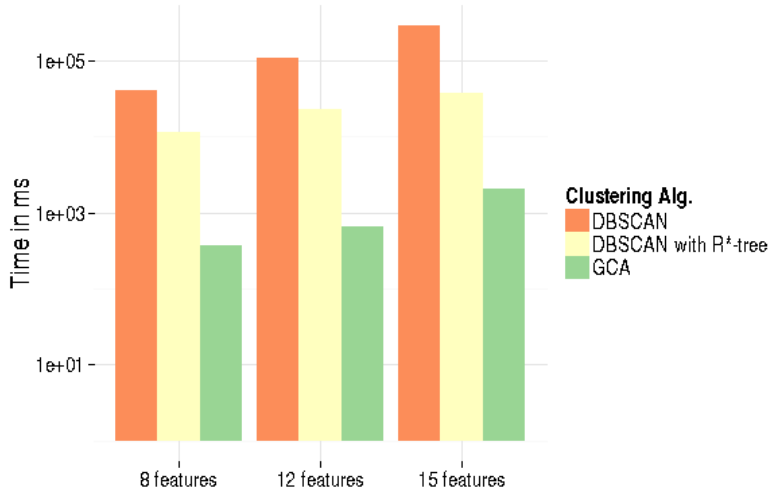

Figure 37: Execution time of UNADA

Figure 37 depicts the mean execution time (over 60 experiments) of UNADA according to the number of flows' features and the clustering algorithm used. The y-axis has a log-scale so that the execution time of UNADA with GCA can be observed. The graph shows that GCA improves clearly the execution time of the detector. This observation is confirmed by Figure 38 which displays the speed up factor of UNADA with GCA compared to DBSCAN and DBSCAN with R*-tree. Indeed, it speeds up the execution time by a factor of at least 100 for DBSCAN and 18 for DBSCAN with an R*tree. Furthermore, it can be noticed that the detector execution time could be further improved by distributing the computation of every subspace on a cluster of servers as proposed in PUNADA.

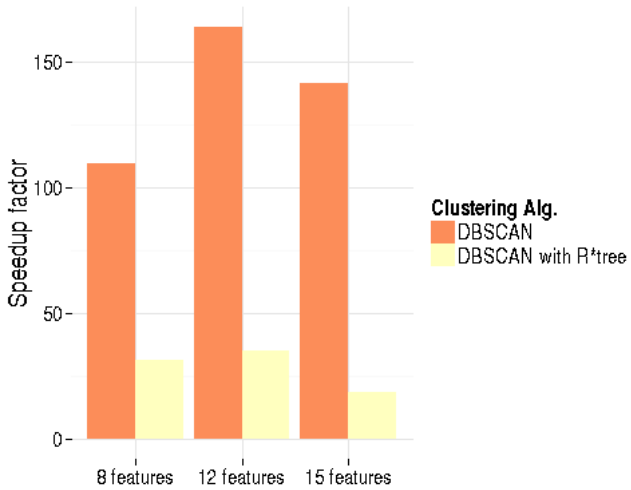

Figure 38 : Speedup factor of UNADA with GCA compared to DBSCAN and DBSCAN with R*tree

This gain in execution time could be at the cost of a degradation of UNADA detection. To determine whether GCA degrades UNADA's detection performance, we compare the similarity of the anomalies found by UNADA-CGA and UNADA-DBSCAN with 17 features and an aggregation at the IPsrc/32. To compare the set of anomalies found by these two detectors, we use the Jaccard index. It is a statistic which reflects the similarity between two sample sets. Let A be a first set and B a second set, the similarity between A and B according to the Jaccard index is computed as follows:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

If the index is close to one then the two sets are very similar and if it is close to 0 then they are considered as very dissimilar. However, this index does not tell anything about the anomalies ' rank similarity, to overcome this issue we use a second similarity measure the Spearman's rank correlation coefficient. Let a set of n elements, each element is associated to two ranks x_i and y_i which are stored in two different vectors X and Y. The Spearman rank correlation between these two vectors is computed as follows:

$$\rho = 1\frac{6.\sum d_i^2}{n(n^2-1)}$$

where $d_i = x_i - y_i$ is the difference between the two ranks given to an element i and $x_i \in X$ and $y_i \in Y$. However, as the set of anomalies for UNADA-GCA and UNADA-DBSCAN may be slightly different, we have modified the formula such as when a flow is an anomaly for only one detector d_i is set as $d_i = x_i - (\text{maxRank}(y) + 1)$ where maxRank(y) is the maximum rank of the vector Y. To point out the difference between the two clustering algorithms (DBSCAN and IGCA), we also compute the Jaccard index and the Spearman rank correlation coefficient for every flow which has been detected as an outlier in at least one subspace. A Spearman rank correlation of 1 reflects a perfect ranking similarity and a Spearman rank correlation coefficient of -1, a complete dissimilar ranking.

To obtain the rank of a flow or of an anomaly the dissimilarity vector is sorted in descending order. As the dissimilarity vector D associates to each flow a score of dissimilarity, the rank of a flow equals its position in the sorted dissimilarity vector.

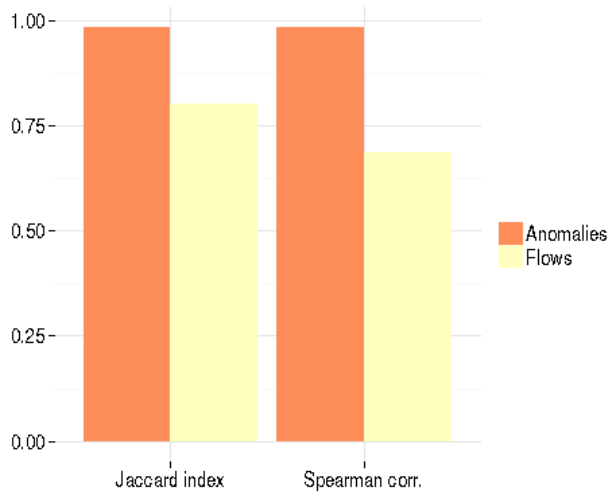

Figure 39: Similarity between UNADA-GCA and UNADA-DBSCAN

Figure 39 shows the mean similarity between the results obtained by UNADA-GCA and UNADA-DBSCAN. In terms of anomalies, it can be noticed that the Spearman rank correlation coefficient and the Jaccard index is quite close to 1 (0.98) which implies that using GCA instead of DBSCAN has nearly no impact on the detection performance of UNADA. We can also observe that these two detectors don't' find the same outliers in every subspace as the Jaccard index and the Spearman rank correlation coefficient is respectively of 0.8 and 0.68 for flows which have been found outliers in at least one subspace (in yellow on the figure). Thus, GCA and DBSCAN do not output the same partitions. However, they both detect extreme outliers, i.e, outliers which are very dissimilar in every subspace and are thus anomalous. As a consequence, the anomalies found by UNADA-GCA and UNADA –DBSCAN are (nearly) the same even though GCA and DBSCAN outputs different partitions. DBSCAN can be replaced by GCA in UNADA with nearly no impact on the detection performance, while improving its speed-up by a factor of 100.

To answer the third question, we evaluate ORUNADA execution time with different micro-slots sizes. The experiments have been performed using 15 features, see Figure 40. It can be noticed that a reduction of the micro-slot size improves ORUNADA mean runtime till reaching a threshold of around 0.2s. ORUNADA is able to process the incoming traffic faster than it arrives as long as the micro-slot size is superior or equal to 0.3 second. The results confirm our assumption that the feature space changes slightly from one micro-slot to another (otherwise they would have been no gain in UNADA runtime with the decrease of the micro-slot size). And most important, the results prove that ORUNADA can detect online and in near real -time anomalies (less than half a second elapse between an anomaly occurrence and its detection) with a detection performance equivalent to UNADA.

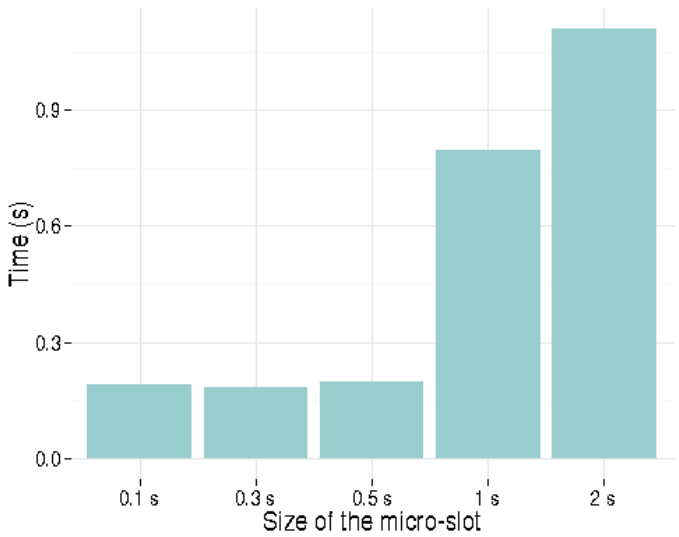

Figure 40: ORUNADA's execution time with different size of micro-slot

5.2.4 Conclusion and Future works

ORUNADA is an Online and Real-time Unsupervised Network Anomaly Detector which relies on a time sliding window to process the incoming traffic and on an incremental grid clustering to update efficiently the evolving feature space partition and. Our approach to compute flows' features based on a time sliding window is generic enough so that any detector which is fast enough can be implemented on top of it and thus detect anomalies in a continuous way. The feature space is processed by an incremental grid clustering algorithm which allows speeding up anomalies detection by a factor of at least 100 compared to usual clustering algorithm. ORUNADA evaluation show that our detector could detect an anomaly in less than half a second after its occurrence and that it could thus be used online and perform near real-time detection. To further improve its speed, we have planned to distribute its computation over a cluster of servers using tools from the Big Data world such as Spark Streaming. Furthermore, we would like to evaluate it on more recent data. In particular, on the ONTS traces collected in the context of the ONTIC project.

5.3 Study and Evaluation of existing unsupervised network anomaly detectors

This study has been motivated by the lack of existing independent evaluation of current unsupervised network anomaly detectors. It aims at pointing out current issues and challenges in the field of unsupervised network anomaly detection and at comparing the performance of a set of the most famous detectors. This study reveals some surprising results:

- Existing systems are sensitive to the tuning of their parameters, especially those which output scores, as there is no clear distinction between scores of anomalous and benign flows.
- There exists for most of the algorithms no rule to tune them. An important effort should be made to propose solutions for their optimal tuning. This study offers a beginning of tuning guidelines.
- The curse of dimensionality has a very low impact on detection performance as long as the algorithms do not rely on distance measures unless they specifically deal with high dimensions.
- The detection performance of a naïve algorithm is as good as, or even better than some of very complex solutions. This observation may be due to the fact that network anomalies are often flows which deviate strongly in at least one dimension and can therefore be easily identified.

In the following a big picture of network anomaly detectors principle is presented. Then, a set of unsupervised network anomaly detectors is described and the tuning of their parameters is discussed. The evaluations of these detectors are presented and the results are analyzed. The detectors are compared in terms of detection performance, detection similarity, execution time, parameters' sensitivity and curse of dimensionality.

5.3.1 Unsupervised network anomaly detection's principles

Existing unsupervised network anomaly detectors consist of two main steps, the preprocessing and the outlier detection step and an optional third one, the postprocessing step). The first step aims at preprocessing the incoming traffic which can be captured one or many links in consecutive time-bins. The packets are then aggregated in flows according to a specific flow key which can be, for example, the IP source, the IP destination, the port numbers, etc. Finally, a set of statistics are usually built to describe each flow like the number of IP destinations, of packets, of ICMP packets, of number of ports, etc. The choices of the time-bin's length, of the aggregation key and of the features may have a huge impact on detection performance; tuning of such preprocessing's parameters is out of the scope of this study. A normalized feature space matrix X of dimension $F * D$ is built, with D being the number of flows' features (or number of dimensions) and F the total number of flows.

The outlier detection step aims at detecting anomalous flows or outliers in the data set generated previously; the feature space X. Their goal is to identify flows which have different patterns from the rest of the traffic. This phase has received most of the researchers' attention as the detectors' intelligence relies in it. The outlier detection algorithm must be finely tuned to discover anomalies.

The postprocessing step aims at extracting and displaying information about the anomalies to assist network administrators in their task. This stage has received little attention for the moment even though it is crucial. Without any postprocessing step, a network anomaly detector may be useless for the network administrator who may be unable to understand, sort

and classify the spotted anomalies in order to take the appropriate counter-measures. This stage and its output can take different forms, for example in [15] the authors built signatures from the anomalies, in [17] they classify the anomalies using clustering techniques and in [24] they remove persistent anomalies to ease the network administrator's task.

5.3.2 Outlier detection algorithms

To identify flows which have different patterns from the rest of the traffic, unsupervised network anomaly detectors rely on outlier detection algorithms which can be classified in three categories [25]: algorithms based on statistical models, algorithms based on spatial proximity and finally algorithms which deal with high dimensions. In the following, d refers to the number of dimensions or features used to describe the flows and p to the number of points or flows in the data or feature space. These algorithms can either have a global view which implies that they consider the whole data to evaluate a point's abnormality or a local view, in which case only the neighborhood of a point is used for its evaluation. Furthermore, outlier detection algorithms can either output a label for each point (normal point vs abnormal point) or a score which reflects its outlierness. In the case of scores, the outlier detection algorithm must be followed by an additional step to extract anomalous points whose scores are above a certain threshold.

Outlier detection algorithms based on statistical models rely on the assumption that the feature space has been generated according to a statistical distribution. Outliers (anomalies) are then flows that deviate strongly from this distribution. Many statistical approaches have been applied to unsupervised network anomalies detection such as histograms [26], EM-clustering [27] and the subspace Principal Component Analysis (PCA) method [17]. The subspace PCA method assumes that the data follows a jointly Gaussian distribution. Statistical tests like the Q-statistic [28] or the chi-squared test [29] offers a solid theoretical framework to this approach. It could be objected that the network traffic and thus the feature space may not follow closely a jointly Gaussian distribution; however Jensen and Solomon point out that the Q-statistic test changes little even when the underlying distribution of the original data differs substantially from Gaussian [28].

Table 1: Comparison of outlier detection algorithms

Alg.	Output	View	Type	Deal with the Curse of dim.	Time Complexity	Parameters
PCA [17]	Scores	Global	Statistical	No	$O(d^3 + pd^2 + d.p^2)$	k: nb of PCs
DBSCAN [14]	Labels	Global	Spatial proximity	No	$O(p.log(p))$	r: radius minPts: min. nb of points
LOF [30]	Labels	Local	Spatial proximity	No	$O(p.log(p))$	nn: nb of neigh.
UNADA [15]	Scores	Global by subspace	Spatial proximity	Yes	$O(d^2.p)$	l: length minPts: min. nb of points
SOD [31]	Scores	Global by subspace	Spatial proximity	Yes	$O(d.p^2)$	nn: nb of neigh. l: nb of ref. points α: thresh. signif. of a dim (0.8)
Naive Alg.	Labels	Global by subspace	Statistical	Yes	O(p)	β: nb of std from the mean

The subspace PCA approach divides the whole space of dimension d (number of features) on two subspaces, the normal subspace made up of the k dominant principal components (PCs) and the abnormal or residual subspace made up of the d − k PCs left. There exist two variants of the subspace PCA method, however in this study we only evaluate the most popular one described in [17]. In this approach, one score of outlierness is computed for each point. A point's score is proportional to its distance, once projected on the abnormal subspace to the abnormal subspace. Points which are assigned a high score are more likely to follow a pattern which does not conform to the "normal or natural one'" and they are considered as outliers.
This approach takes as input one parameter k which defines the number of PCs of the normal subspace. Ringberg et al. [32] study different existing techniques for setting k and show that they are not reliable. Therefore, there exists, to our knowledge, no accurate rule to fix this parameter. However, their study points out that it must be picked such that the k dominant PCs capture most of the total deviation of the data. This method can be divided in three steps: (1) the computation of the feature space's principal components (2) the computation of the projection matrix onto the abnormal subspace and finally (3) the projection of the data on the abnormal subspace. The time complexity of these three stages is respectively $O(d^3 + pd^2)$, $O(k.p^2)$ and $O(d.p^2)$. As $k < d$, the overall complexity of PCA is $O(d^3 + pd^2 + d.p^2)$.

Many outlier detection algorithms rely on models based on spatial proximity like DBSCAN [14], K-mean [33], LOF [30] etc. Algorithms based on spatial proximity should be used with an index like the r-tree or the k-d tree to improve their time complexity. These detectors are based on the idea that points isolated from the other are outliers.
DBSCAN [14] is a density-based clustering algorithm which groups points that are closely packed together in clusters. Points that lie in low-density regions are considered as outliers. It can discover clusters of various shapes and sizes from a large amount of data which contains noise. It takes two parameters, a radius r which defines the neighborhood of a point and $minPts$ which defines the minimum number of neighbors for a point to be a cluster's core point. There is no consensus about the method to use in order to fix these parameters, especially since the tuning of these parameters may differ with the data and the problem considered. In order to avoid that DBSCAN groups flows which belong to similar anomalies in the same cluster (they won't then be detected as anomalies) $minPts$ must be superior to the maximum number of flows which could be induced by a same type of anomaly. Furthermore, as anomalies are flows which deviate strongly from the others, r must be chosen large enough so that points which are slightly different from the majority belong to a cluster. To reach this goal, we propose to set r as a percentage of the maximum distance between each pair of points and $minPts$ as a percentage of the total number of flows. Its time complexity is $O(p.log(p))$ [14].

LOF (Local Outlier Factor) [30] is a local spatial-based approach which assigns to each point an outlier factor, which represents its degree of outlierness regarding its local neighborhood. A point whose density is much lower than its nn nearest neighbors is considered as an outlier. Thus, LOF is able is to deal with regions of different densities. It takes as input one parameter nn, which represents the number of nearest neighbors considered to evaluate a point's abnormality. The value of nn must be carefully chosen. Indeed, if nn is too low, LOF may then compare an anomalous flow only with other anomalous flows of the same type and not detect them as outliers. To overcome this issue, nn must be set larger than the maximal number of flows induced by a same type of attack. For example, if there is at maximum 9 flows induced by SYN attacks, then nn should be fixed larger than 9. We propose to fix it as a percentage of

the total number of flows. For medium to high-dimensional data, the algorithm provides an average complexity of $O(p.log(p))$.

UNADA has been devised to deal with the curse of dimensionality by using subspaces and ensemble clustering techniques. It has been proposed by the LAAS-CNRS in [15] and presented previously in this deliverable. However for the sake of this section's completeness, it is described briefly. UNADA divides the whole feature space made up of d dimensions in $\binom{d}{2}$ subspaces of 2 dimensions. It then applies DBSCAN on each subspace. As proposed in section 5.1.2, we replace DBSCAN by GCA to improve UNADA's speed. It finally combines multiple partitions in one final partition, by summing the distance between each outlier with the center of the biggest cluster in every subspace. This sum represents the score of a point. It is all the more important that the point is an anomaly. It takes two parameters:

- minPts which represents the minimum number of points to form a cluster and can be set as a percentage of the total number p of points (or flows).
- l which represents the length used to divide each dimension in cells (or intervals), it can be set as a percentage of the average distance between points in each subspace.

The Subspace Outlier Degree (SOD) [31] is a local outlier detection which deals with high dimensions by selecting in an intelligent way subspaces to compute each point's score. It computes a score for each point which reflects how well it fits to the subspace that is spanned by a set of l reference points. The l reference points of a point are chosen such that they share a high number of nearest neighbors with the point. The subspace is then made up of the set of dimensions whose variance is low with respect to the set of l reference points. SOD takes three parameters: α that specifies a threshold to decide about the significance of a dimension, l the number of reference points and nn the number of nearest neighbors require computing the shared nearest neighbors. The authors advise to set α at 0.8. Furthermore, to avoid comparing an anomalous point with only similar anomalous points, l should be chosen much higher than the maximum number of flows related to a type of attack. Its time complexity is $(d.p^2)$.

For the sake of comparison, we propose a naive outlier detection algorithm which aims at detecting points with extreme values. For each dimension, this algorithm detects as outliers the points which are β standard deviations from the mean. As it deals with one dimension at a time, our naïve algorithm should be able to deal with high dimensions

For algorithms (LOF, subspace PCA, SOD, UNADA) which output scores, a final step is required to extract outliers. We identify, in the literature, three main methods to perform this task:

- The top k method which selects as outliers the k points with the biggest scores. However, for this technique to get good results, the number of anomalies should be known in advance which is rare.
- The knee method. This approach sorts the scores and plots them. It then searches the knee point on the plot. This knee point represents the threshold beyond which points are considered as outliers. However, there is no rule to choose this knee point and depending whether it is fixed at the beginning, at the middle or at the end of the knee, the obtained results can be very different and may lead to a very high number of incorrect classifications (see Figure 41). Furthermore, most articles don't provide any algorithm to perform this task. In some cases, the selection of the knee point is made by hand which prevents this method from being used at a large scale. A similar

approach consists of looking for a change in slope of this curve to fix this threshold or knee plot.

- The standard deviation method. This method aims at selecting points which score are far away from the others, i.e. which are δ standard deviation from the mean score or median score [31].

For most outlier detection algorithms, there exists no consensus on how to tune their parameters. Good sense and a well understanding of the current problem are essential to pick some relevant values for these parameters.

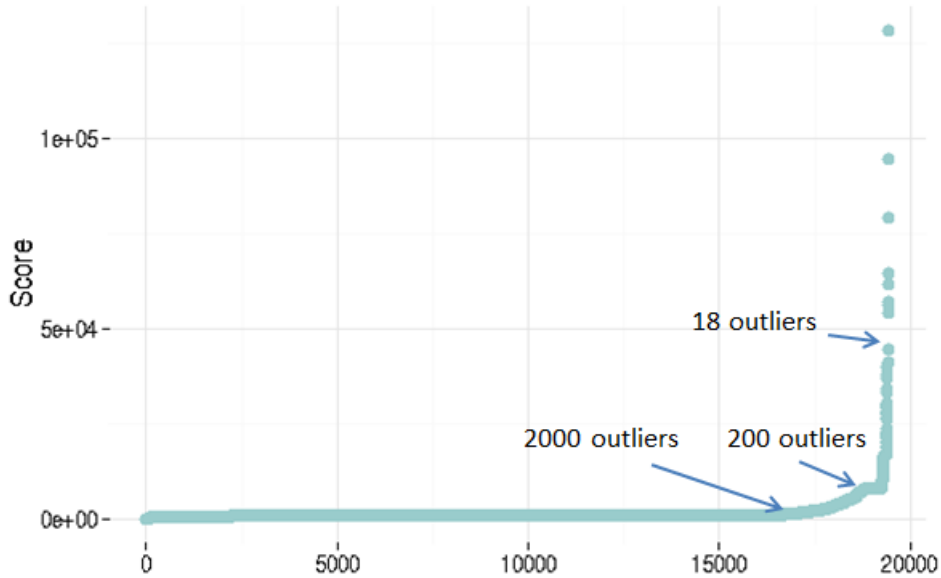

Figure 41 : Plot of the sorted score obtained with the subspace PCA method on an ONTS trace. It displays the value of three thresholds and the number of anomalies associated to each. These thresholds are obtained respectively at the beginning, middle and end of the knee.

5.3.2.1 Evaluation on the KDD99 data set

In the field of network anomaly detection, there is a lack of available public ground truth as pointed out in [34]. In the literature, two main public available ground truths are often cited: the KDD99 ground truth (summary of the DARPA98 traces) and the MAWI ground truth. The KDD99 contains multiple weeks of network activity from a simulated Air Force network, generated in 1998. Although the KDD99 dataset is quite old, it is still considered as a landmark in the field. On the contrary, the MAWILab data base is recent and is still being updated. It consists of labeled 15 minutes network traces collected daily from a trans-Pacific link between Japan and the United States. However, the MAWILab ground truth is questionable as it has been obtained by combining the results of four unsupervised network anomaly detectors [16]. Furthermore, the name of the labels given to the anomalies is not very relevant, for example many anomalies are labeled as 'HTTP traffic'. A manual inspection of these anomalies often does not provide a clear understanding of the reasons of their classification as outliers.

Our evaluation has been performed on the KDD99 dataset as it is the only public available ground truth that is fully accepted by the community and outputs consistent labels. The evaluation has been performed on the 10% KDD99 dataset as it contains 23 different types of attack. These attacks can be classified into 4 categories (DoS, probe, R2L, and U2R) see [35] for more information on these attacks. The packets have been aggregated in flows according to the famous four tuples (IP src, IP dest, port src and port dst) and are described by 41

attributes, 34 of which are numeric and 7 categorical. Every categorical variable has been turned into dummy variables, lifting the total number of features to 118 variables. The data set cannot be used as it is, due to a too large number of anomalous flows; in some cases there are even more anomalous flows than normal one. No detector based on outlier detection techniques can possibly detect the attacks as they are not rare. This problem could have been solved by aggregating the flows into another level (by IP source for example), but this is not possible with the KDD99 dataset as the IP addresses are not displayed. To overcome this issue, we have, selected randomly some flows, as in [15] and in [36], so that the percentage of attacks stays under a certain threshold. We build two data sets: the first one is made up of 1000 flows and includes 160 attacks and the second one is made up of 10000 flows and includes 979 attacks.

Table 2: Parameters' value used for the algorithms' evaluation

Alg.	Parameter	Value	Range of values
UNADA	length l	10% of max dist. between points	From 1 to 20%
	minPts	1% of the tot. nb of flows	From 1 to 20%
DBSCAN	radius r	10% of max dist. between points	from 1 to 30%
	minPts	not fixed	from 1 to 20%
LOF	nn	20% of the tot. nb of flows	from 10 to 40%
SOD	l	30% of the tot. nb of flows	from 10 to 50%
	nn	20%	from 10 to 40%
	α	0.8	fixed
PCA	k	90%	from 80% to 99%
Naive Alg.	β	not fixed	from 1 to 5

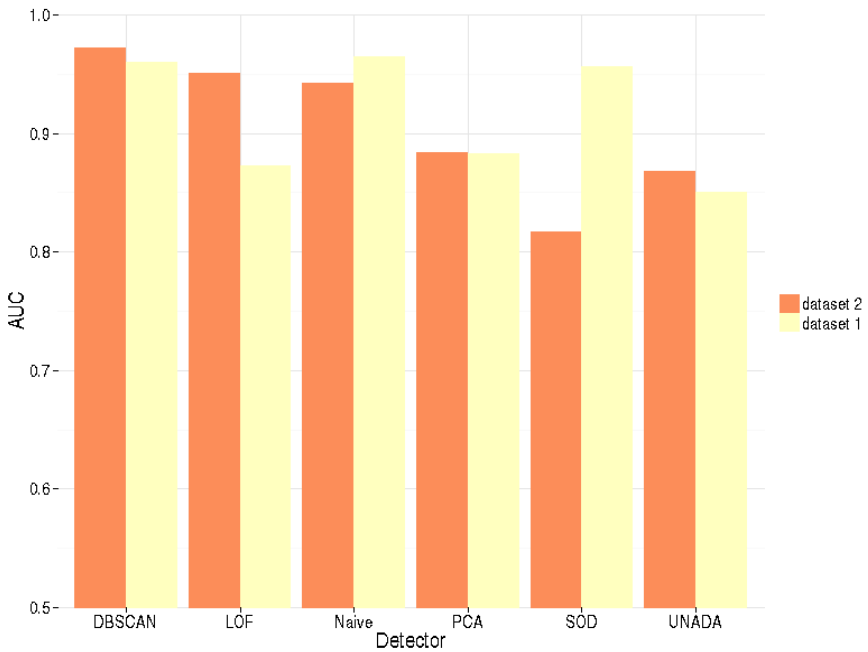

Figure 42 : Comparison of outlier detection algorithms using the AUC

Figure 42 displays the area under the ROC curve (AUC) obtained by each detector with 1000 (dataset 1) and 10000 flows (dataset 2). A ROC curve is obtained by plotting the true positives rate (TPR) against the false positive rate (FPR) at various threshold setting. The AUC takes its value in [0,1]; an AUC of 1 represents a perfect detector and an AUC of 0.5 a detector with

complete random guess. . The parameters used for this evaluation are displayed in Table 2. The points of the ROC curve have been computed by varying:

- The abnormal score threshold for the algorithms which output scores (SOD, LOF, UNADA, and PCA).
- The radius r for DBSCAN.
- The parameter β for the naive outlier algorithm.

It can be noticed that the naive algorithm's detection performs better than most detectors except DBSCAN (in both data sets) and LOF (in the first dataset only). This result can be explained by the nature of network anomalies which, in most cases, possess extreme values in at least one feature. DBSCAN and the naïve algorithm gets very good results in both data sets. We can observe that the detectors' performance varies from one dataset to another in particularly LOF's performance, which implies that the detection performance of a system may not be very stable. The AUC, like the ROC curve gives information about the proportion of TPR and FPR of each detector, but does not give any information about:

1. The numbers of false negatives (FNs). Indeed, many unsupervised network anomaly detectors suffer from a high number of false positives (FPs) which overwhelms the network administrator. This issue has been pointed out in [34].
2. The parameters' sensitivity. Indeed the AUC shows that in some configurations, a detector can have good results. However, it does not specify whether these configurations can be "easily" obtained and therefore whether a network administrator can be able to tune the algorithms' parameters in an optimized way. If a detector can't be tuned properly then it becomes useless even though it has a high AUC.
3. The similarity between the results obtained by the different detectors. It could be interesting to know whether the detectors find the same TPs and FPs. If they don't, it could be interesting to consider a combination of algorithms to improve the detection performance.

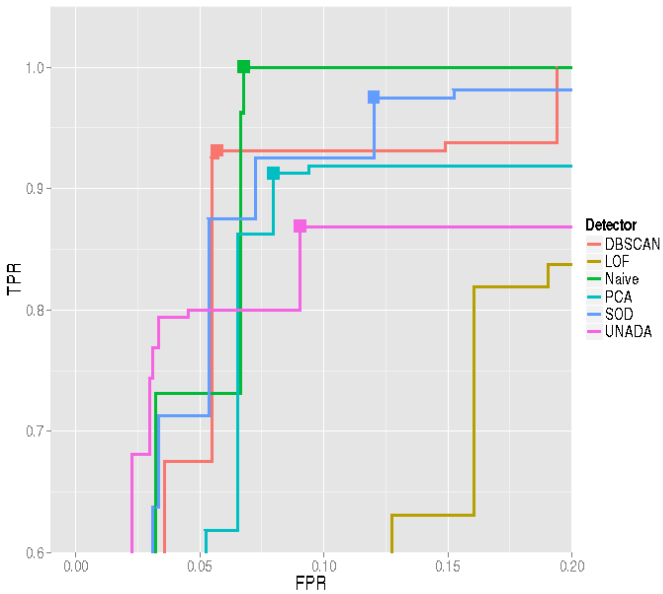

Figure 43 : First dataset ROC curve

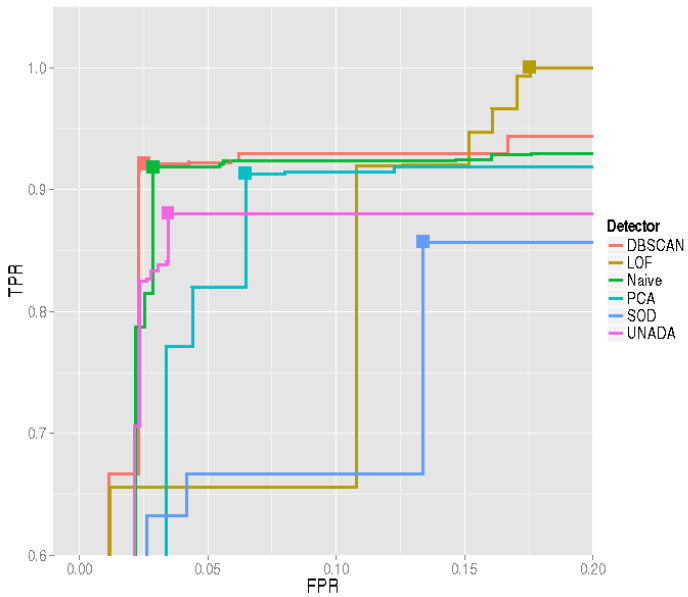

Figure 44 : Second dataset ROC curve

We decide to evaluate the number of TPs and FPs of each detector at its best setting. We consider that the best setting of a detector is the setting where the informedness of the

detector is maximal. . The informedness takes its values between 1 (good detector) and -1 (very bad detector) and is computed as follows:

$$informedness = TNR + TPR - 1$$

Figure 43 and Figure 44 depict the ROC curve of every detector for the first and the second dataset, respectively. The maximum informedness of a detector is represented by a point on its curve.

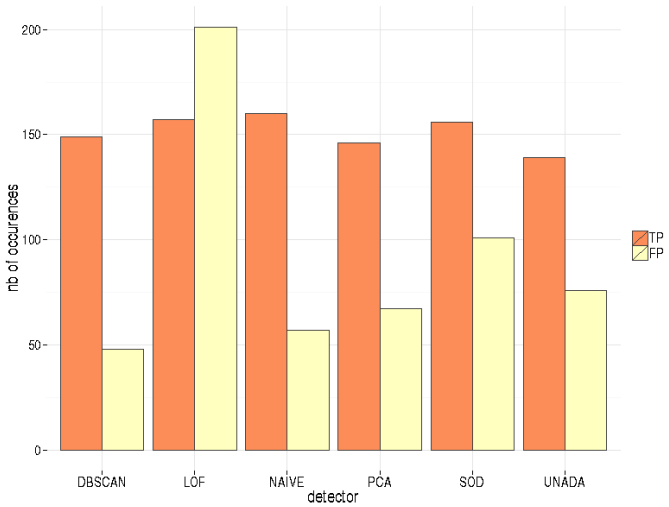

Figure 45 : Number of TPs and FPs obtained with the first dataset

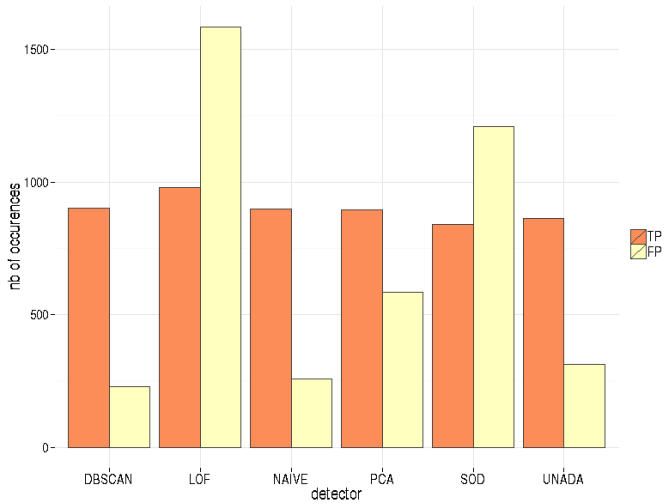

Figure 46 : Number of TPs and FPs obtained with the second dataset

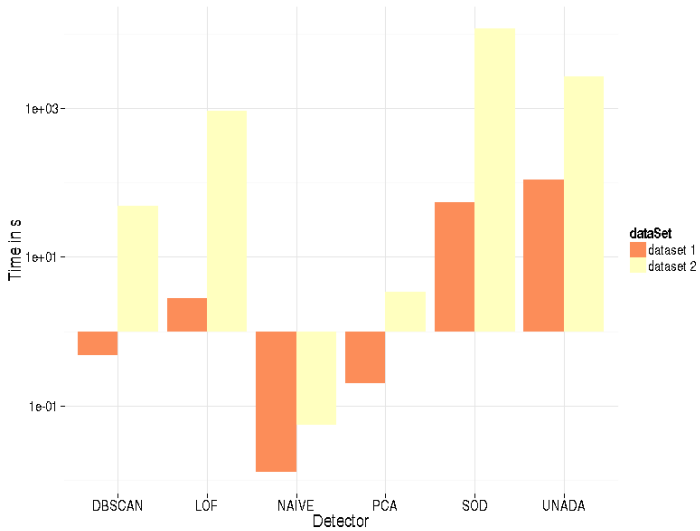

Figure 47: Execution time of the algorithms with dataset1 and 2

For each detector, we then evaluate the number of TPs and FPs it generates for each dataset (see Figure 45 and Figure 46). It can be noticed that even if a detector gets a high AUC, it can get a high number of false positives and may then be useless. This is the case of LOF in the second dataset, the AUC of LOF is 0.95 which implies that according to the AUC, LOF is a good detector. However, as it has more FPs than TPs (1583 FPs vs 979 TPs) it generates a lot of extra work for the network administrator and it can lead him to misclassify many flows.

Therefore, even though LOF has a good AUC, it may not (in the context of this dataset) be very useful for any network administrator due to its high number of FPs.

Figure 47 depicts the execution time of each algorithm for both datasets. These results have been obtained on a single machine with 16 Gbit of RAM and an Intel Core i5-4310U CPU 2.00GHz. The vertical axis has a logarithmic scale. It can be noticed that the naïve algorithm outperforms its competitors and that UNADA and SOD are very slow and they do not scale well with the number of flows. SOD is the only algorithm to exceed one hour to detect the anomalies in dataset 2; it takes 3 hours to complete.

To evaluate the similarity between the anomalies found by the different algorithms we use the Jacquard index. The Jaccard index measures the similarity between finite sample sets, and is defined as the size of the intersection divided by the size of the union of the sample sets. Thus, if A is the set of anomalies identified by a detector and B the set of anomalies identified by a second detector, their similarity is computed as follows:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

If the similarity is close to one then the detectors are very similar and if it is close to 0 then they are considered as very dissimilar. Figure 48 and Figure 49 display the similarity between the TPs of the different detectors for dataset 1 and dataset 2, respectively. It can be noticed that Jaccard index is high for every algorithms (all the square are close to red) in both datasets, which implies that the detectors mainly find the same anomalies. Figure 50 and Figure 51 display the similarity between the FPs found by the detectors for dataset 1 and dataset 2, respectively. One can observe that the Jaccard index is often very low (many squares are close to yellow), which implies that their FPs are different. Thus, it would be useful to consider combining the outputs of these different algorithms to keep only the anomalies found by most detectors. As the similarity between their FPs is very low, most FPs would then be discarded and as the similarity between their TPs is high, most TPs would be kept. Thus, combining the detectors would allow reducing the number of FPs while maintaining a high TPR.

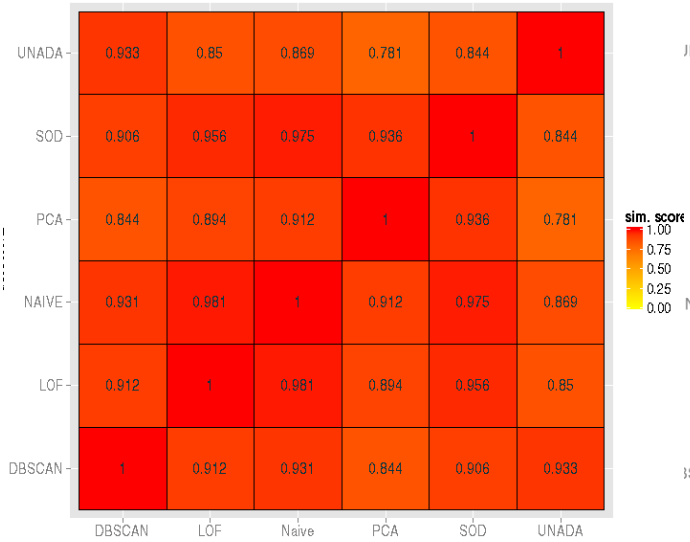

Figure 48 : Jacquard coefficient between the detectors TPs with dataset 1

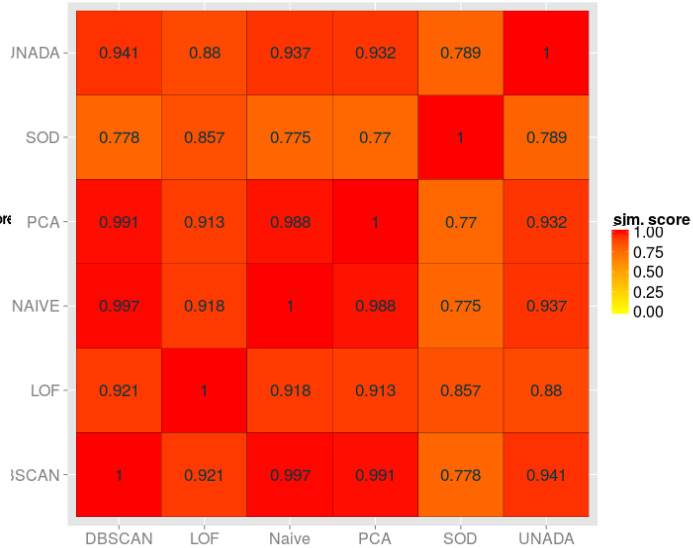

Figure 49 : Jacquard coefficient between the detectors FPs with dataset 2

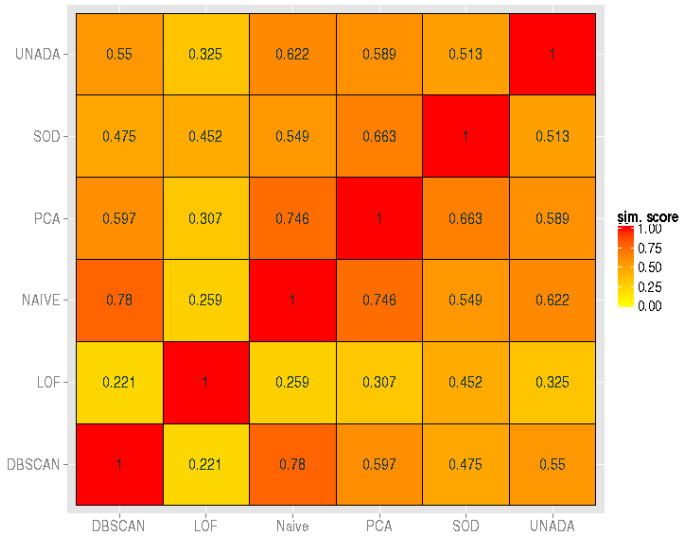

Figure 50 : Jacquard coefficient between the detectors' FPs with dataset 1

Figure 51 : Jacquard coefficient between the detectors' FPs with dataset 2

In order to evaluate the sensitivity of each detector's parameters, we use the most common approach of changing one factor at a time (OFAT). Thus, we assess the informedness of each detector while moving one parameter at a time. The other parameters are set such as the obtained informedness is maximal. Each parameter's entire range of possible values is described in Table 2. Each range is chosen such that the detectors attain good detection performance. For detectors which output scores, two techniques are used to extract the anomalies from the scores: the standard deviation and the knee method. For the standard deviation method, possible values for the standard deviation denoted std, are in [0.5, 3]. The knee is computed with the Kneedle algorithm described in [37].

Figure 52 depicts, using boxplots, the informedness obtained for each parameter of each detector. It can be noticed that LOF is very sensitive to its nn. parameter. Indeed, when nn is small an anomalous flow may be compared only with other anomalous flows induced by the

same type of attack (for example they may be all generated by Smurf attacks). Therefore, the anomalous flow may not be detected as anomalous. Furthermore, SOD gets very bad performance when *std* varies; this is due to the fact that SOD gets its best performance when *std* is much lower than the proposed range. Indeed, SOD gets is best performance when *std* =0.2, i.e. when all the points whose score is above 0.2 standard deviation from the median score are considered as anomalous. Thus, SOD does not differentiate clearly anomalous and benign flows as their scores are very close.

Furthermore, one can observe that most detectors which output scores are very sensitive to the std parameter used to extract anomalies. Indeed, there is not a clear boundary between the score of anomalous and benign flows. This parameter is very difficult to set; therefore the standard deviation method may not be an appropriate method to extract anomalies from scores.

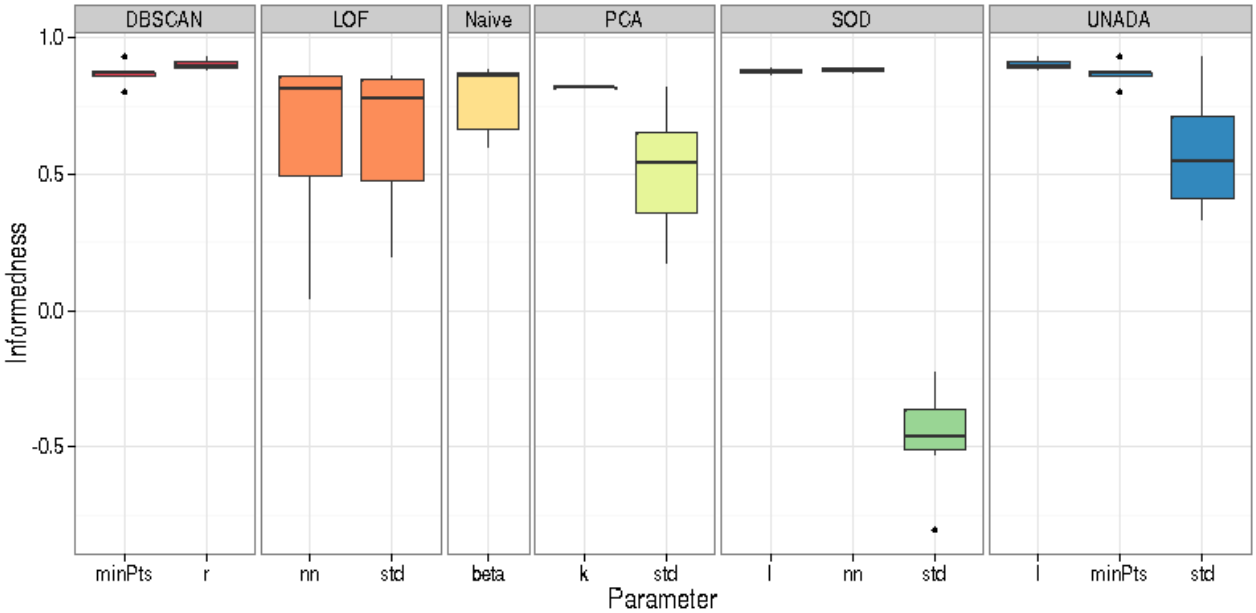

Figure 52 : Results of the sensitivity analysis performed with the standard deviation approach to extract outliers from the scores

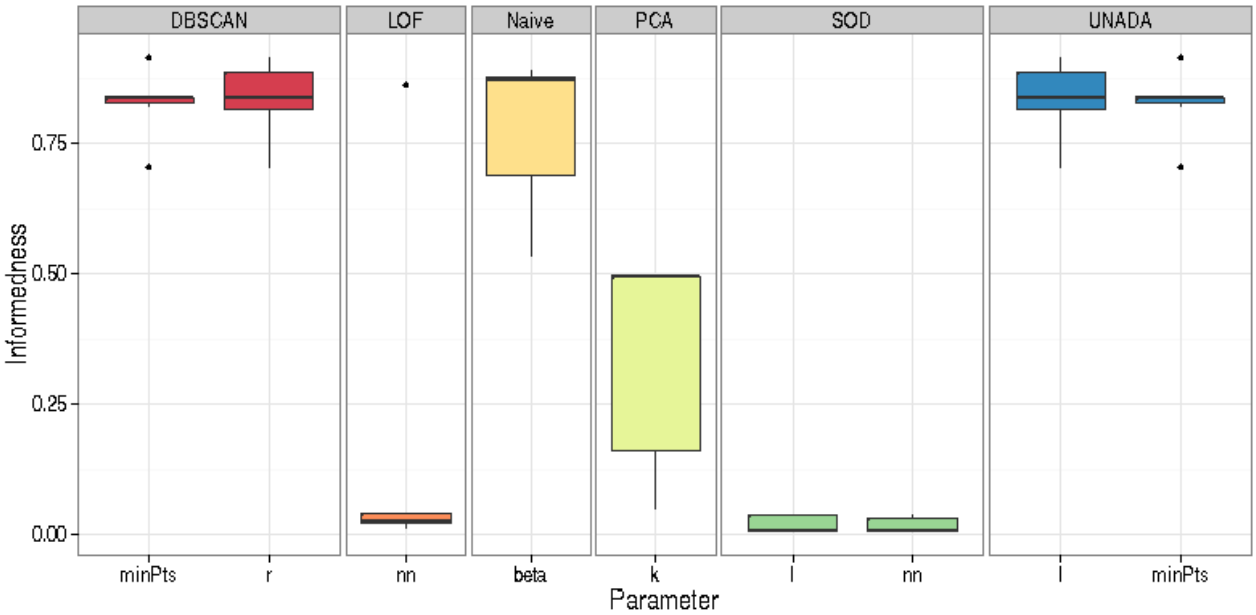

Figure 53: Results of the sensitivity analysis performed with the Kneedle algorithm to extract outliers from the scores.

Figure 53 shows the results of the sensitivity analysis performed with the Kneedle algorithm. As it can be seen, the performance of the detectors which output scores (PCA, LOF, SOD and UNADA) is very bad, except UNADA. This is due to the problem described previously related to the difficulty to set the knee value. Figure 54 and Figure 55 illustrate this issue by plotting the sorted scores obtained by LOF and the computed knee point value when the informedness is of 0.86 and 0,025, respectively. As for the standard deviation method, the knee approach fails in dividing the anomalous scores from the benign ones. Therefore, it is of central importance to propose new methods to extract anomalies from scores or to propose new detection systems which divide more clearly anomalous scores from benign ones. It can also be noticed that UNADA's performance changes slightly when the kneedle or the standard deviation method is applied (in contrary to other algorithms), which implies that UNADA may better differentiate anomalous and normal flows scores. This may be due to the fact that UNADA accumulates, for each flow, its degree of outlierness in every subspace.

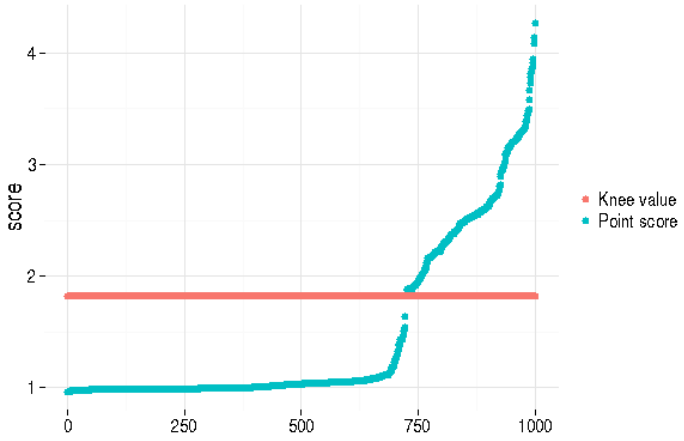

Figure 54 : Sorted LOF's scores (the informedness= 0.86)

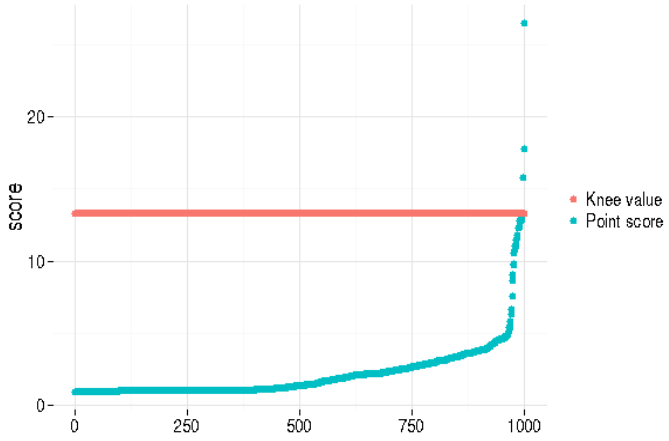

Figure 55 : Sorted LOF's score (the informedness= 0.025)

In order to evaluate whether the considered detectors deal efficiently with high dimensions, we have added some noisy dimensions to the data. As in [38], these noisy dimensions have been generated with a random uniform distribution which takes values in [0, 1]. Figure 57 and Figure 56 show respectively the FPR and TPR of each detector as a function of the number of added dimensions. UNADA, SOD and the naïve algorithm deal well with high and noisy dimensions, indeed UNADA and SOD have been specially devised to solve this issue while the naïve algorithm processes one dimension at a time. As it is not based on distance but on neighboring, LOF reacts well to the curse of dimensionality: a point's neighbors stay the same when new and noisy dimensions are added. DBSCAN is the only detector which suffers from the curse. Due to the curse, distance becomes meaningless and every point is considered as an outlier in DBSCAN. Before adding these noisy dimensions, the curse had no effect on DBSCAN, even though KDD99 has many dimensions. This phenomenon can be explained by the fact that each dimension in KDD99 brings information and no noise. Similar behaviors have been observed in [38].

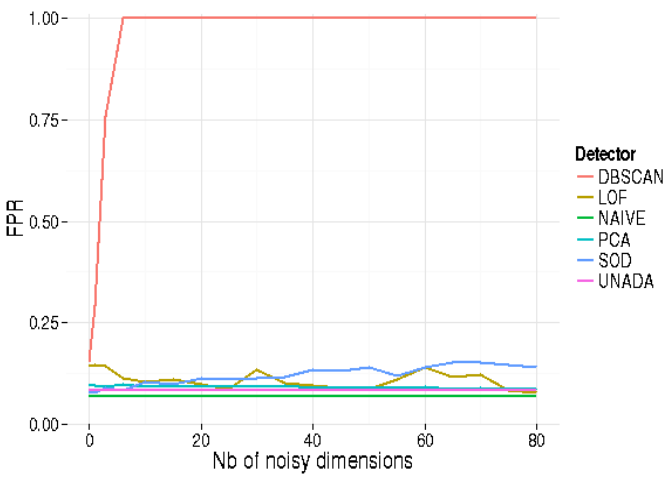

Figure 56: Detectors' FPR according to the number of noisy dimensions

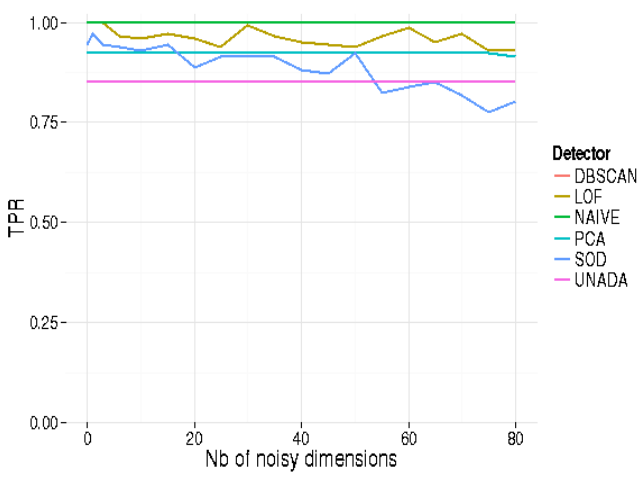

Figure 57: Detectors' TPR according to the number of noisy dimensions. DBSCAN's curve is hidden by the NAÏVE curve

5.3.3 Conclusions

This study has shown that most attacks have at least one extreme value in one dimension and that a very naïve outlier algorithm is able to get as good or even better performance than most existing detectors. Furthermore, the naïve algorithm has a very low complexity and deals well with the curse of dimensionality.

However, if an attack is more subtle the naïve algorithm may be unable to detect it. It should be then interesting to process twice the data, once to get rid of the anomalies which have extreme values and a second time to detect more subtle attacks in the data if there are any, with "smarter" algorithm.

Furthermore, we have pointed out that algorithms based on subspaces have a longer execution time like SOD and UNADA. However, UNADA's execution time can be improved by processing in parallel the subspaces like proposed in PUNADA.

The sensitivity analysis points out the lack of appropriate techniques to extract anomalies from scores. It shows that there is not a clear separation between scores of anomalous flows and benign ones. At the light of our results, UNADA seems to outperform other score-based detectors as it differentiates more clearly abnormal flows by accumulating for each flow, its degree of outlierness in every subspace. However, efforts should still be made to propose algorithms which extract anomalies from scores or which better separate anomalous from normal flows.

The similarity study offers a new perspective. It demonstrates that combining the output from different algorithms could help decreasing the rate of FPs.

Last, the network anomaly detection should be done online and in real-time on large network traffic in order to rapidly take appropriate counter-measures when an attack occurs and to gain advantage of this large amount of unexploited data. This could be achieved with an incremental approach relying on a continuous update of the feature space via a time sliding window as proposed in section 5.2.1.

6 Challenges and future works

6.1 Lack of Ground truths for network anomaly detection

Evaluation is a crucial step while building network anomaly detectors for proving their efficiency. However, it is a challenging task due to the lack of public available network data and ground truth. To our knowledge, there are two main available ground truths, the KDD99 ground truth (summary of the DARPA98 traces) and the MAWI ground truth. The KD99 dataset is quite old and has received many criticisms mainly due to its synthetic nature [39] but, it is still considered as a landmark in the field. On the contrary, the MAWILab data base is recent. However, its labels are questionable as they are obtained by combining the results of four unsupervised network anomaly detectors [16] and are often unintelligible, like the label ``HTTP traffic'' which contains many anomalies which seem after manual inspection harmless. In order, to overcome the lack of available dataset, researchers often build their own ground truth. We have identified three main techniques used in the literature, the manual inspection of network traces [17] [24] [40], the generation of synthetic traces via simulation or network emulation [41] [42] and the injection of anomalies in existing network traces [17]. None of these methods are perfect. They possess their own drawbacks and they cannot guarantee accurate evaluation study; the values of true positives and negatives and false positives and negatives cannot be exactly estimated. In manual inspection neither automated algorithms nor human domain experts can identify all the anomalies of a trace with complete confidence [41]. Furthermore, due to the fuzzy definition of a network anomaly, it is hard, even for an expert, to decide when a flow becomes an anomaly, i.e. when a flow becomes rare enough to be considered as an anomaly. On the other hand, to build synthetic traces, normal traffic needs to be modeled, however, existing models often fail to catch the complexity of this traffic and the generated traffic is often not realistic. The injection of anomalies consists in injecting anomalies in existing traffic. Furthermore, the injection must be well tuned so as to obtain network traces as realistic as possible.

6.2 Sensitivity of unsupervised network anomaly detection algorithms

Our comparison of current network anomaly detectors has pointed out that many detectors suffer from a high parameter's sensitivity, especially those which output scores. Some important effort should be made to overcome this issue. This could be done, for example, by devising algorithms which set automatically the detectors' settings or by adding a semi supervised network anomaly detector which could consider the network administrator's feedback to adjust its setting. An alternative approach would be to combine the outputs of different settings in order to improve detection performance.

6.3 Future works on anomaly detection

To evaluate our algorithms, we need a recent and reliable ground truth. However as pointed in section 6.2, there is an important lack of ground truths in this field, which can be explained by the sensitive nature of the data. Indeed, the inspection of network traffic can reveal highly sensitive information about an organization. In order to overcome this issue, we would like to build a ground truth with the ONTS traces using manual inspection and injection of synthetic traces of attacks. The challenge is to inject the good proportion of attacks so that the data stays realistic and not to miss out or misclassify anomalies during the manual inspection.

Furthermore, as our study of current network anomaly detectors has shown that most anomalies deviate in at least on dimension, we would like in the future to process the network

traffic in two times; once with a naive algorithm to identify and get rid of anomalies which possesses extreme values and may hide more subtle anomalies and then, with a more "intelligent" algorithm to detect more elaborate attacks in the data, if there is any.

Finally, we would like to process online and in real-time the incoming traffic. To reach this goal we would like to lean on the approaches proposed in ORUNADA, i.e. on an incremental algorithm and a continuous update of the feature space thanks to a time sliding window. The final solution should also take advantage of emerging Big Data tools for real-time analytics like Spark Streaming.

6.4 Future work on online feature selection and low-rank approximation

The work on feature selection carried out in WP2 and WP3 reveals an interesting fact: the flow-aggregated traffic in the ONTS data set contains small subsets of features that can provide a low-rank approximation of the full matrix close to the best existing equal-rank one. The algorithms that have been developed by the ONTIC consortium for finding such a subset can handle large data sets efficiently.

However, these algorithms have been designed for the offline setting. It would be desirable to explore the applicability of the same principles to online scenarios, where long-term storage is not possible. To this end, we plan to undertake the following research directions:

- Online leverage scores: The notion of statistical leverage score is a cornerstone of the researched methods for unsupervised feature selection. However, their computation is based on the right singular vectors of the data, although the QR decomposition can be used as well. In any case, the decompositions involved in the computation of the leverage scores require several passes over the data and are not suitable for online applications. It would be interesting to evaluate the applicability of existing one-pass factorization approximations and update schemes. Alternatively, other sampling scores could be devised.

- Long lasting models for self-representation: The column subset selection problem, which is the driving force behind most of the work that has been done by the ONTIC consortium in feature selection, is implicitly a self-representation regression model. It would be interesting to assess if a model of this kind trained offline could endure the passing of time. Otherwise, a periodical retraining approach could perhaps suffice to keep it up to date. This could be very useful for practical applications, since it would allow for the retrieval of rich data sets comprised of tens of features by just computing a handful of them, significantly alleviating the computations needed for real-time flow aggregation. The availability of the enterprise-grade cloud environment that the ONTIC consortium will have access to during the third year will make it possible to perform experiments for this purpose.

6.5 Enhancing network management

The algorithms developed in this work package take advantage of deep analytics on large and unexploited network traces for improving the network management in terms of QoS and security. By analyzing online large amount of traffic, these algorithms can help network administrators take appropriate decisions for enhancing network performance. Algorithms for traffic pattern evolution can help re-size a network for optimizing the network resource utilization. Algorithms for network anomaly detection can help detect attacks, on the fly in an unsupervised way, i.e. with no previous knowledge on the attacks. They can generate

automatically signatures to inject in routers for protecting the network. By using the same big data platform implemented with Spark, these algorithms could deal with large amount of data and take benefit of Spark's resilient, scalable and fault-tolerant features while mutualizing the resources.

7 References

[1] H. Liu, H. Q. Tian, D. F. Pan et Y. F. Li, «Forecasting models for wind speed using wavelet, wavelet packet, time series and Artificial Neural Networks,» *Applied Energy,* n° %1107, pp. 191-207, 2013.

[2] H.T Nguyen et I.T. Nabney, «Short-term electricity demand and gas price forecasts using wavelet transforms and adaptative models,» *Energy,* vol. 35, n° %19, pp. 3474-3685, 2010.

[3] J. Jin et J. Kim, «Forecasting Natural Gas Prices Using wavelets, Time Series, and Artificial Neural Networks,» *PloS one,* vol. 10, n° %111.

[4] H. Gavin, *The Levenberg-Marquardt method for nonlinear least squares curve-fitting problems,* Department of Civil and Environmental Engineering, Duke University, 2011, pp. 1-15.

[5] T. Otoshi, Y. Ohsita, M. Murata, Y. Takahashi et K. Shiomoto, «Traffic Prediction for Dynamic Traffic Engineering,» chez *Computer Networks*, 2015.

[6] R.J. Hyndman et Y. Khandaker, *Automatic time seriesfor forecasting: the forecast package for R,* Monash University, Department of Econometrics and Business Statistics, 2007.

[7] E. Castillo, B. Guijarro-Berdinas, O. Fontenla-Romero et A. Alonso-Betanzos, «A very fast leraning method for neural networks based on sensitivity analysis,» *The Journal of Machine Learning Research,* vol. 7, pp. 1159-1182, 2006.

[8] «LM transform,» [En ligne]. Available: http://people.duke.edu/~hpgavin/ce281/lm.pdf.

[9] M. Shafie-Khah, M. P. Moghaddam et M. K. Sheikh-El-Eslami, «Price forecasting of day-ahead electricity markets using a hybrid forecast method,» *Energy Conversion and Management,* vol. 52, n° %15, pp. 2165-2169.

[10] Z. Tan, J. Zhang, J. Wang et J. Xu, «Day-ahead electricity price forecasting using wavelet transform combined with ARIMA and GARCH models,» *Applied Energy,* vol. 87, n° %111, pp. 3606-3610, 2010.

[11] A. Patcha et J-M. Park, «An Overview of Anomaly Detection Techniques: Existing Solutions and Latest Technological Trends,» *Computer networks,* vol. 51, n° %112, pp. 3448-3470, 2007.

[12] L. Portnoy, E. Eskin et S. Stolfo, «Intrusion detection with unlabeled data using clustering,» *in Proc. of ACM CSS Workshop on DMSA,* pp. 5-8, 2001.

[13] D.Brauckho, B. Tellenbach, A. Wagner et M. May, «Impact of packet sampling on anomaly detection metrics,» chez *Proc. of the 6th ACM SIGCOMM Conference on Internet Measurement*, 2006.

[14] M. Ester, H.-P. Kriegel, J. Sander et X. Xu, «A density-based algorithm for discovering clusters in large spatial databases with noise,» *in Proc. 2nd International Conference on Knowledge Discovery and Data mining,* p. 226–231., 1996.

[15] P. Casas, J. Mazel et P. Owezarski, «Unsupervised Network Intrusion Detection Systems: Detecting the Unknown without Knowledge,» *Computer Communications,* vol. 35, n° %17, pp. 772 - 783, 2012.

[16] R. Fontugne, P. Borgnat, P. Abry et F. Kensuke, «MAWILab: Combining Diverse Anomaly Detectors for Automated Anomaly Labeling and Performance Benchmarking,» chez *Proc. ACM CoNEXT*, 2010.

[17] A. Lakhina, M. Crovella et C. Diot, «Mining anomalies using traffic feature distributions,» chez *Proc of the ACM SIGCOMM*, 2005.

[18] T. N. G. -. P. d. Torino, «TSTAT-TCP STatistic and Analysis Tool,» 2008. [En ligne]. Available: http://tstat.polito.it/. [Accès le 20 01 2016].

[19] Foundation, The Apache Software, «Spark, Ligthning-fast cluster computing,» [En ligne]. Available: http://spark.apache.org. [Accès le 21 12 2015].

[20] «Grid5000,» [En ligne]. Available: https://www.grid5000.fr. [Accès le 29 04 2015].

[21] J. Mazel, Unsupervised network anomaly detection, Toulouse: PhD in Network, Télécommunications, Systems and Architecture, 2011.

[22] N. Chen, A. Chen et L-X. Zhou, «Incremental grid density based clustering algorithm,» *Journal Of Software,* vol. 13, n° %11, pp. 1-7, 2002.

[23] N. Beckmann, H-P. Kriegel, R. Schneider et B. Seeger, «The R*-tree: An Efficient and Robust Access Method for Points and Rectangles,» *SIGMOD Rec.,* vol. 19, n° %12, pp. 332-331, 1990.

[24] K. Julish, «Clustering intrusion detection alarms to support root cause analysis,» *ACM Trans. Inf.Syst.Secur.,* vol. 6, n° %14, pp. 443--471, 2003.

[25] D. Fudenberg et D. Kreps, *Outlier Detection Techniques,* Columbus, OH: Tutorial at the 10th SIAM International Conference on Data Mining (SDM), 2010.

[26] M. Goldstein et A. Dengel, «Histogram-based Outlier Score (HBOS) : A fast Unsupervised Detection Algorithm,» *KI--2012: Poster and Demo Track,* pp. 59-63, 2012.

[27] I. Syarif, A. Prugel-Bennett et G. Wills, «Unsupervised Clustering Approach for Network Anomaly Detection,» *Network Digital Technologies,* vol. 293, pp. 135-145, 2012.

[28] D. R. Jensen et S. Herbert, «A Gaussian Approximation to the distribution of a Definite Quadratic Form,» *Journal of the American Statistical Association,* vol. 67, n° %1340, pp. 898-902, 1972.

[29] Mey-Ling Shyu, Shu-Ching Chen et Kanoksri Sarinnapakorn, «A novel Anomaly detection scheme based on principal component classifier,» chez *Proc. of the IEEE Found. and New Directions of Data Mining Workshop in conjunction with ICDM'03*, 2003.

[30] M. Breunig, H-P Kriegel, R. Ng et Jörg S., «LOF: Identifying Density-based Local Outlier,» *SIGMOD REC.,* vol. 29, n° %12, pp. 93-104, 2000.

[31] H-P. Kriegel, P. Kröger, B. Kijsirikul, N. Cercone et T. Ho, «Outlier Detection in Axis-Parallel Subspaces of High Dimensional Data,» *Advances in Knowledge Discovery and Data Mining,* vol. 5476, pp. 831-838, 2009.

[32] H. Ringberg, A. Soule, J. Rexford et C. Diot, «Sensitivity of PCA for Traffic Anomaly Detection,» *SIGMETRICs Perform. Eval. Rev.,* vol. 35, n° %11, pp. 109-120, 2007.

[33] Amuthan Prabakar Muniyandi , R. Rajeswari et R. Rajaram, «Network Anomaly Detection by Cascading K-means Clustering and C4.5 Decision Tree algorithm,» *Procedia Engineering,* vol. 30, pp. 174-182, 2012.

[34] R. Sommer et V. Paxson, «Outside the Closed World: On using Machine Learning for Network Intrusion Detection,» *IEEE Symposium on Security and Privacy,* pp. 305-316, 2010.

[35] Swati Paliwal et Ravindra Gupta, «Denial-of-Service, Probing & Remote to User (R2L) Attack Detection using Genetic Algorithm,» *International Journal of Computer Applications,* vol. 60, n° %119, pp. 57-62, 2012.

[36] Jiong Zhang et M. Zulkernine, «Anomaly based Network Intrusion Detection with Unsupervised Outlier Detection,» chez *Communications, ICC'06. IEEE International Conference on*, Istanbul, 2006.

[37] V. Satoppa, J. Albrecht, D. Irwin et B. Raghavan, «Finding a "kneedle" in a Haystack,» chez *in Proc. ICDSW*, 2011.

[38] A. Zimek, E. Schubert et H-P Kriegel, «A Survey on Unsupervised Outlier Detection in Higth-Dimensional Numerical Data,» *Statistcal Analysis and Data Mining,* vol. 5, 2012.

[39] J. McHUGH, «Testing Intrusion Detection Systems: A Critique of the 1998 and 1999 DARPA Intrusion Detection System Evaluations As Performed By Lincoln Laboratory,» *ACM Trans. Inf. Syst. Secur.,* vol. 3, n° %14, pp. 262-294, 2000.

[40] Silveira, F et Diot, C, «URCA: Pulling out anomalies by their root causes,» chez *in Proc. INFOCOM*, 2010.

[41] H. Ringberh, M. Roughan et J. Rexford, «The Need for Simulation in Evaluating Anomaly

Detectors,» *SIGCOMM Comput. Commun. Rev.,* vol. 38, n° %11, pp. 55-59, 2008.

[42] R. Agrawel, J. Gehrke, D. Gunopulos et P. Raghavan, «Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications,» *Data Mining and Knowledge Discovery,* vol. 11, n° %11, pp. 5-33, 2005.

Annex A : Documentation of the NTFF package

Link to the code repository: https://gitlab.com/ontic-wp4/NTFF

Description

Network Traffic Forecasting Framework (NTFF) combines data extraction and different time-series forecasting techniques with wavelet transform preprocessing, for Network Traffic predictions.

NTFF is divided in 3 modules as Preprocessing, Wavelets and Forecasting:

- **Preprocessing Module**: This module contains different tools for data extraction from .pcap captured files and preprocessing. That resultant data can be used with the Wavelets and Forecasting modules. Preprocessing tools:
    1. **Tstat**: a script to extract only tcp features from .pcap files.
    2. **FlowsAgregator**: an Apache Spark script in python to aggregate the number of flows by seconds from the output files of Tstat. There is also another script ("flowsAgregatorFromPcap.sh") that combines Tstat script and Agregation script in one.
- **Wavelets Module**: This module contains different Wavelet decomposition functions in Python. Wavelets techniques:
    1. **Stationary Wavelet Transform (with Haar wavelet)**: similar to the Discrete Wavelet Transform but without sub-samplings.
    2. **Stationary Wavelet Packet Transform (with Haar wavelet)**: similar to the Discrete Wavelet Packet Transform but without sub-samplings.
- **Forecasting Module**: This module contains different forecasting classes in Python. Forecasting techniques:
    1. **Artificial Neural Networks (ANN)** : with the training method "Sensitivity-Based Linear Learning Method" adapted for real numbers forecasting.
    2. **Autoregressive Integrated Moving Average (ARIMA)**: from the "forecast" package of R.

Also, in this API, you are able to apply different forecasting methods for different wavelet components.

How To

Prerequisites

Executed with **Operative System**: **CentOS 7.2.1511 (Core)** and **Kernel**: *_3.10.0-123.4.4.el7.x86_64* _ Using 4GB of RAM and 2 cores.

1. Python 2.7.5 Installed by default in the Operative System mentioned above.

2. Numpy 1:1.7.1-11.el7 Install: `yum install numpy`
3. gcc 4.8.5-4.el7 Install: `yum install gcc`
4. python-devel 2.7.5-34.el7 Install: `yum install python-devel`
5. Cython 0.23.4 Install: `easy_install cython`
6. PyWavelets 0.4.0 Install: `easy_install PyWavelets`
7. pip 7.1.2 Install: `easy_install pip`
8. ipython 4.0.2 Install: `pip install ipython`
9. epel 7-5 Install: `yum install epel-release`
10. R 3.2.3-1.el7 Install: `yum install R`
11. R packages : forecast and parallel Inside R,
    install: `install.packages("forecast")` `install.packages("parallel")`
12. readline-devel 6.2-9.el7 Install: `yum install readline-devel`
13. rpy2 2.7.7 Install: `pip install rpy2`
14. nbformat 4.0.1 Install: `pip install nbformat`
15. Apache Spark 1.4.1 Download: http://www.apache.org/dyn/closer.lua/spark/spark-1.4.1/spark-1.4.1-bin-hadoop2.6.tgz For more information : http://spark.apache.org/
16. Tstat 3.0 Download: http://tstat.polito.it/software.php For more information : http://tstat.polito.it/

Imports

To import functions from StationaryWaveletTransform.py:

```
from SWT.StationaryWaveletTransform import hspwt, hswt
```

To import functions and classes from WPF.py:

```
from WPF import WaveletPrediction, getWaveletByName, WPFAlgorithm
```

To import the SBLLMF class from SBLLMF.py:

```
from SBLLMF.SBLLMF import SBLLMF
```

To import the ARIMA class from ARIMA.ipynb:

```
from IPython import get_ipython

get_ipython().magic(u'run ./ARIMA/ARIMA.ipynb')
```

Forecasting Example

There is Example.py an example program that executes 2 forecasting algorithms with SWT and SWPT and without them for the ExampleDataset.csv.

The input dataset "ExampleDataset.csv" is a vector of active sessions in each second during the day of 4 of April of 2015 of the ONTS dataset.
To run the program (in the same directory as the Example.py): `ipython Example.py`

The output will be written in to the standart output.

- All output lines that starts with [+] or [-] are informative logs.
- The lines that starts with Result, indicates the output of the forecast.
- The lines that starts with Expected, indicates the values of a perfect forecast (expected values).
- The lines that starts with Absolute Error, is the absolute value of the error of each predicted step
- The lines that starts with Percentage Absolute Error, and the percentage of the absolute error respectively with the expected output.

Example of the output for SBLLMF without any wavelet transform:

```
SBLLMF  without wavelet transform:

--------------------------------------------------------------------------------

 [-] Loading datasets...

 [+] Loaded datasets...

[-] Construction Wavelet Prediction Framework object and training it...

[+] Constructed and trained.

[-] Predictiong the next 4 steps...

Result:  [ 4049.83128171  4070.82738591  4094.24900881  4120.37005851]

Expected:  [ 4055.  4095.  4083.  4024.]

Absolute Error:  [  5.16871829  24.17261409  11.24900881  96.37005851]

Percentage Absolute Error (%):  [ 0.12746531  0.59029583  0.27550842  2.39488217]

[+] Done.

--------------------------------------------------------------------------------
```

Preprocessing Example

For this phase we used:
Apache Spark : 1.4.1 Tstat : 3.0 (Spirit of St. Louis flavor) -- Mon Jun 22 18:22:37 CEST 2015
Example of use: Taking in account some of the system variables: `$TSTAT` is the directory of the Tstat executable. `$SPARK_HOME` is the directory of the installed Apache Spark. `$PCAPD` is the directory where are some .pcap files.

The complete procedure of transform some .pcap files into a dataset that aggregates the number of flows per second:

```
bash ./Preprocessing/flowsAgregatorFromPcap.sh $TSTAT/tstat $SPARK_HOME/bin/pyspark ./O
UT1 ./OUT2 $PCAPD/*.pcap
```

`$TSTAT/tstat` : Tstat executable `$SPARK_HOME/bin/pyspark` : Pyspark executable `./OUT1` : Temporal Tstat output directory `./OUT2/part-00000` : Flows/second dataset output file`$PCAPD/*.pcap` : Capture .pcap input files
Now you can use the output file `./OUT2/part-00000` for the Forecasting example file with different forecasting steps:

```
ipython Example.py ./OUT2/part-00000 10
```

`./OUT2/part-00000` : Flows/second dataset output file of the previous script. 10: Number of steps for the prediction.

Preprocessing Scripts

Tools of the preprocessing phase:

```
tstat_con.sh <tstat executable> <output dir> {<input file .pcap or .pcap.gz>}+
```

Description: apply Tstat with some preset options as the concatenation (specially useful if there are more than one .pcap file) and tcp data only. Parameters:

1. *tstat executable*: The executable file of Tstat.
2. *output dir*: Output directory for Tstat.
3. *input file .pcap or .pcap.gz*: One or more .pcap or .pcap.gz input files.

Output: Tstat statistics per tcp flow (a flow per line) in the `output dir`.

```
flowsAgregatorFromPcap.sh <tstat executable> <pyspark executable> <temporary dir> <outp
ut dir> {<input file .pcap or .pcap.gz>}+
```

Description: apply Tstat with some preset options as the concatenation (specially useful if there are motre than one .pcap file) and tcp data only. Parameters:

1. *tstat executable* : The executable file of Tstat.

2. *spark executable* : The executable file of Apache Spark.
3. *temporary dir*: Output directory for Tstat.
4. *output dir*: Output directory for FlowsAgregator.py.
5. *input file .pcap or .pcap.gz*: One or more .pcap or .pcap.gz input files.

Output: Flows/second dataset output file inside of the `output dir`.

```
$SPARK_HOME/bin/pyspark --master local ./FlowsAgregator.py <tstat executable> <output_dir> {<input file .pcap or .pcap.gz>}+
```

Description: Aggregates the number of flows per second from the output files of "tstat_con.sh".
Parameters:

1. *input dir* : The directory that contains statistics files generated by Tstat, inside the Tstat output directory.
2. *output_dir*: Output directory.

Output: Flows/second dataset file (similar to `ExampleDataset.csv`).

Functions and classes

This API present:

Stationary Wavelet Transform (./SWT/StationaryWaveletTransform.py):

```
def hswt(data, components):
```

Description: Apply Stationary Wavelet Transform (SWT) with Haar wavelet to an input signal.
Parameters:

1. *data* : (Numpy Array with one dimension) the input signal for preprocessiong.
2. *components*: (Integer) the number of components for the SWT.

Output: List with ("componenets"+1) 2-dimensional Numpy Arrrays as SWT outputs (1 column a row for each value).

```
def hspwt(data, components):
```

Description: Apply Stationary Wavelet Packet Transform (SWPT) with Haar wavelet to a input signal. Parameters:

1. *data* : (Numpy Array with one dimension) the input signal for preprocessiong.
2. *components*: (Integer) the number of components for the SWT.

Output: List with (2"componenets") 2-dimensional Numpy Arrrays as SWPT outputs (1 column a row for each value).

Sensitivity-Based Linear Learning Method adapted for Forecasting (./SBLLMF/SBLLMF.py):

```
class SBLLMF:
```

Description: Sensitivity-Based Linear Learning Method adapted for Forecasting (SBLLMF) is a forecasting model.

```
def __init__(self, data, lags,  hidden_units=None,

                  nlf=sig, inlf=invsig, dnlf=dsig,

                      parameters_amplitude=None, p = 1, ni=None, Q = None, MSE=None):
```

Description: constructor of the SBLLMF class, sets the main models parameters and the training dataset. Parameters:

1. *data* : (Numpy Array with two dimensions) the input signal for the training dataset, with one column and N rows.
2. *lags*: (Integer) the number of previous steps used during each one step prediction.
3. ***hidden_units*** : (Integer) is the number of neurons in the hidden layer (**by default:** is the number input values + 1).
4. ***nlf*** : (Function(Float):Float) is a nonlinear function with a float input and a float output (**by deafult:** is the sigmoidal function).
5. ***inlf*** : (Function(Float):Float) is an inverse of the nonlinear function **nlf** with a float input and a float output (**by default:** is the inverse of the sigmoidal function).
6. ***dnlf*** : (Function(Float):Float) is a derivate of the nonlinear function **nlf** with a float input and a float output (**by default:** is the derivate of the sigmoidal function).
7. ***parameters_amplitude*** : (Float) is the amplitude of the initial weights of the Neural Network (**by default:** is `2.0/(100*(lags+1))`).
8. ***p*** : (Float) is the effect of the hidden layer output update (**by defult:** is 1).
9. ***ni*** : (Float) is the amplitude of the initial error for the hidden layer of the Neural Network (**by default:** is `2.0/(100*(lags+1))`).
10. ***Q*** : (Float) is the initial Sensitive Error for the future reduction and convergence (**by defult:** is `100000`).
11. ***MSE*** : (Float) is the initial Mean Square Error for the future reduction and convergence (**by default:** is `10000`).

```
def train(self, epochs = 75, er0 = 0.00001, er1 = 0.00001, verbose=False):
```

Description: This is the training method of the SBLLMF class, which starts the training for the trainig dataset declared in the constructor. Parameters:

1. ***epochs***: (Integer) the number of maximum steps for the model training (**by defult:** is 75).
2. ***er0***: (Float) the minimal error for the Sensitive Error between two consecutive steps (**by defult:** is 0.00001).
3. ***er1***: (Float) the minimal error for the Mean Square Error between two consecutive steps (**by defult:** is 0.00001).
4. ***verbose***: (Boolean) True if we want to see log information about each iteration of the training (**by default:** is False).

Output: Nothing (Only update the model information).

```
def forecast(self, data, steps):
```

Description: This is the forecast method of the SBLLMF class, which predicts a number of future steps of the input signal. Parameters:

1. *data* : (Numpy Array with two dimensions) the input signal for the testing dataset, with one column and N rows.
2. *steps*: (Integer) the number of future points to predict.

Output: (Numpy Array with one dimension) the predicted signal for the testing dataset, with "steps" values.

ARIMA (./ARIMA/ARIMA.ipynb):

```
class ARIMA:
```

Description: Autoregressive Integrated Moving Average (ARIMA), is a forecasting model with calls to R functions of the "forecast" package.

```
def __init__(self,input_data, max_p=10, max_q=5, max_d=2):
```

Description: constructor for the ARIMA class, which sets the training dataset and some of the ARIMAS training limitations. Parameters:

1. *input_data*: (Numpy Array with two dimensions) the input signal for the training dataset, with one column and N rows.

2. ***max_p***: (Integer) the max value of the parameter p during the training (**by default:** is 10).
3. ***max_q***: (Integer) the max value of the parameter q during the training (**by default:** is 5).
4. ***max_d***: (Integer) the max value of the parameter d during the training (**by default:** is 3).

```
def train(self, verbose=False):
```

Description: This is the training method of the ARIMA class, which starts the model selection (p,q and d) and then estimating the model parameters for the training dataset declared in the constructor. Parameters:

1. ***verbose***: (Boolean) True if we want to see log information about the trained model (**by default:** is False). Output: Nothing (Only update the model information).

```
def forecast(self, input_data, steps, window=None):
```

Description: This is the forecast method of the ARIMA class, which predicts a number of future steps of the input signal. Parameters:

1. *input_data*: (Numpy Array with two dimensions) the input signal for the testing dataset, with one column and N rows.
2. *steps*: (Integer) the number of future points to predict. Output: (Numpy Array with two dimensions) the predicted signal for the testing dataset, with one column and "steps" rows.
3. ***window***: (Integer) the number of point used during the intial phase of the prediction, when the previous error are estimated (**by defult:** is None, equivalent to "all data").

Output: (Numpy Array with one dimension) the predicted signal for the testing dataset, with "steps" values.

WPF (./WPF.py):

```
def getWaveletByName(name):
```

Description: Returns a function with a SWT or SWPT based on the name. Parameters:

1. *name*: (String) Can be "hswt" for SWT or "hspwt" for SWPT.

Output: Returns a function with a SWT or SWPT based on the name (or None and prints an error message).

```
class WaveletPrediction:
```

Description: Combine a forecasting method with a preprocesing method with a similar to the other forecasting methods interface.

```
def __init__(self,Data,waveletFunction, components, algorithms):
```

Description: Constructor of the WaveletPrediction class, which sets the training data, a wavelet function for the preprocessing and the forecasting algorithm to combine. Parameters:

1. *Data*: (Numpy Array with one dimension) the input signal for the testing dataset, with N values.
2. *waveletFunction*: (a wavelet function returned by "getWaveletByName") wavelet function that to apply to the testing and training dataset.
3. *components*: (Integer) the parameter that will use the wavelet function.
4. *algorithms*: (Class) a class that implements WPFAlgorithm interface (explained below). is a forecasting class, usually with some predefined parameters.

```
def train(self):
```

Description: This is the training method of the WaveletPrediction class, which generate and train a model based on the "algorithms" parameter set in the contructor for each signal generated by the wavelet transform. Parameters : No parameters. Output: Nothing (Only update the model information).

```
 def forecast(self, input_data, steps):
```

Description: This is the forecast method of the WaveletPrediction class, which predicts a number of future steps of the input signal. Parameters:

1. *input_data*: (Numpy Array with one dimension) the input signal for the testing dataset, with N values.
2. *steps*: (Integer) the number of future points to predict.

Output: (Numpy Array with one dimension) the predicted signal for the testing dataset, with "steps" values.

```
class WPFAlgorithm:
```

Description: A class that we must to implement with some predefined forecasting algorithm to use the WaveletPrediction class.

```
def train(self,data):
```

Description: This is the train method of the WPFAlgorithm class, which we must redefine.
Parameters:

1. *data*: (Numpy Array with one dimension) the input signal for the training dataset, with N values.

```
def forecast(self, input_data, steps):
```

Description: This is the forecast method of the WPFAlgorithm class, which we must redefine.
Parameters:

1. *input_data* : (Numpy Array with one dimension) the input signal for the testing dataset, with N values.
2. *steps*: (Integer) the number of future points to predict.

Output: (Numpy Array with one dimension) the predicted signal for the testing dataset, with "steps" values.

Annex B : Documentation of UNADA package

Link to the code repository: https://gitlab.com/ontic-wp4/UNADA

UNADA is a tool which detects anomalies in PCAP files. It relies on subspace clustering and evidence accumulation techniques to identify anomalies and generate signatures to describe them.

The algorithm is described in the deliverable 4.2 of the ONTIC project which can be found on the ONTIC site.

The clustering step can be performed either with the DBSCAN algorithm (Density-Based Spatial Clustering of Applications with Noise) or the DBSCAN algorithm with an Rtree index or the GCA (Grid Clustering Algorithm) algorithm. These clustering algorithms exhibit the same performance in terms of detection but different time complexity. From medium to larger files, GCA is faster than DBSCAN with an Rtree index, which is also faster than DBSCAN.

Input dataset

UNADA accepts as input dataset a pcap file. We advise using a pcap file of 15 seconds for better results. Furthermore, UNADA may take time if the input file is very large and may then need a lot of RAM. UNADA looks for anomalous flows in this pcap file.

Content of the package

This package contains:

- a file `README.md`: it describes the package and how to use it.
- a directory `src`: it contains the sources of the program.
- a file `pom.xml`: it provides the necessary information to MAVEN to build the project.
- a file `ExampleInputFile.pcap`: an example of input file.
- a jar `UNADA.jar`: a jar file to launch UNADA.

Requirements

If you want to recompile the code source, you must install MAVEN. We recommend using Apache Maven 3.3.3. To execute UNADA, with the pcap file 'ExampleInputFile.pcap' we recommend a machine with at least 12GB of RAM to apply UNADA with DBSCAN and 8GB of RAM to apply UNADA with GCA.

You need to install the jNetPcap1.4 library

Arguments

UNADA takes 4 mandatory arguments plus 2 optional ones. The four mandatory arguments are:

- the path to the pcap file to analyze.

- the direction of the aggregation. You have to specify whether the aggregation is made at the IP source 'src' or at the IP destination 'dst'.
- the mask of the aggregation. You need to specify if it is '8', '16', '24', '32'.
- the clustering technique you want to apply. It can be dbsan then you specify 'dbscan', dbscan with an Rtree index then you specify 'rtreeDbscan', GCA then you specify 'gca'. The two optional arguments are:
- if you have chosen the clustering algorithm DBSCAN with or without Rtree, you need to specify a percentage to fix the radius of DBSCAN. The radius is then computed for each subspace as (percentage/100)*distance max between two points in this subspace. If you have chosen GCA, you need to specify a percentage to fix the intervals' length of GCA in each subspace. The interval length is computed for each subspace as (percentage/100)*d with d the distance max between two points in this subspace.
- if you have chosen the clustering algorithm DBSCAN with or without Rtree or GCA, you need to specify a percentage to fix the minimum number of points to form a cluster . The minimum number of points is then computed as (percentage/100)*tot with tot the total number of flows. If you don't provide the two optional arguments, some defaults ones are used.

How to run

To launch UNADA with GCA and the aggregation level at the IP source with the mask 32 and with no optional arguments, the command line is:

```
java -Djava.library.path=/pathToTheJnetPcapLibrary/jnetpcap -Xms7G -jar UNADA.jar /path
ToThePCAPFile/file.pcap src 32 gca
```

To launch UNADA with DBSCAN and the aggregation level at the IP source with the mask 16 and with the two optional arguments, the command line is:

```
java -Djava.library.path=/pathToTheJnetPcapLibrary/jnetpcap -Xms11G -jar UNADA.jar /pat
hToThePCAPFile/file.pcap src 16 dbscan 10 5
```

How to compile

To compile the code source, the command line is:

```
mvn compile
```

To create a package from the code source, the command line is:

```
mvn package
```

Output

It outputs on the standard output some information about UNADA's execution. It also creates an XML which lists the anomalous flows found in the PCAP file. For each anomalous flow it specifies its features, its score of dissimilarity and its signature.

Annex C : Documentation of PUNADA package

Link to the code repository: https://gitlab.com/ontic-wp4/PUNADA

PUNADA is a tool which detects network anomalies. It relies on subspace clustering and evidence accumulation techniques to identify anomalies. It distributes its computation over a cluster of server using Spark.

The algorithm is described in the deliverable 4.2 of the ONTIC project which can be found on the ONTIC site.

The clustering is performed with the DBSCAN algorithm (Density-Based Spatial Clustering of Applications with Noise).

Input dataset

UNADA accepts as input the path where are put text files like ExampleInputFile.txt (see package). Each file describes the flows extracted from network traffic. Each file defines the features name of each flow. The first element of the line is the aggregation key of the flow. Each line represents a flow; the first element is its flow key. The number of features must be inferior to 100. Each element of a flow is separated by a blank. The flows features must have been previously extracted from network traffic. We recommend extracting these features from 15 seconds network traffic to improve detection performance.

Content of the package

This package contains:

- a file `README.md`: it describes the package and how to use it.
- a directory `src`: it contains the sources of the programm
- a file `pom.xml`: it provides the necessary information to MAVEN to build the project.
- the `ExampleInputFile.pcap`: an example of input file.
- a jar `PUNADA-1.jar`: a jar file to launch UNADA.
-

Requirements

If you want to recompile the code source, you must install MAVEN. We recommend using Apache Maven 3.3.3. You need to install Spark 1.5.1 to run the .jar.

Arguments

PUNADA takes 1 mandatory argument plus 2 optional ones. The mandatory argument is:

- the path of the directory where are stored the files to process. The two optional arguments are:

- the DBSCAN's radius.
- an integer to fix the minimum number of points to form a cluster in DBSCAN. The minimum number of points is computed as (percentage/100)*tot with tot the total_number_of_flows. You need to provide the value of this percentage which must lie in [1, 99]. If you don't provide the two optional arguments, some defaults ones are used.

How to run

To launch PUNADA, the command line is:

```
spark-submit --master spark://hostname:7077 --class spark.PUNADA  --conf "spark.default
.parallelism=100" --executor-memory 26g PUNADA-1.jar /Path/To/The/File
```

To launch PUNADA with the two optional arguments, the command line is:

```
spark-submit --master spark://hostname:7077 --class spark.PUNADA --conf "spark.default.
parallelism=100"  --executor-memory 26g PUNADA-1.jar /PathToTheFiles 0.1 5
```

How to compile

To compile the code source, the command line is:

```
mvn compile
```

To create a package from the code source, the command line is:

```
mvn package
```

Output

It creates two directories res and processed. In the directory processed are put the input text files which have been processed. The results are stored in the res directory. For each input text file processed a text file of results is generated. This file lists the flows which have been detected as anomalies by PUNADA. Each line represents an anomaly and specifies its key and its features.

Annex D : Documentation of ORUNADA package

Link to the code repository: https://gitlab.com/ontic-wp4/ORUNADA

ORUNADA is a tool which detects anomalies in PCAP files in a continuous way. It relies on a sliding window and on grid subspace clustering and evidence accumulation techniques to identify in continuous anomalies and generate signatures to describe them.
The algorithm is described in the deliverable 4.2 of the ONTIC project which can be found on the ONTIC site.
The clustering step is performed with IGCA (Grid Clustering Algorithm) algorithm.

Input dataset

UNADA accepts as input a folder with pcap files. Furthermore, ORUNADA may take time (due to the extraction of the features which should be done in C to gain time) if the input files are very large and may then need a lot of RAM. UNADA looks for anomalous flows in this pcap file.

Content of the package

This package contains:

- a file `README.md`: it describes the package and how to use it.
- a directory `src`: it contains the sources of the program.
- a file `pom.xml`: it provides the necessary information to MAVEN to build the project.
- a file `ExampleDirectory`: an example of directory with pcap files to process.
- a jar `ORUNADA.jar`: a jar file to launch UNADA.
-

Requirements

If you want to recompile the code source, you must install MAVEN. We recommend using Apache Maven 3.3.3. To execute UNADA, with the pcap file 'ExampleInputFile.pcap' we recommend a machine with at least 12GB of RAM to apply UNADA with DBSCAN and 8GB of RAM to apply UNADA with GCA.
You need to install the jNetPcap1.4 library

Arguments

UNADA takes 3 mandatory arguments plus 4 optional ones. The four mandatory arguments are:

- the path to the pcap file to analyze.

- the direction of the aggregation. You have to specify whether the aggregation is made at the IP source 'src' or at the IP destination 'dst'.
- the mask of the aggregation. You need to specify if it is '8', '16', '24', '32'. The four optional arguments are:
- Time of a slot in seconds;
- Nb of micro-slots in a slot;
- for computing the size of the intervals for IGDCA. Each dimension has a different size of interval. This value is set as a percentage of the maximum distance between every pair of points for each dimension. To fix them, you need to specify this percentage and specify a value between 1 and 99.");
- for computing the minimum number of points to form a cluster in IGDCA. This number is set as a percentage of the whole number of points. To fix it, you need to specify this percentage and specify a value between 1 and 99."); If you don't provide the four optional arguments, some defaults ones are used.

How to run

To launch ORUNADA with the aggregation level at the IP source with the mask 32 and with no optional arguments, the command line is:

```
java -Djava.library.path=/pathToTheJnetPcapLibrary/jnetpcap -Xms7G -jar ORUNADA.jar /pathToThePCAPFolder/ src 32
```

To launch UNADA with DBSCAN and the aggregation level at the IP source with the mask 16 and with the two optional arguments, the command line is:

```
java -Djava.library.path=/pathToTheJnetPcapLibrary/jnetpcap -Xms11G -jar ORUNADA.jar /pathToThePCAPFile/file.pcap src 16 15 30 5 10
```

How to compile

To compile the code source, the command line is:

```
mvn compile
```

To create a package from the code source, the command line is:

```
mvn package
```

Output

It outputs on the standard output some information about ORUNADA's execution. It also creates an XML file for each micro-slot processed (apart for the n first micro-slots, n equals to the number of micro-slots in a window). This XML lists the anomalous flows found in the PCAP file at the end of each micro-slot considering the packets contained in the current window. For each anomalous flow it specifies its features, its score of dissimilarity and its signature.