Getting started with the Data Analysis as a Service (DAaaS)

Contact person: José Camacho
Last modification: 22/05/2023

1 Introduction

In this document the architecture of the **Data Analysis as a Service** (**DAaaS**) is introduced and an example of data workflow is explained in detail.

1.1 Server Architecture

This section briefly describes the ***Data Analysis as a Service*** architecture. This information is detailed in the Deliverable 7 (in Spanish) of project ANIMaLICoS [1], accessible on the project website.

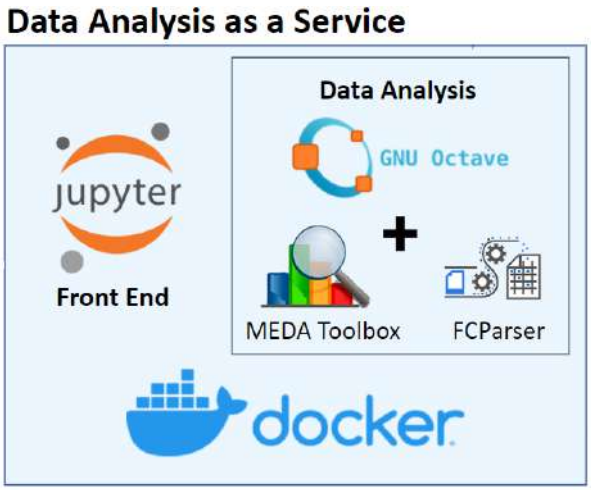


Figure 1: Diagram of the Data Analysis as a Service.

First, the DAaaS basis is a container manager, ***Docker***. The DAaaS itself is a ***Docker*** container. So far the container is not available onlie, but it can be provided on-request. The container provides an installation of ***JupyterHub***. In the *notebook*, we have installed, on the one hand, the ***FCParser*** [2], a library that allows a general and highly configurable parsing of data from different sources, and on the other hand, ***Octave*** and ***MEDA Toolbox***(*Multivariate Exploratory Data Analysis Toolbox*)[3], which is a set of multivariate analysis tools for the exploration of datasets.

2 Example of DAaaS usage

2.1 Server access

First of all, you have to access the server at https://jcp.ugr.es:8000/hub/login. If you don't have an account yet, you have to sign on with a new account and contact the administrator (J. Cama-

cho) providing your username, real name and organization.

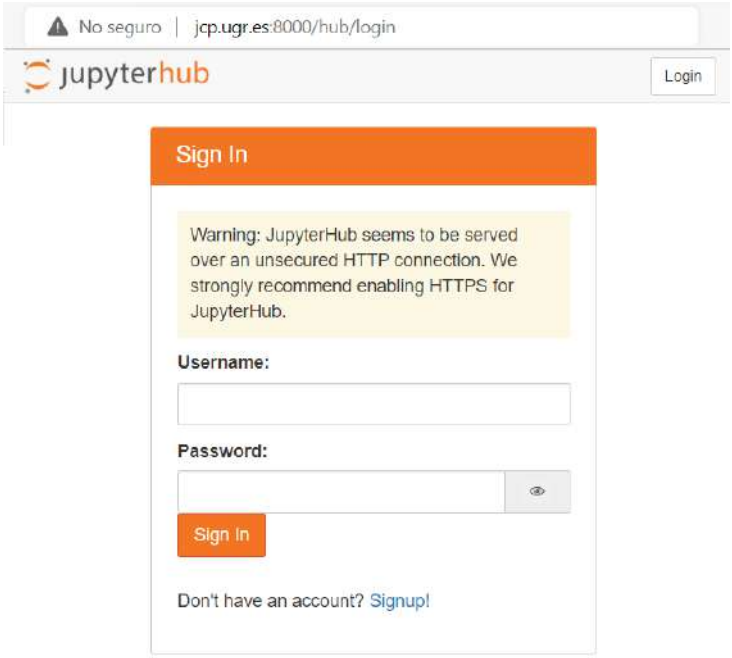


Figure 2: Data Analysis as a Service access.

2.2 Dataset

Before starting with the example, we need to know what data is going to be analyzed and where it is located. The dataset that is going to be used is UGR'16 Dataset [4], which is built with real traffic and up-to-date attacks. These data come from several *netflow v9 collectors* strategically located in the network of a Spanish *ISP*. It is composed of two differentiated sets of data that are previously split in weeks : *calibration* (from March to June of 2016) and *test* (from July to August of 2016). For more information, refer to the reference above or click on the link. The actual data used by the server has been updated to a new nfdump version.

For this example, we are going to use a small portion of the data, since the total trace includes close to **6 months of traffic**. These files are very large and we would need a lot of computing resources to process and analyze them. We have selected a time window of **one hour** in which there is interesting activity. However, starting from this example, the interested users may carry out a wider analysis.

The period that has been selected is the day **13/04/2016** from **3am** to **4am**. The data of the capture is stored in
april_week3_nfcapd, that is the collected binary netflow file of the week. In order to process the data we must extract **one hour** of that day and pass it to '*csv*' format. To do this we have used *nfdump* with the following command:

nfdump -r nfcapd_week3_capd -t 2016/04/13.03:00:00-2016/04/13.04:00:00 -q -N -o 'csv' > nf_1304.csv

2.3 Jupyter Notebook interface

We also need to know the enviroment that we are going to use for the example. This section will show where we are going to use the tools and the main files we need to know.

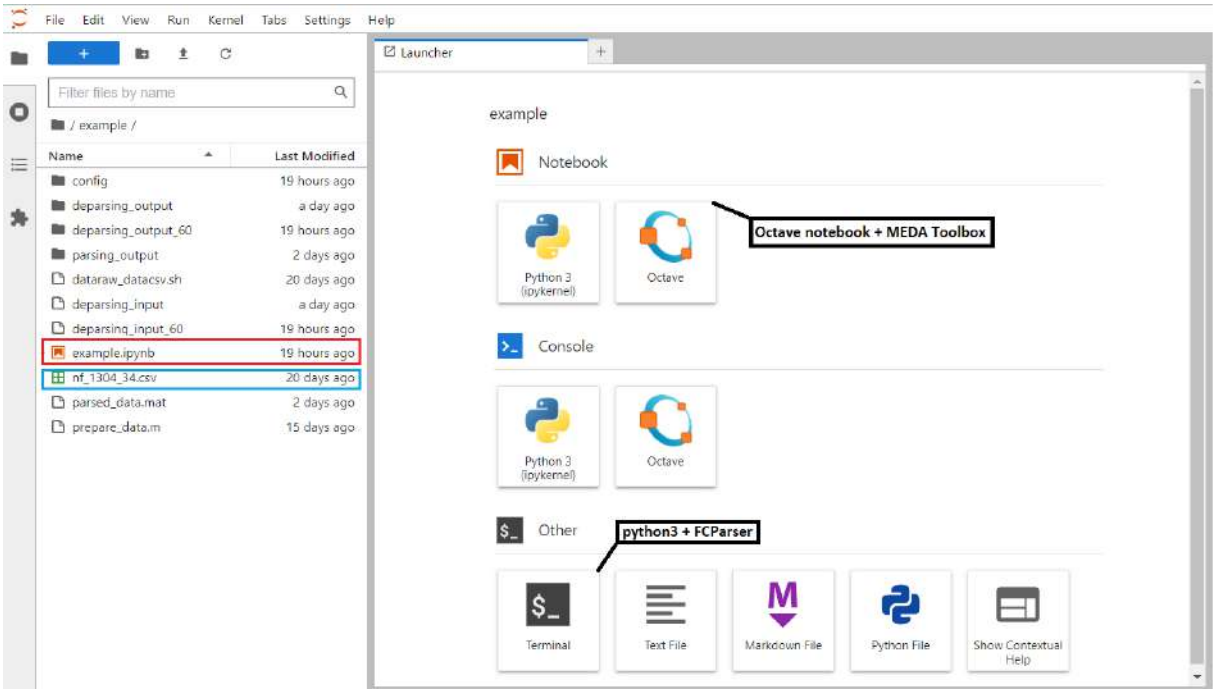


Figure 3: Jupyter Notebook interface.

- **nf_1304_34.csv** : capture of traffic used.
- **Python3 + FCParser** : parsing data.
- **Octave notebook + MEDA Toolbox**: data analysis.
- **example.ipynb** : notebook with commands for the example done.

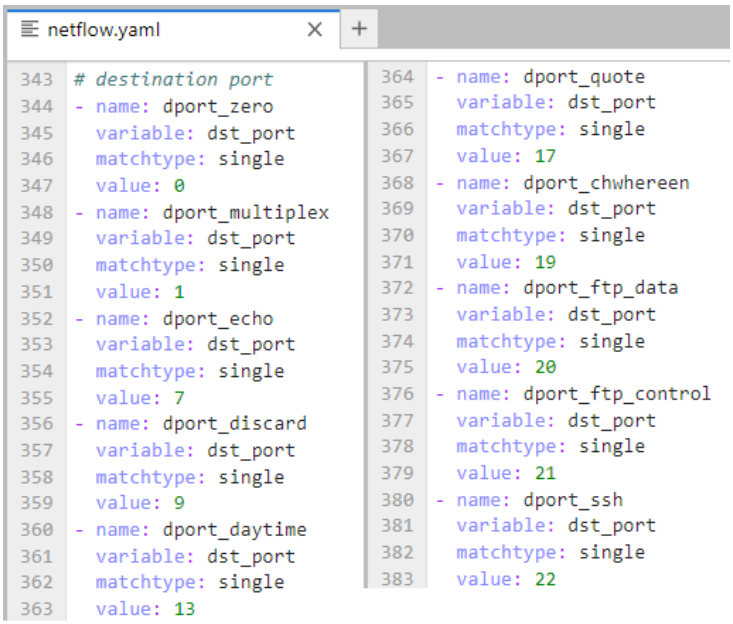
2.4 Parsing data with FCParser

The first step is to parse the data so that we will extract features from a data set in order to process and interpret them in a more simplified way. All this, starting from a set of decisions that the analyst needs to make. In the example, we use predefined setting for most of this decisions.

For this purpose, as indicated in the previous section, we will make use of the **FCParser tool**, which is a parser for data streams composed of various (structured and unstructured) sources. This tool converts variables into feature observations to facilitate further analysis, using the *Feature as a Counter* (*FaaC*) approach [2]. Then it aggregates the observations according to specific criteria and fuses the observations from different data sources. For more information and installation help, refer to github.com/josecamachop/FCParser.

First we start by defining the features that we want to extract (count) from the traffic capture. For this purpose we have used the ***netflow.yaml*** file (located in the config folder) in which 142 features have been defined, such as source/destination ports or IPs, which will be useful for

traffic analysis. For example, some features of destination port:

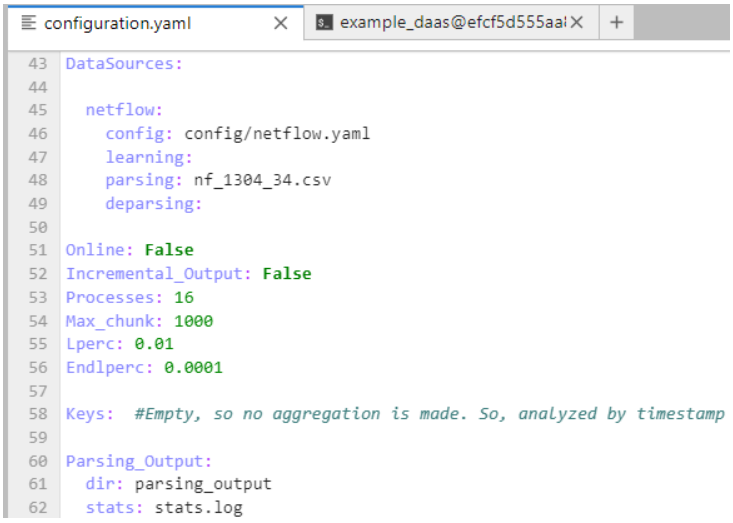


```yaml
# destination port
- name: dport_zero
  variable: dst_port
  matchtype: single
  value: 0
- name: dport_multiplex
  variable: dst_port
  matchtype: single
  value: 1
- name: dport_echo
  variable: dst_port
  matchtype: single
  value: 7
- name: dport_discard
  variable: dst_port
  matchtype: single
  value: 9
- name: dport_daytime
  variable: dst_port
  matchtype: single
  value: 13
- name: dport_quote
  variable: dst_port
  matchtype: single
  value: 17
- name: dport_chwhereen
  variable: dst_port
  matchtype: single
  value: 19
- name: dport_ftp_data
  variable: dst_port
  matchtype: single
  value: 20
- name: dport_ftp_control
  variable: dst_port
  matchtype: single
  value: 21
- name: dport_ssh
  variable: dst_port
  matchtype: single
  value: 22
```

Figure 4: Part of netflow.yaml

Then we define the configuration file (***configuration.yaml***), where we have to indicate apart from some processing features (CPUs, memory), the data sources and the directory where we want to save the files of the parsed capture.

In this case, we have used the *netflow* data source that we have defined before and the traffic capture *nf_1304_34.csv*. In processes, we have selected the maximum number of CPUs available. Finally, the *parsing_output* folder has been selected as the output where the files will be saved.



```yaml
DataSources:

  netflow:
    config: config/netflow.yaml
    learning:
    parsing: nf_1304_34.csv
    deparsing:

Online: False
Incremental_Output: False
Processes: 16
Max_chunk: 1000
Lperc: 0.01
Endlperc: 0.0001

Keys:  #Empty, so no aggregation is made. So, analyzed by timestamp

Parsing_Output:
  dir: parsing_output
  stats: stats.log
```

Figure 5: Part of configuration.yaml

Finally we run it on the terminal (click on the terminal shown in Figure 3):
python3 /FCParser/bin/fcparser.py config/configuration.yaml

Warning: For this 1 hour capture the *FCParser* takes about 10 minutes to process and get the output files. The folder including the outputs is already provided. The user can do both : run it or otherwise use the provided output folder directly.

```
example_daas@efcf5d555aa8:~/example$ python3 /FCParser/bin/fcparser.py config/configuration.yaml
LOADING GENERAL CONFIGURATION FILE...
* Offline mode (multiprocess)
* Incremental_output: False
* Cores: 16
* Max_chunk: 1000 MB
* Time sampling window: 1 minutes
** Creating output directory parsing_output/
 ** Defining default weights file: 'weights.dat'
GENERAL CONFIGURATION FILE... OK
LOADING DATA SOURCES CONFIGURATION FILES...
* File: config/netflow.yaml
--------------------------------------------------------------------------------
Data Sources:
 * netflow           15 variables   142 features
 TOTAL 142 features
Output:
  Directory: parsing_output/
  Stats file: stats.log
  Weights file: weights.dat
--------------------------------------------------------------------------------
Note: Malformed logs or inaccurate data source configuration files will result in None variables which will not be counted in any feature.
Run program in debug mode with -d option to check how the records are parsed.
--------------------------------------------------------------------------------
Elapsed: 1 secs
netflow  #1 / 1  nf_1304_34
Elapsed: 8 mins, 15 secs
```

Figure 6: Running FCParser.

If we go to the specified folder (*parsing_ouput*), we will see that a file has been generated for each minute of the capture (61 files in this case). We suggest to delete the last file as *nfdump output* includes some spurious files with flows that slightly delayed, making the *FCParser* create an almost empty final interval. It will alter our analysis.

***NOTE**: In case of using another traffic capture you would only have to change the configuration.yaml indicating the data source with the capture and a new output folder in order to mix it and run it again.*

2.5 Data analysis with Octave and MEDA Toolbox

In this section, we explain the example of data analysis with ***PCA***(*Principal Component Analysis*) given on the DAaaS. Once we have obtained the output folder of our data we can start with its analysis. First, the user has to open the example given (click on *example.ipynb* in Figure 3). In order to use the *MEDA-Toolbox*, we need to add the directory of the ***MEDA Toolbox*** in *Octave*:

```
% Make sure we have the MEDA Toolbox installed, so we can add it to the path in
    Octave:
addpath('/MEDA-Toolbox');
addpath('/MEDA-Toolbox/BigData');
pkg load statistics
```

2.5.1 Data preparation and initialization

The next step is prepare input data from octave. To do this we will make use of the *prepare_data.m* function :

```
% First, we have to prepare data using the function 'prepare_data.m'
prepare_data('parsing_output');
```

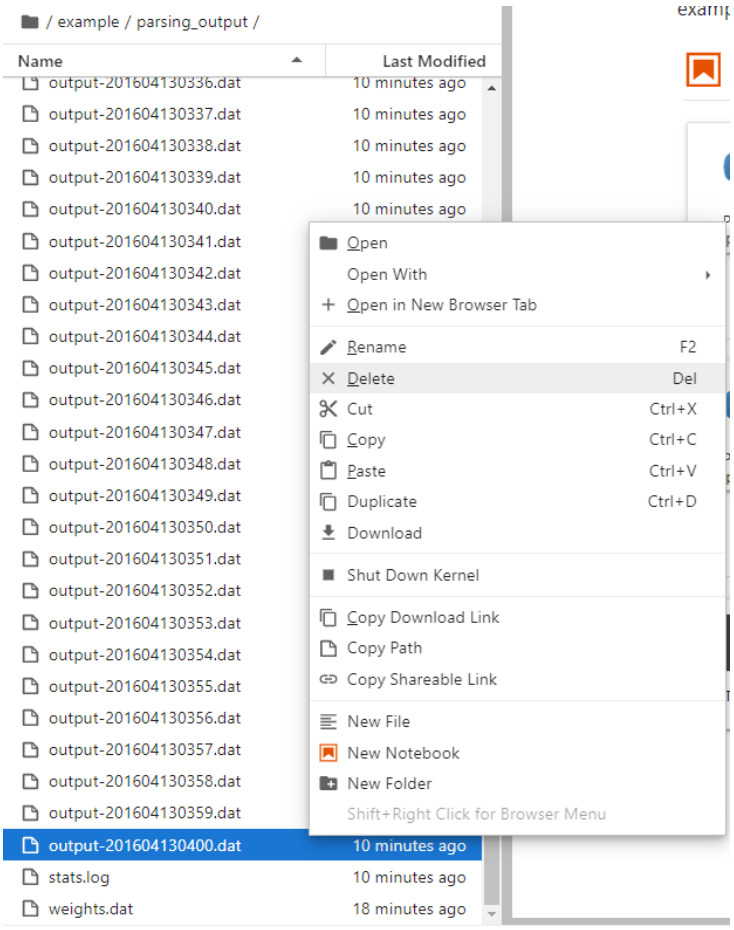


Figure 7: Deleting last output of the folder.

```
% coded by: Jose Camacho - josecamacho@ugr.es
% last modification: 30/Jun/20.

function prepare_data(folder)

fname = dir([folder '/out*']);

obs_l = cell(1,length(fname));
b = importdata([folder '/weights.dat']);
weight= b.data;
var_l = strsplit(b.textdata{:},',');

x = zeros(length(fname),length(var_l));

for i=1:length(fname)

    timestamp = strsplit(fname(i).name,'output-');
    timestamp = strsplit(timestamp{2},'.dat');
    obs_l{i} = timestamp{1};
    x(i,:) = load(strcat(folder,'/',fname(i).name));
end

save ('parsed_data.mat','x','obs_l','var_l', 'weight')
```

This function has as input the folder containing the output files of the *FCParser* (*parsing_data*) and combines all the files *'output_xxxx'* of the 60 minutes, obtaining 4 data elements in *'parsed_data.m':*

- **x** : matrix with the values of the observations for each variable (60 observations x 142 features).
- **obs_l** : vector that identifies the labels of the observations (60 observations).
- **var_l** : vector that identifies the labels of the variables (142 variables).
- **weight** : vector with the value of the weight of each variable (142 variables).

Then, the file we just obtained is loaded (*'parser_data.mat'*) with the 4 data elements (*x, obs_l, var_l and weight*).

```
% Load the file which includes x, obs_l, var_l & weight
load('parsed_data.mat'); %scalar structure

% Inicialization of some parameters
prep_x = 2; % autoscaling (the nature of the variables is different)
max_PCs = 10; % maximum number of PCs to take into account
```

In addition, other parameters to be used during the analysis are initialized:

- **prep_x** : type of preprocessing that is going to be used. We select number 2 as it is autoscaling (see explanation using *'help preprocess2D'* in *Octave*). Alternatively, one may also use 0 (no preprocessing) and 1 (mean-centering).
- **max_PCs** : maximum numbers of Principal Components to take into account.

2.5.2 Selection of the number of PCs (Principal Components)

The first step for the ***PCA*** analysis is to select the appropiate number of *PCs* to make the analysis.

```
%% Data Analysis - PCA (Principal Component Analysis)
% 1. Selection of the Principal Components (PC)
var_pca(x,1:max_PCs,prep_x); % We select 2 PCs despite 80% of residual variance
% For more detail, we should study more PCs
```

To do so, the function *'var_pca(x,pcs,prep)'* of the *MEDA Toolbox* is used, which has as input : **x** (*matrix with observations and features*), **pcs** (*from 1 to the maximum number of pcs*) and **prep** (*type of preprocessing*).

In this case we select **2 PCs**. Typically, this selection is complex, but a general rule of thumb is to select the *"knee"* in the curve. For a more accurate analysis of the data, it should be redone with more components or take into account the residuals, as we are leaving a lot of information in it.

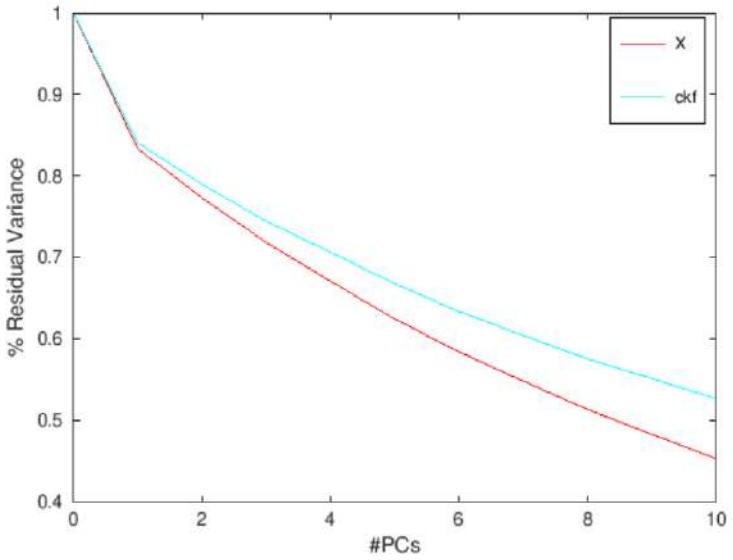


Figure 8: Variability captured in terms of the number of PCs.

2.5.3 Observations distribution (Scores)

Then, the obtained scores are plotted to show the distribution of the observations, so that we can identify the outliers. To do this we use *'scores_pca(x,pcs,test,prep,opt,label,classes)'*, where:

- **x** : matrix with observations and features (*parsed_data.x*).
- **pcs** : principal components taken into account (*in this case, 2 pcs*).
- **test** : data set with the observations to be compared (*to do testing of a dataset, we don't want this now*).
- **prep** : type of preprocessing (*2 for auto-scaling*).
- **opt** : options for data plotting: binary code (*type help scores_pca* for more information).
- **label** : name of the observations (*we don't add them because we would not be able to see the dots well. Add 'parsed_data.obs_l' to see them*).
- **classes** : groups for different visualization (*used to compare different groups, not needed here*).

```
% Step 2: observations distribution and relationships, outliers detection

scores_pca(x,1:2,[],prep_x,1);
% we can see the distribution where almost all of them are similar
% obs 26 and 60 are outliers, should be studied with more detail
```

In the scores we can see that almost all the observations are similar and are around the axis. However, there are two observations that are far from this trend (**outliers 26 and 60**). To determine why they are different from the rest we must look at the **loadings** corresponding to the *distribution of the variables* and study both in detail.

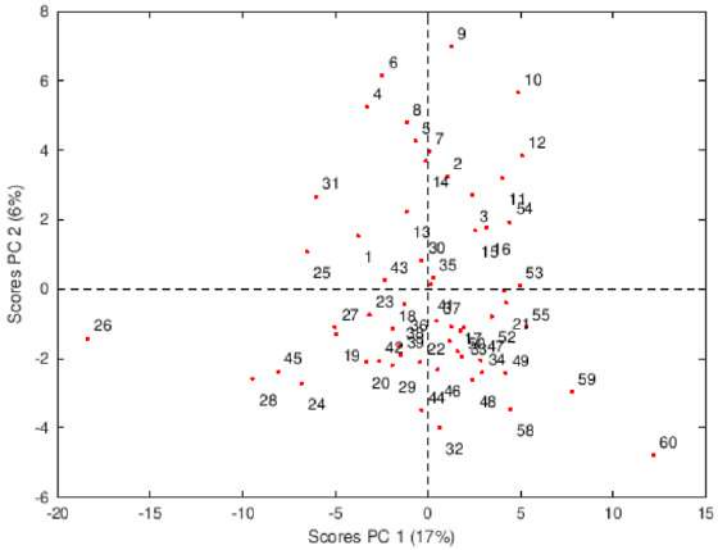


Figure 9: Scores (distribution of observations).

2.5.4 Loadings distribution (Loading)

As mentioned above, we now need to look at the distribution of the variables to determine the behavior of the outliers in the scores. For this we will use *'loadings_pca(x,pcs,prep,opt,label,classes)'* which has the same inputs that we have used in the function *'scores_pca()'*.

```
%% Step 3: variables distribution and relationships, selection of variables

loadings_pca(x,1:2,prep_x,1);
% too many variables to identify patterns, but can aproximate variables affecting
% look at bottom left & right. Should be studied with more detail
```

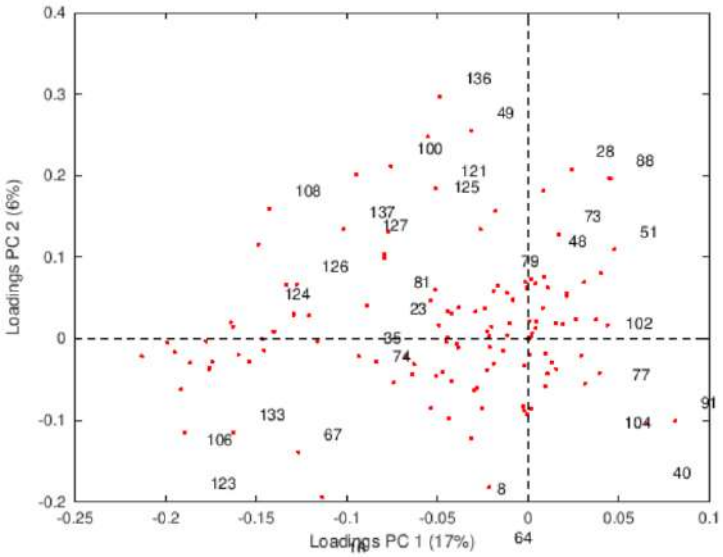


Figure 10: Loadings (distribution of variables).

There are too many variables to determine a pattern, but we can more or less see the distribution where as an intuition we can assume that the variables that are bottom left will be those that

affect **outlier 26** and those that are bottom right will be for the case of **outlier 60**.
To check this, we need to do a more detailed study to find out the variables that are affecting these outliers.

2.5.5 Outliers investigation with oMEDA

For a more detailed study, we will compare the values of the variables by placing positive and negative weights according to their distribution in the observations. To do this, we use *'omeda_pca(x,pcs,test,dummy,prep,opt,label,classes)'* where all the inputs are the same as the functions used before 'scores_pca()' and 'loadings_pca()', except *dummy*.

Dummy is a variable containing weights for the observations to be compared, and 0 for the rest of the observations that are not to be taken into account. So, the weight has been set to **-1** for the group of centered points, **0** for those too close or too far away and **1** for the weight of the outlier.

```
%% Step 4: investigate differences with outliers, oMEDA and line plots of outliers

% outlier 26
dummy = -ones(60,1);
dummy([45,28,24,25,31,59,60])= 0;
dummy(26)=1;
omeda_pca(x,1:2,x,dummy,prep_x,111,var_l);

% outlier 60
dummy = -ones(60,1);
dummy([45,28,24,25,31,59,26])= 0;
dummy(60)= 1;
omeda_pca(x,1:2,x,dummy,prep_x,111,var_l);
```

- **Outlier 26.**

In these graphs we have to pay attention to the peaks, because if they are very high they have an unusual value and different behavior to the group.

We look at the identifying numbers of the peaks and save them in a vector with the names of the most significant variables. We also save the timestamp where we have found the anomaly.

```
%% Step 5.1: identify variables affecting outliers
% look at the peaks (positives in this case)
% outlier 26
features = [var_l(107) var_l(122) var_l(110) var_l(45) var_l(35)]

timestamps = {'2016-04-13 03:25:00'}

% save it to use fcdeparsing
save ('deparsing_input','features','timestamps')
```

In this case, the variables that make the outlier different from the rest are : **'dport_register', 'srctos_other', 'protocol_udp', 'sport_oracle' and 'sport_smtp_ssl'** and the timestamp is **'2016-04-13 03:25:00'**. We should look at what they mean in the *netflow* file and investigate if it is relationed with any type of attack.

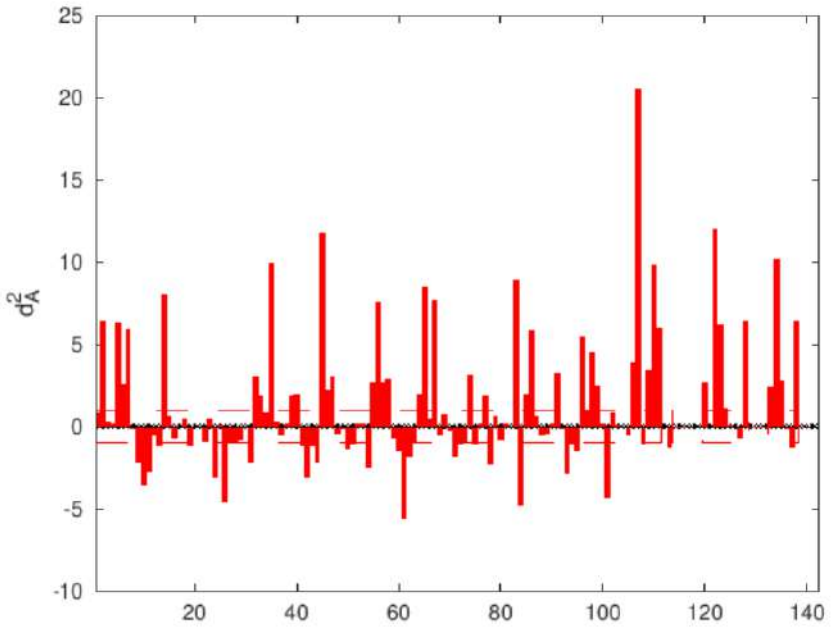


Figure 11: oMEDA for outlier 26.

- **Outlier 60.**

In this case we also have negative peaks so we will also look at them to see if they are related to the positive ones.

We do the same process as we have done with the previous outlier. We identify the peaks and save the positive ones and the timestamp, in order to study them later with the *FCParser*.

```
%% Step 5.2: identify variables affecting outliers
% outlier 60
features = [var_l(65) var_l(14) var_l(67) var_l(107)var_l(83)]

min_60 = [var_l(61) var_l(84) var_l(101) var_l(10) var_l(71)]

timestamps = {'2016-04-13 03:59:00'}

% save it to use fcdeparsing
save ('deparsing_input_60','features','timestamps')
```

In this case, the variables that make the outlier different from the rest are :

- **Positive peaks : 'dport_ftp_data', 'sport_ftp_data', 'dport_ssh', 'dport_register' and 'dport_https'.**
- **Negative peaks : 'dport_discard', 'dport_mds', 'dport_metasploit', 'sport_discard' and 'dport_bootp'.**

The timestamp is **'2016-04-13 03:59:00'**. We should also look at what they mean in the *netflow* file and investigate if they are relationed between them and if it is relationed with any type of attack too.

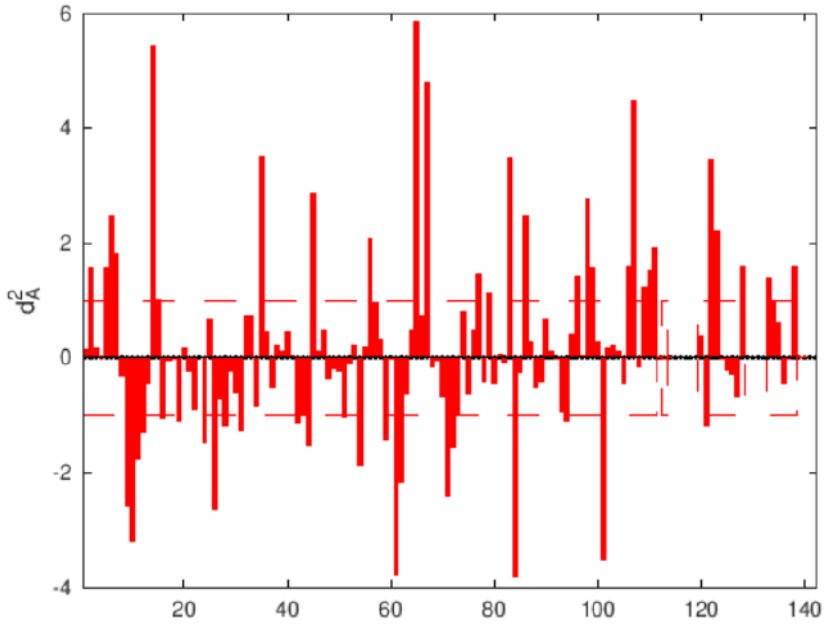


Figure 12: oMEDA for outlier 60.

2.6 Deparsing data with FCParser

Finally, we are going to identify the features in the capture, so that we can study in detail what is happening, whether it is a bug or a real attack. For this we are going to make use of the **FCParser** tool, in this case the *deparsing* one.

The way and syntax of executing it is similar to the one used in the section **2.4**. The command is ***python3 bin/fcdeparser.py example/config/configuration.yaml example/deparsing_input*** , where we have to edit the files *'configuration.yaml'* and *'deparsing_input*. We will have one for each outlier (*'deparsing_input'* and *'deparsing_input_60'*).

deparsing_input | deparsing_input_60

```
features:
{
  [1,1] =  dport_register
  [1,2] =  srctos_other
  [1,3] =  protocol_udp
  [1,4] =  sport_oracle
  [1,5] =  sport_smtp_ssl
}

timestamps:
{
  [1,1] = 2016-04-13 03:25:00
}
```

Figure 13: Deparsing input for outlier 26

We add to the configuration file, the deparsing capture and the directory where we are going to save the output.

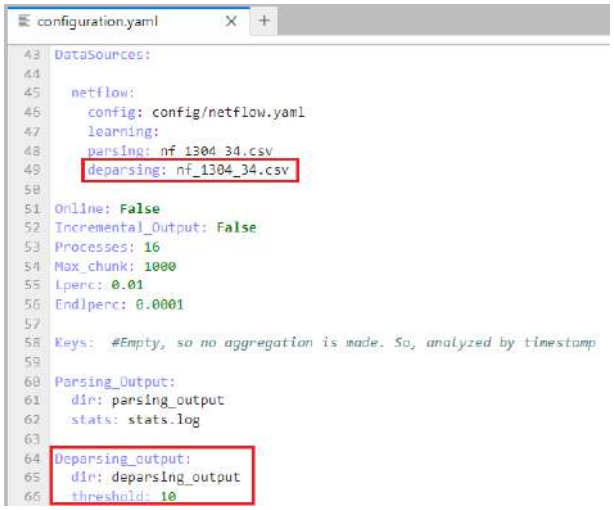


Figure 14: Configuration for deparsing

We go to the terminal and execute both of them, changing the deparsing directory:

```
example_daas@efcf5d555aa8:~/example$ python3 /FCParser/bin/fcdeparser.py config/configuration.yaml deparsing_input
* Threshold: 10 log entries per data source
* Time sampling window: 1 minutes
** Creating output directory deparsing_output/
 ** Defining default log file: 'stats.log'
GENERAL CONFIGURATION FILE... OK
LOADING DATA SOURCES CONFIGURATION FILES...
* File: config/netflow.yaml
Loading FCdeparser... Run program in debug mode with -d option in order to check the selection criteria in more detail
------------------------------ FCDEPARSER ------------------------------
* Loaded Deparsing input file.
- Features to search: ['dport_register', 'srctos_other', 'protocol_udp', 'sport_oracle', 'sport_smtp_ssl']
------------------------------------------------------------------------
Loading 'netflow' data source...
Number of logs with 5 matched features: 0
Number of logs with 4 matched features: 0
Number of logs with 3 matched features: 28369
Number of logs with 2 matched features: 46912
Number of logs with 1 matched features: 41098
Total number of logs in file: 5537039
Considering the feature counters and a threshold of 10 log entries, we will extract logs with >=3 matched features
Note that the output will be generated in different files according to their number of features
Elapsed: 2 mins, 17 secs
------------------------------------------------------------------------
Search finished:
Elapsed: 2 mins, 17 secs
 Structured logs found:  28369 out of 5537039
 Unstructured logs found: 0 out of 0
------------------------------------------------------------------------
```

Figure 15: Deparsing for outlier 26

As we can see, for the outlier 26, we have found out many logs that match the features we have selected for the input. It has saved the logs that matches 3 features. If we go to the specified folder, we can see them:

Then, we do the same with the outlier 60, obtaining in this case that the features aren't matched. We can also see them in the output directory.

Finally, we should study this logs deeply, in order to confirm that there have been real attacks or it was just a fail in the network.

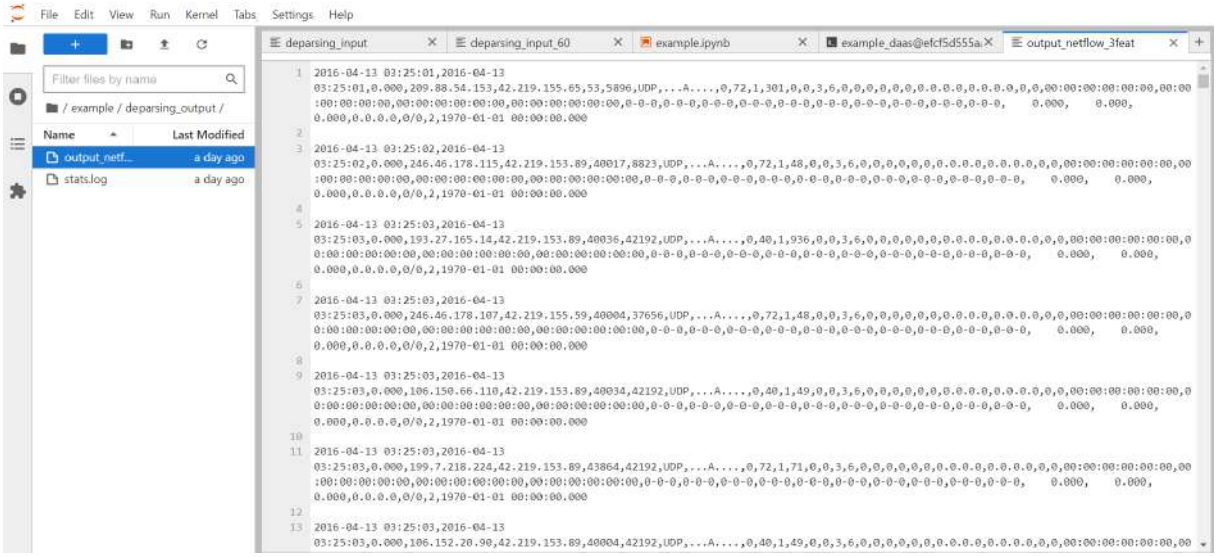


Figure 16: Output file with 3 matched features for outlier 26

```
example_daas@efcf5d555aa8:~/example$ python3 /FCParser/bin/fcdeparser.py config/configuration.yaml deparsing_input_60
* Threshold: 10 log entries per data source
* Time sampling window: 1 minutes
** Creating output directory deparsing_output_60/
 ** Defining default log file: 'stats.log'
GENERAL CONFIGURATION FILE... OK
LOADING DATA SOURCES CONFIGURATION FILES...
* File: config/netflow.yaml
Loading FCdeparser... Run program in debug mode with -d option in order to check the selection criteria in more detai

-------------------------------- FCDEPARSER --------------------------------
* Loaded Deparsing input file.
- Features to search: ['dport_ftp_data', 'sport_ftp_data', 'dport_ssh', 'dport_register', 'dport_https']
----------------------------------------------------------------------------

Loading 'netflow' data source...
Number of logs with 5 matched features: 0
Number of logs with 4 matched features: 0
Number of logs with 3 matched features: 0
Number of logs with 2 matched features: 0
Number of logs with 1 matched features: 29674
Total number of logs in file: 5537039
Considering the feature counters and a threshold of 10 log entries, we will extract logs with >=1 matched features
Note that the output will be generated in different files according to their number of features
Elapsed: 1 mins, 43 secs
----------------------------------------------------------------------------
```

Figure 17: Deparsing for outlier 60

References

[1] ANIMaLICoS. Advanced networkmetrics: Interpretable machine learning for intelligent communication systems. https://www.codas.ugr.es/animalicos/en.

[2] Feature as a counter parser for networkmetrics. Available online: https://github.com/josecamachop/FCParser.

[3] Multivariate exploratory data analysis (meda) toolbox. Available online: https://github.com/josecamachop/MEDA-Toolbox.

[4] Gabriel Maciá Fernández, José Camacho, Roberto Magán-Carrión, Pedro García-Teodoro, Roberto Theron, Ugr'16: a new dataset for the evaluation of cyclostationarity-based network IDSs, In Computers Security, 2017.