Bots detection by Conformal Clustering

Giovanni Cherubin

Submitted for the Degree of Master of Science in

Machine Learning

Department of Computer Science
Royal Holloway University of London
Egham, Surrey TW20 0EX, UK

August 28, 2014

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count:

Student Name:

Date of Submission:

Signature:

Abstract

Botnets, networks of infected machines controlled by an attacker, are a widespread and dangerous threat over the Internet, since they are used to perform most of the malicious activities such as distributed denial of service attacks (DDoS), phishing or spamming. This project aims at detecting botnet infected machines (*bots*) by analysing high level network traffic features (NetFlow data). Specifically, our main goal in this scenario is to detect previously known botnets, and to cluster together similar kinds of them. We approach the problem by using Conformal Clustering, a clustering method based on Conformal Prediction, where significance level is used to trim the hierarchy of clusters. This report introduces a few novel aspects to previous research, such as a substantial and motivated increase of the feature set, a feature evaluation algorithm to apply before clustering (*SIM-DIFF*), an algorithm for periodicity detection by using *Partial Autocorrelation Function* (PACF) instead of *Discrete Fourier Transform* (DFT), and the use of *Purity* as an accuracy criterion for Conformal Clustering. The use of NetFlow informations makes our approach reliable even when the traffic within the botnet is encrypted. This report also documents the creation of a *Python* package as an implementation of our work. This code has a structure which can be easily used and extended for future research.

Contents

Acknowledgement

I put many efforts and a lot of passion in this project. However, all this work has been helped and made possible by a few people I would like to thank here.

I would like to offer a special thanks to Prof. Alexander Gammerman for his expert, wise and friendly supervision. He helped me a lot during this project, and his passion for Machine Learning made me understand and interest in many new topics.

A big gratitude goes to Dr. Ilia Nouretdinov, for patiently answering in great detail all my questions, and for sharing with me his great knowledge and previous experience on this subject. I sincerely appreciated all this.

I would also like to thank Prof. Lorenzo Cavallaro, Dr. Zhi Wang and Dr. Davide Papini, for their great support on technical questions about the Information Security aspects of this project. They are currently working on this, and their knowledge was very important for approaching the topic.

I want to give a big thank to my good friend Mr. Lorenzo Grespan, his help and encouragement have made me walk a lot in these years, and he spent some time to discuss with me about this report.

First for importance, I would like to thank my parents, my brother and Elisa. All the efforts I put in what I do would be useless without the support and the love you give me every day.

1 Introduction

Network Intrusion Detection (NIDS[1]) is the use of Machine Learning techniques to detect attacks to computer networks. The general naive idea is to make a model of networks communications in "normal" or "under attack" conditions. These models should be then used to detect and report possible attack behaviours in the network. However, respect to other fields, the use of Machine Learning for network attacks detection must be cautious. In fact, as suggested by Sommer et al. in [15], it can be very hard to define what is "normal behaviour" in network communications, since legitimate traffic can assume very different behaviours from a statistical point of view. For this reason it is currently considered a good approach to create a model of known attack scenarios and create a system to detect them, rather then trying to spot anomalies within normal traffic flow.

This project takes part of the wide area of NIDS. We aim at detecting if a computer inside a network is part of a known botnet by looking at high level network traffic features. In order to achieve this, we proceed as others [16] did, by clustering informations extracted from NetFlow data. Our claim is that if we produce homogeneous clusters, where botnets of the same kind are clustered together, we are also able to detect them in a real world setting. As a consequence of this, it may be possible that our system is able to detect some new botnet threat, but this is not guaranteed by the assumption we work under. Section 6 presents and verifies our system on a *test* set. The use of NetFlow data makes our system reliable also when the data sent within the botnet is encrypted.

This section introduces the general structure of a botnet, the data we were provided, and gives an overview of the system we suggest. This section is also useful because it defines some conventions used in this report.

1.1 Definition of the problem: what a botnet is

We here describe from high level what are botnets, what kind of network traffic they produce, and where our observer sits when collecting data for our system. A reader interested in a more precise description of our problem can look at *A survey of botnet technology and defenses* from Bailey [2].

A *botnet* is a network of computers (*bots*) infected by a malware, which are controlled by an attacker, the *botmaster*. The idea behind botnets is that an attacker infects[2] many computers on the Internet, installs on them

[1] More properly *Network Intrusion Detection System.*

[2] There are many different ways an attacker can obtain the control on a computer on

a software to control all of them at once, and uses them to carry on malicious activities such as sending spam emails, performing Distributed Denials of Service attacks (DDos) or stealing user informations. From a general point of view the attacker, to control the botnet, uses a *Command & Control* server (C&C). This server is able to send commands to the bots (e.g.: "tell me what's in your user's browser cache" or "launch a denial of service attack to $ip address"), and to retrieve informations from them. Figure 1 represents at high level a botnet structure. Red elements represent the bots (infected computers), while the black computer represents the C&C. Dashed lines represent virtual communication circuits between C&C and the bots, passing through the Internet (blue cloud). In real cases the number of bots can reach many millions [17].

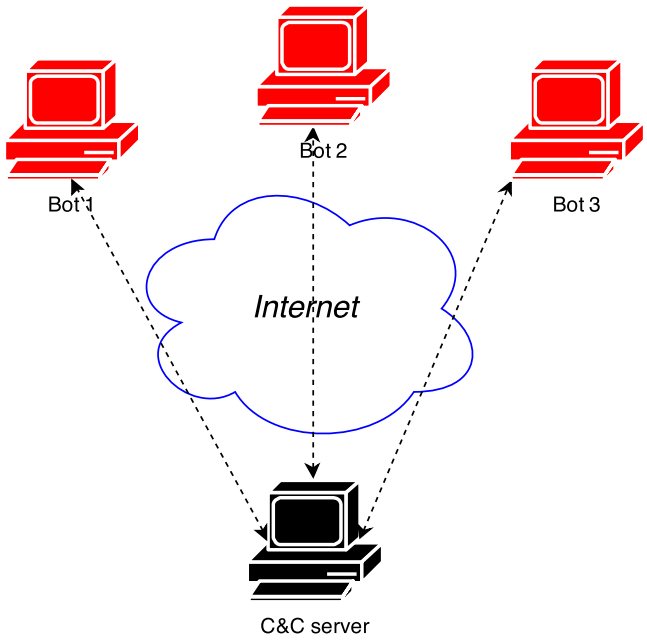

Figure 1: Botnet high level structure

Many kinds of botnets exist, and some of them can be characterized by the use of different protocols. The bots we analyse in our data were: *HTTP*,

the Internet. We do not discuss them here because this is irrelevant for our detection system.

IRC, and *p2p* based. The third category has a different structure from the one shown in Figure 1, but this can be ignored for the purpose of this report.

We collect data from an observer sitting between an infected computer (*bot*) and the Internet, as shown in Figure 2. This is done by running bots software on a honeypot: a protected environment in which we can safely execute malicious software. From our point of view, the honeypot can be thought as an infected computer within a network we control. Honeypots are not presented in this report, but a very good reference for this subject is a book from Joshi and Sardana [9].

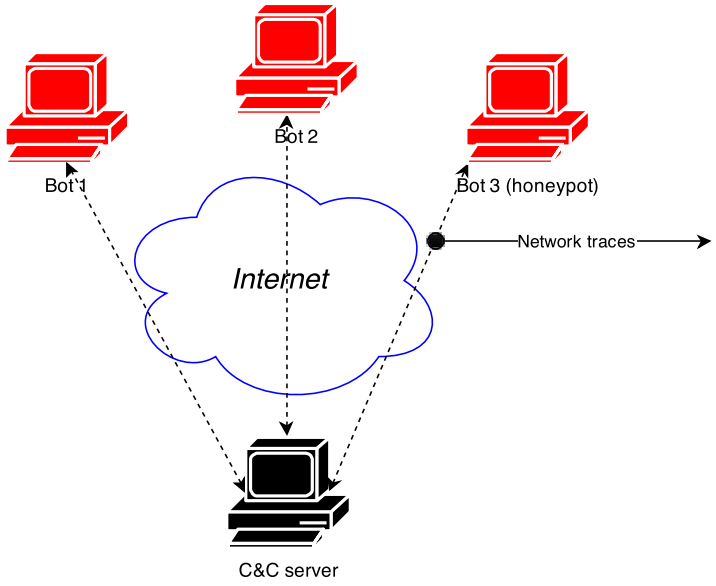

Figure 2: Where data is collected from

Our observer, the black circle on the virtual communication circuit between C&C and the honeypot, is able to see network traffic going from the bot to the C&C, and vice-versa from the C&C to the bot. This traffic is then analysed and *network traces* are produced. Follows a description of how this data is composed. Different meaning of words *kinds*, *types* and *classes* of botnets used in this section are defined in Section 1.4.2.

1.2 Network traces

Past studies [5, 7] have used *deep packet inspection* to detect botnet threats. This means they were modelling the content of the exchanged network packets in order to perform their analysis. This approach is however quite slow, and it is not reliable in case botnets encrypt their traffic. For this reason we base our study on high level data, which does not contain informations about the content of the network packets exchanged.

The data we use was mainly collected from *Anubis*[3] in 2009 in form of NetFlow[4], and it is composed of around 1.8 million flows. The use of NetFlow data, as suggested by Tegeler [16], makes the detection system reliable even when the packets are encrypted. In this document we will refer to the data presented in this section as *network traces* or simply *data*, differently from *dataset*, which is composed of the features extracted from *network traces* (as presented in Section 3). We here illustrate through definitions in a top-down fashion how this data was collected.

As we described before, the collection system was located between a bot and the C&C server. This means that from a high level prospective we observe all the network traffic between the infected computer and the Internet. We define a *network trace* to be all the network traffic collected in a certain amount of time (i.e. from the moment we start to the moment we decide stop the collection system). A *network trace* can be logically divided into *network flows*: since we work at 3rd layer of TCP/IP suite, TCP flows can be defined as the TCP traffic within a connection establishment and a connection termination. Since UDP is a stateless protocol, we can define a UDP network flow to be the traffic within a certain window, or until no communication happens for a certain timeout τ.

From a network flow defined as above we can extract some high level features, such as those defined in Table 1. It is quite clear that we cannot directly use these features for prediction. This is further explained in Section 3. For this reason a part of this project was spent in selecting and extracting suitable features from this data.

We were given a set of network traces generated by different kinds of botnets, as shown in Table 2. They are 9 kinds of botnets, and it makes sense to divide them into three main groups, respect to their type: *HTTP based*, *IRC based* and *p2p based*. In fact there are many similarities between them. However, from our experiments we discovered that traces from `mebroot` are

[3] `http://anubis.iseclab.org/`

[4] `http://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html`

1	**timestamp**	at which day, hour, minutes, seconds the flow happened
2	**duration**	duration of the flow
3	**srcIP**	source IP address (2nd layer TCP/IP suite address)
4	**srcPort**	source port (3rd layer TCP/IP suite address)
5	**dstIP**	destination IP address
6	**dstPort**	destination port
7	**proto**	3rd layer TCP/IP suite protoocol ('TCP' or 'UDP')
8	**txPkts**	transmitted packets
9	**rxPkts**	received packets
10	**txBytes**	transmitted Bytes
11	**rxBytes**	received Bytes
12	**totPkts**	total packets = txPkts + rxPkts
13	**totBytes**	total Bytes = txBytes + rxBytes
14	**flags0**	flags from the source
15	**flags1**	flags from the destination
16	**assured**	'y' if the flow was assured, 'n' otherwise
17	**finished**	'y' if the flow was finished, 'n' otherwise

Table 1: Features of a network flow

easily separable from all the other classes. We thus will consider from now on to have 4 classes of botnets: *HTTP based*, *IRC based*, *p2p based* and *mebroot*.

1.3 Approach

We give here a general overview of our approach. Because of the research nature of this project, this system was progressively created by performing different experiments. Section 2 shows the process we followed to create the system, starting from existent literature, while here we present the structure of our system.

The idea of this project is to cluster objects from our dataset by using Conformal Clustering, and evaluate the clustering respect to their classes. Our claim is that, if our system is able to group in different clusters the different kinds of botnets, it will be also able to detect each of them in a real world setting. Later in this report we will give an explanation of the criteria we chose to test the performances of our system.

From a high level perspective, our system is composed of four logical

File name prefix	Type of botnet
`http_mebroot`	HTTP based
`http_results (TORPIG)`	HTTP based
`new_results`	HTTP based
`irc_results`	IRC based
`p2p_results_storm_e`	p2p based
`p2p_results_storm_gateway`	p2p based
`p2p_results_storm_u1`	p2p based
`p2p_results_storm_u2`	p2p based
`p2p_results_storm-`	p2p based

Table 2: File prefixes in our data, and respective botnet types

blocks:

- Features extraction
- Preprocessing
- Conformal Clustering
- Evaluation

The data we had, as suggested in the previous subsection, needs first to be parsed and features must be extracted from it. In fact, network traces cannot be used directly as informations for our model. From *Features extraction*, which will be extensively analysed in Section 3, we obtain a *dataset*. The section suggests many novel features respect previous research, and a technique, which is new to the best of our knowledge, to extract periodicity from timestamps of network traces by using *Partial Autocorrelation Function* (PACF). In order to perform Conformal Prediction we need, for motivations expressed in Section 4.3, our dataset to be represented in as few dimensions as possible. We decide to use *t-Distributed Stochastic Neighbor Embedding* (t-SNE) as a dimensionality reduction algorithm, which is able to create a faithful two dimensional map of the high dimensional objects of our *dataset*. In order to apply t-SNE it's however necessary that the features of the dataset are all consistently normalized. *Preprocessing*, presented in Section 4, covers both normalization and t-SNE embedding. A preceding log-transformation, lately motivated, is also part of *Preprocessing*. Section 5 presents *Conformal Clustering* method, by which we create predictions for

a grid of points, create clusters of them, and predict clusters for new objects. *Evaluation* is done on all the steps, in order to progressively check them. In this context, a set of novel coefficients called *SIM-DIFF* is proposed in Section 3 to visualize the contribution of each feature in t-SNE for our clustering problem. t-SNE is evaluated by using the natural parameter Kullback-Leibler divergence. *Conformal Clustering* is evaluated by *Average P-Value*, as efficiency criterion, and *Purity*, as accuracy criterion. The use of the latter is motivated and theoretically evaluated in Section 5. After singularly analysing each component of the system we perform some conclusive experiment in Section 6 to evaluate the general method followed. In this experiment we introduce the use of a *train* and *test* set for validating our results.

1.4 Conventions

We briefly define here some conventions useful to read this document.

1.4.1 Objects and examples

In this report we will refer to *example* as the collection of an object and its label together. Object is a generic dataset row. We may refer to an object as a "feature vector", to focus the attention on it as a collection of features. Objects from the dataset are indicated by x_i. Also 2-D entries from t-SNE embedding are called objects, because they are unlabelled examples, and we refer to them as z_i.

1.4.2 Botnets kinds, types, classes

Kinds of botnets means all the 9 kinds as presented in the first column of Table 2. *Types* of botnets refer to their protocol: *irc*, *http* and *p2p* based. *Classes* or *labels* refer to the partition of *kinds* of botnets we made: *irc*, *http*, *p2p*, *mebroot*.

1.4.3 Traces and dataset

It is important to make a distinction between what we call *network traces* and *dataset*. The former, also called *data*, represent the data collected, introduced in Section 1.2, the latter is the dataset we generate with feature extraction in Section 3. We may say that *network traces* are raw data, and *dataset* is composed of the meaningful informations extracted from them. In

general, where nothing specified, we will mention *features* as the attributes of *dataset*.

1.4.4 Network protocol notation

When referring to a network protocol at n-th layer, we should specify which notation we use. In fact, otherwise, it would be unclear if we are referring to *TCP/IP suite* (4 layers) or to *ISO/OSI* model (7 layers), which can lead to ambiguities. So far in this section we have always indicated the convention used, by saying for example "TCP is a protocol at third layer of TCP/IP suite". However, since this form is quite redundant, from now on we will implicitly refer to TCP/IP suite.

1.4.5 Source code listing

In order to execute commands the reader should first obtain the source code as explained in the Appendix, and install all its dependencies. Source code within this report is conveniently presented in three different ways, each of which has a precise scope. We here give a summary of them.

Shell command In order to execute it properly, the reader should enter the directory *above* `code/` , and execute the listed command. By using this notation we want to show how to execute a script from within the shell. Example:

```
$ python -m code.scripts.create_dataset [...]
```

***Python* interactive session** We use this form when we need to show a little *Python* snippet of code, which can be executed from its *interactive environment*. In order to get the same, type in a shell from the directory above `code/` :

```
$ python
Python 2.7.5 (default, Oct 27 2013, 22:30:06)
...
Type "help", "copyright", "credits" or "license" for
more information.
>>>
```

From now on the reader can type the commands after `>>>`. The `>>>` prologue is in this report omitted, in order to facilitate copy-paste from pdf.

To exit this environment press: `ctrl-D`. In one case *R* interactive session commands are presented in this report. For this circumstance the reader will need to enter from a shell the command:

```
$ R
R version 3.0.2 (2013-09-25) -- "Frisbee Sailing"
Copyright (C) 2013 The R Foundation for Statistical
 Computing
...
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

to enter the *R* interactive session. As before, `>` prologue will be omitted. As before, to exit this environment press: `ctrl-D`. Code presented in interactive session is usually a proof-of-concept for performing specific tasks (such as plotting some results). For this reason it is not included in the code obtained as explained in Appendix.

Repository Appendix will provide instructions to obtain and use the code discussed and not listed here.

2 Background research

This section gives an outline of background research conducted for the different parts of the system. Because of this, it also gives an introduction to how the project developed toward the system creation.

2.1 General idea

The main idea of this project was to experiment with the problem of botnets detection by using the approach which Smith et al. [14] applied to anomaly detection of trajectories in the maritime domain. By botnets detection, as previously explained, we here intend to label as "malicious" the network traffic produced by an infected computer which is part of a botnet. In respect to our problem Smith et al. had some different aspects: i) maritime trajectories, as explained further later, can have arbitrary length, and this

is a problem when creating a model, and ii) in their context it made sense to produce *positive* examples (i.e.: artificial trajectories to be labelled as anomalies). Their approach to the first problem was using a dimensionality reduction algorithm, specifically *t-SNE* (t-Distributed Stochastic Neighbour Embedding), which is able to faithfully preserve the distances between objects in high dimensional data when embedded into lower dimensional data. The algorithm was in their case fed with the Hausdorff distance matrix, as Laxhammar et al. suggested before [10]. t-SNE algorithm is presented by Van Der Maaten in [19], and will be further discussed in Section 4.3. Our system will be using t-SNE, but for other motivations, which allows us to use *Euclidean* distance. The second difference raises from reasons explained in the previous section and in Sommer [15]: in our case we couldn't produce any reasonable "normal traffic". This is an important constraint of our problem, and because of it we focused on distinguishing between different kinds of botnet. Our claim is that if we are able to produce clusters keeping separate different botnets threats, then we are also able to detect them in a real world setting in presence of normal traffic. In terms of this assumption our approach is very similar to the one used by Tegeler et al. for *BotFinder* [16], which aims at clustering separately different kinds of botnets. Within this section we will outline similarities and differences respect to their approach.

2.2 Features

Previous studies on botnets detection [5, 7] used *deep packet inspection*: they modelled the content of exchanged packets within the botnet to perform their predictions. However, since this approach is not reliable for botnets encrypting packets, we make use of high level data (NetFlow), as suggested by Tegeler et al. [16], and extract from them features to create a dataset. This makes our approach working with the frequent case of encrypted traffic in botnets communications. Furthermore, this aspect makes our approach privacy aware, since no user data is read, and IP addresses can be obfuscated before analysis. We also noticed that detection systems from Tegeler et al. [16] and Gu et al. [6] only used a few of the available informations, which we show to be useful in Section 4.2. Respect to their approach we introduce the use of: percentage of *TCP* or *UDP* protocol use, variance on different informations, a set of features considering the percentage use of destination port numbers in standardized partition, and a set of features taking care of the source IP address (see Section 3). In Section 3.3 we suggest a novel -to the best of our knowledge- algorithm for periodicity detection. This algorithm is based on the use of *Partial Autocorrelation Function* (PACF),

which has shown to work well even under certain degrees of noise. From this we make use of the derived new feature, *frequency*, which we believe to cover informations of *BotFinder*'s *average time* between communications and *period*. In fact, since our algorithm is able to detect whether periodicity exists, we do not consider the average time between communications as a feature, because in presence of "normal traffic" this may be highly dependent to the context we work in. Section 3 extensively explains how we perform extraction of our features. The total number of features of our created dataset is 18.

2.3 Dimensionality reduction

Smith et al. [14] applied a dimensionality reduction algorithm to their system mainly because they had $4D$ features, where D could have been different between dataset objects. For this reason, they also used Hausdorff distance matrix, which is able to compute distances between sets. As explained before, we did not have the same problem. However, we faced the high dimensionality as a problem when using Conformal Clustering. In fact, for applying Conformal Clustering to our data we needed to create a p-dimensional grid of p-values, where p is the number of the features. However this meant that we needed to compute ℓ^p p-values, where $\ell = 18$ is the length of one grid side, which would be computationally infeasible for large resolution. Furthermore, clustering in high dimensions may lead to have objects too separated, and thus to produce a large number of clusters. This will be explained extensively in Section 4.3 and Section 5. We decided to apply t-SNE as Smith et al. did, because of its property of keeping the relative distances of objects close between high and low dimensional data. The choice of t-SNE algorithm parameters will be explained in Section 4.3. A good source to understand how to evaluate it was the home page of t-SNE [18] , were a *FAQ* section suggests to run the algorithm for many PRNG seeds (t-SNE is a non deterministic algorithm) and with different parameters, and to look for the one that produced the smallest Kullback-Leibler divergence (KL divergence). In fact, t-SNE works by progressively trying to reduce KL divergence, which represents a divergence between the similarities (in form of probability) of objects in high dimension, and the ones of the objects in low dimension. Our choice was, as will be shown later, to test different parameter settings within a suggested range, and to look for the best KL divergence.

2.4 Conformal Clustering

Conformal Clustering makes use of Conformal Prediction, introduced by Vovk et al. [20], to create sets of predictions. The idea of distribution free predictions and Conformal Clustering appears in Lei [12, 11]. Laxhammar et al. [10] and Smith et al. [14] use p-values grid to create clusters of objects. We follow their approach to produce clusters, and predict labels for new objects by looking at neighbouring clusters (cluster containing one object which has distance 1 or less from the new object). We use smoothed conformal predictor to compute p-values, as it was recommended in Gammerman et al. [4] for exact validity.

2.5 Performance criteria

Smith et al. [14], who both used t-SNE and Conformal Clustering on a dataset, introduced *Average P-Value* (APV) as an efficiency criterion and used partial Area Under Curve (p-AUC) for validity. However, since we could not use any "normal traffic" labelled data, we had to replace the validity criterion with a cluster-related one. For this aspect we referred to *Introduction to information retrieval* book from Manning, Raghavan and Schtze [13] to find a clustering based evaluation criterion. Here, they present four different kinds of clustering evaluation. The first one, *Purity*, is a simple measure of how many classes appear in the same cluster. The others, such as *Rand index* and *F measure*, are useful to make comparisons, for example, respect to the number of clusters, or to False Positives or False Negatives considerations. In Conformal Clustering we are not interested in how many clusters we generate, since some similar examples may be far but still clustered with other examples of the same class. For this requirement, we chose to use *Purity*, which we may refer to as *accuracy* criterion. Performance criteria will be further discussed later in Section 5.

3 Features selection

This section presents the work done to select and extract the features from the *network traces* presented in Section 1. As explained in Section 1.4, we will refer to the network traces we were given as *data*. We here remind that every network trace is composed by many network flows, which have the attributes described in Table 1. Every row in our dataset will be then derived from a network trace (e.g.: one feature of the row can be the average transmitted bytes for all flows in the network trace). We here also introduce

a visualization and evaluation algorithm to observe the specific contribution of each feature for our successive dimensionality reduction and clustering. The results of this evaluation are then shown and analysed.

3.1 Challenges

Translating security related informations to consistent data for Machine Learning analysis can be quite difficult. For example, averaging a port number as a feature would make no sense at all: port numbers are qualitative objects, and they can have a particular meaning respect to the standards defined by Internet Assigned Numbers Authority (IANA)[5]. From the other hand, it may be meaningless to use them as proper qualitative features, because they can be almost arbitrarily changed, regardless to their standardized meaning. For this reason some time was spent on understanding which feature could be used, and how to make use of our data.

3.2 Selected features

Six kinds of features were extracted.

3.2.1 Periodicity

In some kinds of botnets, bots communicate periodically with the C&C server. For this reason it seemed a good idea to use communication period, as Tegeler et al. [16] did, as a feature. The algorithm to extract this feature is presented afterwards. This feature was extracted by using *timestamps* of network flows in a trace.

However, since not all the botnets communicate periodically, in some cases the feature period T, measured in seconds, would need to be ∞. Now, as this would have been difficult to handle for our learning algorithms, we chose to use **frequency [Hz]** as a feature. This feature can be set to:

$$\textbf{frequency} = \begin{cases} 0, & \text{if } T = \infty (\text{no period}) \\ 1/T, & \text{otherwise} \end{cases}, \tag{3.1}$$

for $T > 0$. This quantity can be easily handled by learning algorithms.

3.2.2 Duration

Sample mean and variance of network flows *duration* (Table 1) were used as features.

[5] `https://www.iana.org`.

Range	Denomination
0, 1023	System ports
1024, 49151	User ports
49152, 65535	Dynamic and/or private ports

Table 3: *IANA* tripartition of port numbers

3.2.3 Bytes

Sample mean and variance were used as features from *received*, *transmitted* and *total* (received+transmitted) Bytes of network flows.

3.2.4 Protocol

In every network flow there is a flag corresponding to the 3rd layer protocol: "TCP", "UDP". For our dataset we used as features the percentages of use of each of the two protocols in a network trace.

3.2.5 Ports

The source port number is selected randomly by the Operating System. For this reason, source port numbers usually do not contain any relevant information for us. However, destination port is meaningful for a flow: since many port are conventionally associated to a service, they can be a characteristic of a botnet and describe its protocol.

As explained before, averaging port values would not make much sense. A solution would be to create P features, each of them corresponding to a port number, describing the percentage of use of that port. However, TCP/IP port numbers have cardinality $P = 65536$, which would mean adding 65536 new features to our dataset. This is not feasible.

Our approach is based on a standardized tripartition of port numbers assigned by IANA [6]. The partition is as in Table 3. We consider as features the percentages of usage in a network trace of ports in each partition.

3.2.6 IP addresses

IP addresses cannot be used as numbers. Furthermore, even by using them as qualitative attributes, it may be difficult to deal with them.

[6] `https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.txt`.

Our idea is to use them to produce more features, from the features already presented here. In Section 1 we said that our data contains network flows coming from the bot to the C&C, and from the C&C to the bot. However, in our network traces, features such as **srcPort** or **txBytes** are always of the entity who initiated the flow, and features like **dstPort** or **rxBytes** are of the entity who responded. We decided to introduce a new set of features. After computing the features as before, we swap where needed the elements in our network traces, so that "src*" or "tx*" always correspond to the bot's IP address, "dst*" or "rx*" corresponds to the entity it is communicating with. From these newly created traces we estract mean and variance of received and transmitted bytes, which we call: **rxBytes_mean_dir**, **txBytes_mean_dir**, **rxBytes_var_dir** and **txBytes_var_dir**. Here the suffix "_dir" shows we calculated the features after inverting the direction.

3.2.7 Features list and dataset creation

Follows a list of the 18 features above explained:
duration_mean, **duration_var**, **totBytes_mean**, **totBytes_var**, **frequency**, **TCP_usage**, **UDP_usage**, **wkn_ports_usage**, **reg_ports_usage**, **dyn_ports_usage**, **txBytes_mean**, **txBytes_var**, **rxBytes_mean**, **rxBytes_var**, **txBytes_mean_dir**, **txBytes_var_dir**, **rxBytes_mean_dir**, **rxBytes_var_dir**.

The dataset we will use through this report can be created by typing, from a command line in the directory above `code/` :

`python -m code.scripts.create_dataset --src [traces] --dst [dataset] --blacklist [blacklist]`

where *traces* is the directory containing our network traces (`data/data-traces/`), *dst* is the destination suffix which can be set to `data/dataset/dataset` , and *blacklist* is an optional blacklist telling which files to skip from `data/data-traces/.` In our case *blacklist* must be `code/scripts/dataset_blacklist.txt.` This script will produce three files: dataset.csv (the dataset), dataset-labels.txt (list of labels) and dataset-ids.txt (list labels translated to numbers). This script tries to automatically detect the IP address of the bot of network traces. We assume the IP address of the bot: i) is private, ii) appears in every row of a network trace. However, it may happen that the scripts is not able to automatically detect it. In this cases, it will prompt something like:

```
ipBot wasn't found, enter it manually
Possible IPs: [('0.0.0.0', '255.255.255.255'),
```

```
('192.168.183.160', '116.50.0.2'), ('192.168.183.160',
'83.26.4.2'), ('192.168.183.160', '87.249.21.2'),
('192.168.183.160', '125.244.24.2'), ('192.168.183.160',
'24.117.41.2'), ('192.168.183.160', '84.244.69.2'),
('192.168.183.160', '65.8.122.2'), ('192.168.183.160',
'70.158.171.2'), ('192.168.183.160', '192.168.183.2')]
Enter the IP:
```

If this appears, the user will need to insert the address more likely to be the bot's address, respect to the previous criteria. In this case: `192.168.183.160`. This issue can be solved in future versions of the code.

Please note that before executing these commands, the user should check to have satisfied all the software requirements, as in Appendix.

3.3 Periodicity detection using PACF

We here present the problem of periodicity detection in our dataset. In our context, periodicity detection means understanding if network flows in our trace happen periodically, and in that case what its period T is. From the timestamps of the flows we create a time series y of zeros and ones where:

$$y(t) = \begin{cases} 1, & \text{if flow occurred at time } t \\ 0, & \text{if no flow occurred at time } t \end{cases}. \tag{3.2}$$

Detecting periodicity in such a time series is a known problem in literature. A common solution is to compute the *Discrete Fast Fourier Transform* (FFT), take its module, and count the peaks of the resulting function. It turns out that the number of peaks of a so created function is the period of the series. However, this approach needs to be able to count FFT peaks, which can be difficult under noisy conditions, where the height of the peaks may also vary.

We suggest a method to detect periodicity which has shown to work well under noisy conditions, and which is pretty simple to apply. Our experiments also showed our method works well for other problems, such as detection of multiple periodicities, but this second task is not presented in this report because not properly related to our discussion.

As a periodicity detection method we suggest the use of Partial Autocorrelation Function (PACF) on our series y and a certain threshold condition on the function. We observed that applying PACF to a series usually shows

an high peak corresponding to the period. For example, the following R code:

```
N <- 50000
period = 23
t <- 1:N
y <- rep(0, N)
y[t%%period == 0] <- 1
pacf(y, 250)
```

produces as in Figure 3 a peak at lag 23, which is our period.

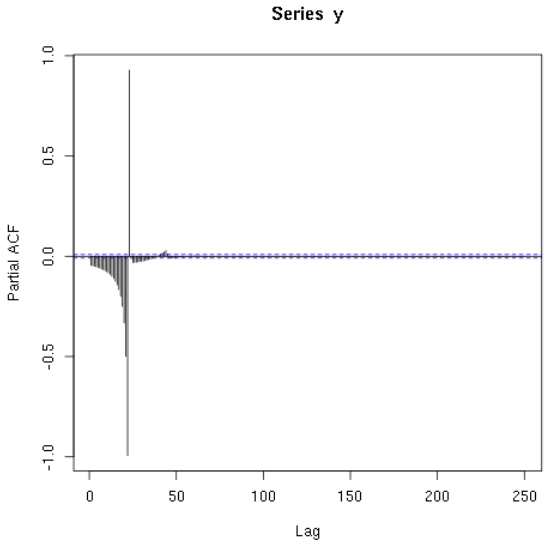

Figure 3: PACF on time series with periodicity 23

This is also true under noisy conditions. To test this we performed an experiment using different degrees of noise $\nu \in \{0, 0.01, 0.04, 0.05, 0.1, 0.2, 0.3, 0.4\}$ and plotting the results. Code in `code/scripts/test-pacf-artificial.R` shows how to obtain the same. In this experiment we assumed to know one periodicity existed. Our prediction rule was to take the PACF lag with the highest value as a period. Follow the detection results produced by this experiment:

Noise	Detection (%)
0	100.0
0.01	100.0
0.04	100.0
0.05	100.0
0.1	98.0
0.2	79.6
0.3	67.3
0.4	52.0

So far we have worked on artificial data, where we knew some periodicity existed. When we do not know if there exists any period in our series, the problem gets more complicated, and it can become difficult to understand if the height of some peaks is relevant. Many criteria were explored to detect the peak under these conditions, and further studies may focus on this aspect. However, since this was not a particularly delicate aspect of the project, it looked adequate the use of an empirical rule for creating our dataset. We first plotted all the PACFs corresponding to time series created as in Equation 3.2 from network traces timestamps. We then looked for a threshold which wasn't infected by noise by visual inspection, and chose this threshold to be $\vartheta = 0.7$. Our method showed that *irc* based botnets used to communicate with their C&C periodically with a period of 21 seconds. For creating the dataset, the period detection rule was therefore to compute PACF on a time series y, look for the highest peak, and if it was greater than ϑ select its lag as the period. Otherwise, no period would have resulted for the series.

As explained before in this section, using **period** directly as a feature would be difficult when no periodicity is detected. For this reason we used **frequency** as in Equation 3.1, which is equal to $1/T$ when period $T > 0$ and 0 when no period is detected ($T = \infty$).

3.4 Features evaluation

During this project it seemed important to understand the contribution of dataset features. This may be helpful for future works to only focus on the most important features in order to improve the performances.

As will be explained in the next chapters, it was necessary to use an *euclidean distance* based dimensionality reduction algorithm on the dataset before using conformal clustering on it. The idea of the algorithm we here present to evaluate features is to show how much each feature in the dataset

contributes to:

- make object from the same class to be far
- make objects from different classes to be far

We specifically define two coefficient vectors: *SIM*, *DIFF*. Their length, d, is the number of the features in a dataset row.

SIM[i] represents an approximation of how much the i-th feature contributes to make objects from the same class to be far. In our definition SIM[i] is the average contribution of i-th feature to the distance of two objects with the same label. In general we want SIM to be as small as possible. It is calculated as in Algorithm 1.

Data: feature vectors $x_1, x_2, ..., x_n$, labels $y_1, y_2, ..., y_n$
Result: *SIM* coefficients
$SIM = [\ ]$
$X = \{x_1, ..., x_n\}$
uY = entries in Y without repetition
for *i = 1 to d* **do**
  $X_i = i$-th column of X
  $SIM[i] = 0$
  for *class in elements of* uY **do**
    $fromclass = X_i$[where $Y == class$]
    $pdist$ = pairwise_distances($fromclass$)
    $SIM[i] = SIM[i]$+ (sum($pdist$) / length($pdist$))
  end
  $SIM[i] = SIM[i]/length(uY)$
end

Algorithm 1: Calculation of *SIM* coefficients

DIFF[i] represents and approximation of how much the i-th feature contributes to make objects from different classes to be far each others. Two objects from different classes have in average distance $DIFF[i]$ respect to the i-th feature. In general we want this coefficient to as big as possible. Follows the algorithm to calculate DIFF:

Data: feature vectors $x_1, x_2, ..., x_n$, labels $y_1, y_2, ..., y_n$
Result: *DIFF* coefficients
$DIFF$ =[ ]
$X = \{x_1, ..., x_n\}$
uY = unique entries in Y: our classes
for *i in 1 to d* **do**
    X_i = i-th column of X
    $DIFF[i] = 0$
    for *class in elements of* uY **do**
        $fromclass = X_i$[where $Y == class$]
        $notfclass = X_i$[where $Y! = class$]
        $pdist$ = respective distances between elements in $fromclass$
        and elements in $notfclass$
        $DIFF[i] = DIFF[i]$+ (sum($pdist$) / length($pdist$))
    end
    $DIFF[i] = DIFF[i]/length(uY)$
end

Algorithm 2: Calculation of *SIM* coefficients

From these definitions we want $SIM[i] \ll DIFF[i]$, so that i-th feature keeps distant similar examples (i.e.: examples with the same label) less than it keeps distant dissimilar examples (i.e.: examples with different labels). On the other hand, if $SIM[i] >= DIFF[i]$ we understand that i-th feature is not relevant or it even makes our clustering worse. We performed our experiments with Euclidean distance as a distance measure. However, other distance measures can be tested in the future.

We represented these two coefficients with barplots, where green bars are SIM coefficients, red bars are DIFF coefficients. As explained before, we want green bars to be as small as possible, or at least smaller than the neighbour red bar. All the features must be normalized before calculating the coefficients, so we can compare the contributions of the features, and potentially look for the most important of them. The code for calculating sim diff was implemented in the function **sim_diff()** in **code.evaluate.eval_features**. *SIM-DIFF* results are shown in next section in Figure 4, when comparing different preprocessing. Next section also shows the code to obtain the same.

SIM-DIFF coefficients helped in selecting the best preprocessing for our dataset. In this setting, thanks to these coefficients, we were able to choose if and when to apply logarithm transformation on some of our features. Thanks to these coefficients we also spotted a bug in the dataset creation

code, which formerly was producing a few irrelevant features. In general we believe that *SIM-DIFF*, thanks to their intuitive representation, can be useful to have an idea regarding how the dataset is composed and how some operations influence a dataset. Graphical examples of this will be presented in the next section.

4 Preprocessing

This section presents the processing phase of our dataset before clustering. Preprocessing included, in order, logarithm transformation of some features, normalization in [0,1] and dimensionality reduction by using t-SNE. We express in this section the motivations of our choices, we give a brief overview of t-SNE algorithm, and we make some evaluation of t-SNE embedding on our dataset. Note that the implemented code for log-transformation and normalization is separated from the code for t-SNE. For the latter in fact we used the implementation from the Open Source package **scikit-learn**, as explained in the Appendix.

4.1 Normalization

Normalization was needed for both features evaluation, as seen in Section 3, and t-SNE embedding. The only requirement for them was the normalization to be consistent for all the features. We applied a normalization in [0,1], before dimensionality reduction, as follows:

$$x_{01} = \frac{x - min(x)}{max(x) - min(x)}, \tag{4.1}$$

where x is the vector to normalize.

4.2 Log-transformation

From past studies it looked important to take the logarithm of bytes-related features, such as: *txBytes*, *rxBytes*, *totBytes*. This because these features have a large range, and log-transformation can improve their contribution.

We chose to apply log-transformation directly on network traces features: txBytes, rxBytes, totBytes. This is automatically done by **create_dataset** script, which can be run as shown in Section 3. Before choosing whether to apply or not this transformation, we however performed an experiment using *SIM-DIFF*, as introduced in Section 3.4, on a dataset created without

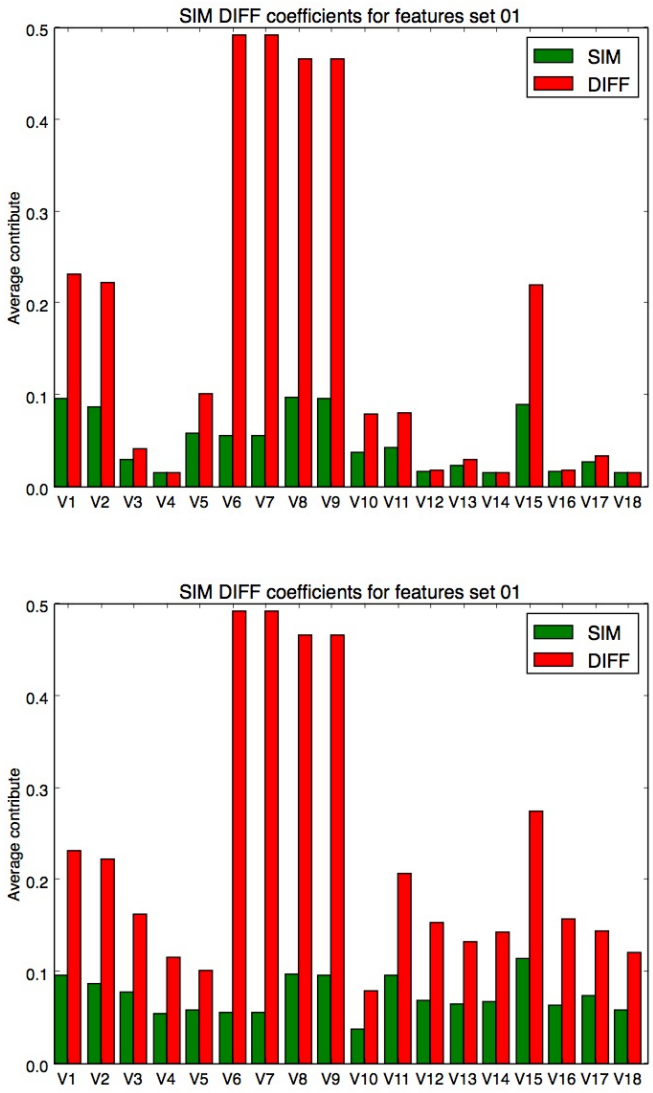

Figure 4: *SIM-DIFF* coefficients when no log-transformation is applied (above) and when log-transformation is applied on network traces, before creating the dataset (below)

log-transformation and one with log-transformation. The results of the two experiments are in Figure 4.

In the figures, we want to consider the bytes-related features: $\{V3, V4, V11, V12, ..., V18\}$. The improvements obtained by introducing logarithm transformation are evident: between the two figures we see the gaps between *SIM* and *DIFF* increasing, and some features which used to be irrelevant (i.e. $SIM \simeq DIFF$), such as $V4$, $V14$ and $V18$, gained importance. In this case the graphical results were evident. If in some cases they are not so evident, we suggest to consider the two difference vectors: $DIFF - SIM$ of the first dataset, $DIFF - SIM$ of the second dataset, and look for the biggest values for each feature.

Logarithm transformation was applied to network traces before creating dataset. After this, features were normalized in [0,1], and *t-SNE* was run on them. Further studies may try to apply log-transformation on the dataset itself, over the features derived from bytes-related network traces attributes.

4.2.1 Code

In order to preprocess the dataset created in Section 3, the reader should run, from the directory above `code/` :

```
python -m code.scripts.preprocess_dataset --dataset [dataset
] --out [dataset-preproc]
```

where *dataset* should be the dataset created before, `dataset.csv` , and *out* should be the output file, such as `data/dataset/dataset-preproc.csv.`

To run the features evaluation over the created dataset, as shown in the previous subsection, the reader should run:

```
python -m code.scripts.evaluate_features --dataset [preprocessed
 dataset] --labels [labels]
```

where *dataset* should be the preprocessed dataset just created, `data/dataset/dataset-preproc.csv` , and *labels* should be the labels file, `data/dataset/dataset-labels.txt` . This will output the *SIM-DIFF* coefficients, and show a barplot visualizing them.

4.3 Dimensionality reduction

4.3.1 The reasons

In *Conformal Clustering* we are given a set of unlabelled objects Z and we want to produce clusters of them, by using the significance level ε to regulate the hierarchical depth of the clustering. A method to do this is to create a grid of points, calculate p-values for each of them (we can call this *"p-values grid"*), and cluster those points for which p-values are greater than ε. For example, if we are working on a 2-dimensional space we construct a grid of

ℓ points in the (min, max) for both the dimension respect to our training set Z, calculate p-value for each new observation (point in the grid) respect to objects in Z and trim the results by using ε. The parameter ℓ regulates the resolution of our grid. This topic will be explained more in depth in Section 5.

However, this means that in 3-dimensions we would need to create a 3-D grid, and so on. Given that we split each dimension in ℓ points, having p features we would need to compute ℓ^p p-values to create the p-values grid. Now, this rapidly becomes computationally infeasible when we want to increase the resolution. For this reason, having selected 18 features for our dataset, we opted for a dimensionality reduction algorithm, in order to reduce p to 2.

A second important reason for using dimensionality reduction is that clustering on high dimensional data tends to keep objects too separate, thus causing the algorithm to generate too many clusters.

4.4 t-SNE

We chose to use as a dimensionality reduction algorithm *t-Distributed Stochastic Neighbor Embedding* (t-SNE), introduced by van der Maaten et al. in [19], because of its capacity of keeping similar objects in high dimensional data close in the low dimensional map. We here give an overview of this algorithm, by mainly referring to van der Maaten paper [19].

t-SNE is a visualization technique which is able to faithfully transform high dimensional data into a two or three dimensional map. It is mainly based on a previous study from Hinton et al. *Stochastic Neighbour Embedding* (SNE) [8], from which it takes the general idea and of which corrects a few weak aspects. The goal of SNE is keeping close in the low dimensional map objects which are similar in high dimensional data, and far in low dimensions dissimilar objects in high dimensions. This algorithm could run by simply knowing the pairwise probability indicating how much high dimensional objects are similar. However, in a general context, we find it useful to empirically determine these probabilities from our data. We define $p_{i|j}$ to be the probability that high dimensional object x_i chooses x_j as neighbour. This probability is computed in SNE assuming a Gaussian distribution centered in x_i with a certain variance σ_i. This is expressed in the form:

$$p_{i|j} = \frac{\exp(-||x_i - x_j||^2/2\sigma_i^2)}{\sum_{k\neq i}\exp(-||x_i - x_k||^2/2\sigma_i^2)}, \text{ if } j \neq i, \tag{4.2}$$

and $p_{i|i} = 0$.

Similarly, we then define $q_{i|j}$ to be the analogous probability for low dimensional objects y_i y_j. Its expression is therefore:

$$q_{i|j} = \frac{\exp(-||y_i - y_j||^2/2\sigma_i^2)}{\sum_{k \neq i} \exp(-||y_i - y_k||^2/2\sigma_i^2)}, \text{ if } j \neq i, \tag{4.3}$$

and $q_{i|i} = 0$.

The basic idea behind SNE is to progressively decrease the mismatch between $p_{i|j}$ in high dimensional data and $q_{i|j}$ in low dimensional data. The measure suggested to describe this discrepancy is Kullback-Leibler divergence. Kullback-Leibler divergence $KL(P||Q)$ is a measure indicating how much of information is lost by approximating distribution P with Q. In case P and Q are discrete distributions, it is defined as:

$$KL(P||Q) = \sum_i P(i) \log \left(\frac{P(i)}{Q(i)} \right). \tag{4.4}$$

SNE minimizes progressively the divergence by using gradient descent over all the data points. In order to determine the variances σ_i SNE uses *perplexity*, which is a parameter set by the user. Perplexity is intuitively the measure of effective number of neighbours.

t-SNE modifies SNE, making it a faithful and computationally efficient algorithm. t-SNE makes use of symmetric SNE, proposed by Cook et al. [3], which has a cost function C of this form:

$$C = KL(P||Q) = \sum_i \sum_j p_{i|j} \log \frac{p_{i|j}}{q_{i|j}}, \tag{4.5}$$

where $p_{i|j} = p_{j|i}$ and $q_{i|j} = q_{j|i}$

An issue in SNE was the one van der Maaten et al. call "crowding problem". The crowding problem usually presents when trying to model high dimensional data into a lower dimensional space. In the conversion it can result that moderately far objects in high dimension do not have large enough space to be accommodated in low dimensions, respect to near objects. For this reason t-SNE uses a Gaussian distribution for converting distances from high dimension, while it uses Student t-distribution with one degree of freedom to convert distances in low dimension. The second distribution has the property of being heavy-tailed, thus allowing to model moderate distances in the high-dimensional space into larger distances in low dimension. The probabilities in low dimension are thus defined as:

$$q_{i|j} = \frac{(1 + ||y_i - y_j||^2)^{-1}}{\sum_{k \neq i} (1 + ||y_i - y_k||^2)^{-1}} \tag{4.6}$$

t-SNE has proven to work well in creating very informative 2-D or 3-D representations showing correlations between objects in high dimensions. We choose to use this method, as Smith et al. did [14], before our clustering. Our experiments on the general system, as will be shown in Section 6, indicate that our approach is valid.

4.5 Experiments, Results, Analysis

t-SNE creators van der Maaten and Hinton suggest in [18] to choose the best result by visual inspection. However this can be very subjective in our case, where visualization is not the main goal. In the same document, however, they recommend that a good approach is to look at the *Kullback-Leibler divergence*, which as shown before can give us an idea of how data in low dimensions is faithful to high dimensional data. Furthermore, since the initialization of the algorithm can be random, they also suggest to run the algorithm multiple times (i.e.: for multiple PRNG seeds) and then choose for the best one.

We conducted two experiments on t-SNE. In the first one we are interested in how KL divergence relates with different values of *perplexity*. The second experiment looks for the smallest KL divergence by testing many values for parameters such as *perplexity*, *learning rate* and *metric*. We here illustrate these experiments and analyse the results.

In the first experiment we choose to only trim *perplexity* parameter. Van der Maaten et al. suggest in [18] that perplexity should have some value between 5 and 50. Perplexity represents our smooth idea of the number of neighbours an object may have. All the other parameters are kept to the default of the software we are using: (**metric**='euclidean', **learning_rate**=100). A second experiment will consider also these parameters.

In order to run this experiment, the user should launch, from the directory above `code/` :

```
python -m code.scripts.evaluate_tsne --dataset [preprocessed dataset]
```

where *dataset* must be the preprocessed dataset, as `data/dataset/dataset-preproc.csv`.

This script will calculate t-SNE for perplexity in range $\{5, 6, .., 50\}$, and finally produce a plot of the results. Figure 5 shows this result. From the plot we can clearly notice a trend of KL divergence toward 0 when perplexity augments. Since KL divergence indicates how much our 2-D representation is unfaithful to high dimensional data, we are induced to think that moving perplexity beyond 50 will improve the system performances. Section 6 will

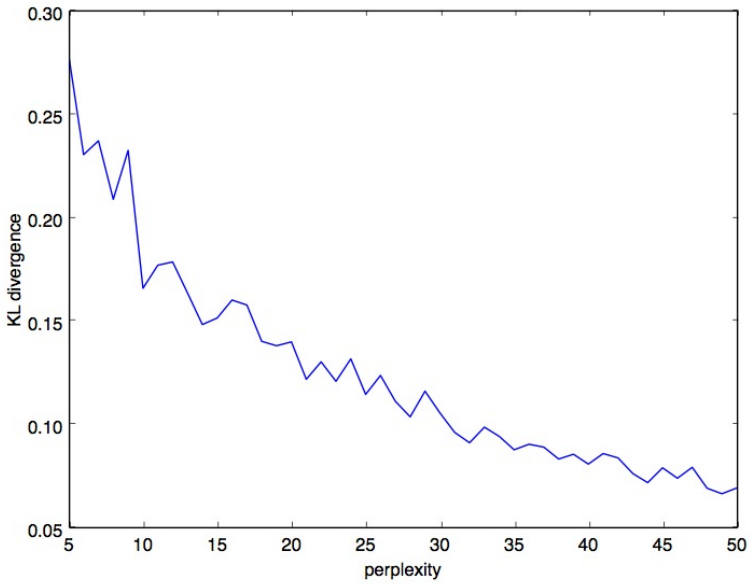

Figure 5: t-SNE: perplexity versus Kullbak-Leibler divergence

take into account this hypothesis, by considering other aspects of our system.

Our second experiment was focused on obtaining the best result by testing different values for t-SNE parameters on the dataset. In order to get the best results, we tested for each unique set of parameters 20 different initializations, by feeding the PRNG seed with numbers in $\{0, 1, .., 19\}$. The parameters were tested in these ranges of values:
perplexity in $\{5, 6, .., 50\}$
learning_rate in $\{50, 100\}$
metric in {'euclidean', 'cityblock', 'seuclidean', 'sqeuclidean', 'cosine', 'correlation', 'chebyshev', 'mahalanobis'}[7].

The script created for this experiment calculates t-SNE final KL divergence for all these values, for every seed, and it only prompts the "best result so far". The output of the script at the end of the experiment was as follows:

```
[i] perplexity: 49/50
```

[7]These distances are all defined in `http://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.pdist.html#scipy.spatial.distance.pdist` from which we used **pdist** function.

```
New best: +0.0630964384 {'perplexity': 49, 'learningr': 50,
 'metric': 'euclidean', 'seed': 15}
[i] perplexity: 50/50
New best: +0.0613313583 {'perplexity': 50, 'learningr': 50,
 'metric': 'euclidean', 'seed': 2}
New best: +0.0611046107 {'perplexity': 50, 'learningr':
100, 'metric': 'euclidean', 'seed': 2}

Overall best: +0.0611046107 {'perplexity': 50, 'learningr':
 100, 'metric': 'euclidean', 'seed': 2}
```

which essentially communicates the overall best set of parameters was (**perplexity**=50, **metric**='euclidean', **learning_rate**=100, **seed**=2), where KL divergence is 0.061105.

Although we see that Euclidean distance performed best, we also notice by the results of this experiment that metrics as 'seuclidean' and 'cityblock' had a good importance. In this case, 'seuclidean' stands for *Standardized Euclidean distance*, which is defined as:

$$d(u,v) = \sqrt{\sum_i (u_i - v_i)^2/\sigma_i}, \tag{4.7}$$

were σ_i is the variance computed over u_i and v_i, and 'cityblock' is the *City Block* or *Manhattan* distance:

$$d(u,v) = \sum_i |u_i - v_i|. \tag{4.8}$$

In order to reproduce the same, the reader can run, from the main directory:

`python -m code.scripts.best_tsne --dataset [preprocessed dataset]`

After this we may want to plot the best result. We can do this by running the following code in a *Python* interactive session from the main directory:

```
import numpy as np
import pylab as pl
import code.scripts.load as load
from sklearn.manifold import TSNE
from scipy.spatial.distance import pdist, squareform
```

```
# Settings
DATASET = 'data/dataset/dataset-preproc.csv'
LABELS = 'data/dataset/dataset-ids.txt'
# Settings after the experiment
METRIC = 'euclidean'
PERPLEXITY = 50
LEARNINGR = 100
SEED = 2
# Load dataset
X = load.load_dataset(DATASET)
Y = load.load_ids(LABELS)
# Compute TSNE
dist = squareform(pdist(X, METRIC))
t = TSNE(perplexity=PERPLEXITY, learning_rate=LEARNINGR,
          random_state=SEED, metric='precomputed')
Z = t.fit_transform(dist)
# Plot results
pl.scatter(Z[:,0], Z[:,1], c=Y, marker='o', cmap='prism')
pl.show()
```

This code produces a plot as in Figure 6. Note that in this representation every kind of botnet has its own color. We did not label here these classes because, as explained in Section 1, we will not consider each of them as a single class in our classification problem. We can although observe that objects with the same color (i.e.: from the same botnet kind) are pretty close each others, which suggests our clustering task will be easier. Later on in this report the objects on the figure will be labelled respect to their class.

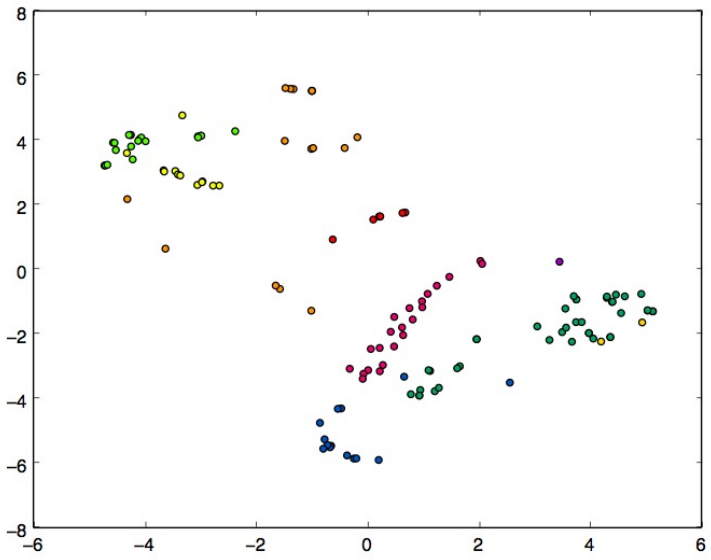

Figure 6: t-SNE with the best KL divergence, unlabelled classes. Every color represents a different kind of botnet.

5 Conformal clustering

In this section we introduce *Conformal Clustering*, a clustering method based on *Conformal Prediction*, and we then outline our experiments and analyse results.

5.1 The algorithm

Conformal Prediction allows to have a confidence measure on predictions. Given a bag of observations $D = \wr z_1, .., z_{n-1} \wr$, $z_i \in \mathbf{Z}$, a new object z and a significance level ε, conformal prediction allows us to determine if z comes from D with an error on the long run of at most ε. Confidence is defined as $1 - \varepsilon$. The only property required by conformal predictor is that the distribution P from which $\wr z_1, .., z_{n-1} \wr$ are generated is exchangeable. Note that exchangeability property is weaker than *iid*, since:

$$iid \xrightarrow{implies} exchangeable$$

We define a non-conformity measure $A : \mathbf{Z}^{(*)} \times \mathbf{Z} \to \mathbb{R}$, to be a function which accepts a bag of objects and an object z_i, and returns a scalar representing how much z_i is conform to the other objects. A large number in output indicates z_i is non conform to the others. The result of Conformal Prediction is a p-value p_n and a boolean answer indicating if the new object is conform to D. Follows a description of Conformal Prediction algorithm.

Data: Bag of objects $D = \wr z_1, .., z_{n-1} \wr$, non-conformity measure A, significance level ε, a new object z
Result: P-value p_n, *True* if z is conform to training objects

Set provisionally $z_n = z$ and $D = \wr z_1, .., z_n \wr$
for $i \leftarrow 1$ ***to*** n **do**
| $\alpha_i \leftarrow A(D \setminus z_i, z_i)$
end
$\tau = U(0, 1)$
$p_n = \frac{\#\{i:\alpha_i > \alpha_n\}| + \#\{i:\alpha_i = \alpha_n\}\tau}{n}$
if $p_n > \varepsilon$ **then**
| Output *True*
else
| Output *False*

Algorithm 3: Conformal prediction using new examples alone

where τ is sampled in *Uni(0,1)* to obtain a *smoothed conformal predictor*, as suggested by Gammerman et al. in [4]. This makes our conformal predictor to be *exactly valid*, which means that the probability of error equals ε.

In Conformal Clustering we use Conformal Prediction to create a bag of predictions Γ^{ε}, which contains the new objects which are conform to old objects D. As in Conformal Prediction, we select a non-conformity measure A and a significance level ε, then we make predictions and we create clusters of the predicted conform objects.

In order to do this, we first create a d-dimensional grid of ℓ points per side equally spaced, where d is the number of features. Since we apply Conformal Clustering after t-SNE embedding on the dataset, in our case $d = 2$. The grid side range is chosen by looking at max and min of the d features. We then compute a p-value for each of the grid points by using our training objects. Finally, we make a prediction for every point respect to the chosen significance level ε. The *True*-predicted points are then clustered

by following the rule: "two points z_i, z_j are in the same cluster if they are neighbours (i.e.: their distance on the grid is 1)". We can notice that ε is a sort of trim for regulating the hierarchy of our clusters. However, respect to other clustering algorithms such as *k-Means* and *Hierarchical clustering*, Conformal Clustering can use a smooth functions to detect neighbours, and in particular it can set a confidence level on the results. In the second experiment of this section we will illustrate this by a graphical example.

Once we produced clusters on the $\ell \times \ell$ grid, we can use them to make a cluster prediction for the objects z_i we used to train them. In fact, p-values grid points were clustered, but we do not know how old objects z_i must be clustered. In order to do this we can associate to z_i the clusters for which the distance between one point in them and z_i is less or equal to 1. This is done in our implementation by **code.conformal_prediction.clustering.predict()** function. This function, however, doesn't merge clusters in case z_i is associated to more than one cluster, which should be implemented in the future.

5.2 Non-conformity measure

A non-conformity measure is defined as a function $A : \mathbf{Z}^{(*)} \times \mathbf{Z} \to \mathbb{R}$, where $z_i \in \mathbf{Z}\ \forall i \in \{1, .., n\}$. Conformal prediction is proved to work for every non-conformity measure, but some of them can be more efficient. We analysed two non-conformity measures: *k Nearest Neighbours* (*k-NN*) and *Kernel Density Estimation* (*KDE*) with a Gaussian kernel. We here present how these two measures are calculated. The experiments and results by using them are afterwards analysed.

A_i, *k-NN* non-conformity measure, is calculated as follows. A_i for object z_i, given d_{ij} to be the j-th smallest distance between z_i and the bag $\wr z_1, .., z_n \wr \setminus z_i$, is:

$$A_i = \sum_{j=1}^{k} d_{ij}, \tag{5.1}$$

where k is the chosen number of neighbours.

Let A_i be a *KDE* non-conformity measure. Then for a given kernel function $K : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$, being d the number of features in a vector, is calculated as:

$$A_i = -\left(\frac{1}{nh^d} \sum_{j=1}^{n} K\left(\frac{z_i - z_j}{h}\right)\right). \tag{5.2}$$

In this expression we can see the term h, which is the bandwidth of the

kernel. We will go through experiments by trimming this parameter. The kernel K is a function centred around z_i. In our experiments we used a Gaussian kernel, which is defined as:

$$K(u) = \frac{1}{2\pi} e^{-\frac{1}{2}u^2}. \tag{5.3}$$

5.3 Evaluation criteria

Conformal Clustering, being based on Conformal Prediction, needs two evaluation criteria: we would like to both look at how clusters reflect our classes (*validity*) and how small is our prediction set (*efficiency*).

For the evaluation of our system we replaced *validity* with an *accuracy* measure. In particular, knowing the true label associated to every object, we decided to use *Purity*, presented in *Introduction to Information Retrieval* [13], as an accuracy criterion. This is formally defined as:

$$Purity(\Omega, \mathcal{C}) = \frac{1}{n} \sum_k \max_j \#\{\omega_k \cap c_j\}, \tag{5.4}$$

where $\Omega = \{\omega_1, ..., \omega_K\}$ is the set of clusters, and $\mathcal{C} = \{c_1, ..., c_J\}$ is the set of classes. *Purity* assigns to every cluster the most frequent class present in it, and sums the count of how many objects of that label are in that cluster. The resulted sum is then normalized by dividing it by the number of objects n. Purity must be carefully used, since it does not take into account the number of clusters. For this reason, having n objects, *Purity*$= 1$ may be easily achieved by creating n clusters. In our case we decided to use it, because we suggest Conformal Clustering is quite stable regarding the number of clusters, as this can be trimmed by the parameter ε. This problem should however be further investigated by future studies.

As an *efficiency* criterion, we considered *Average P-Value*, introduced for the first time by Smith et al. in [14]. This criterion is defined to be the average of the p-values in the grid. We want this parameter to be as small as possible, because it is directly related to the size of our prediction set Γ^ε. This criterion is convenient to use in Conformal Clustering, because we already have a p-values grid, and calculating its average is computationally fast.

5.4 Experiments and analysis of the results

Conformal Clustering experiments were done for objects produced by a fixed set of parameters of t-SNE, namely those who obtained the best Kullbak-Leibler divergence in Section 4.3. The obtained objects $\{z_1, ..., z_n\}$ were

k	APV	Purity
1	**0.082001**	**1.000000**
2	0.088266	0.985075
3	0.095214	0.940299
4	0.101682	0.932836
5	0.107819	0.925373
6	0.117725	0.888060
7	0.124504	0.888060
8	0.129991	0.895522
9	0.136139	0.910448
10	0.141672	0.910448

Table 4: k-NN non-conformity measure for Conformal Clustering, $\varepsilon = 0.2$

used to compute a ℓ sided p-values grid, $\ell = 35$. The p-values of the grid were then predicted by setting "1" when $p_i > \varepsilon$ "0" otherwise, and the "1" points were clustered as explained before. Finally, the clusters generated by the p-values grid, were used to associate a cluster to objects $\{z_1, ..., z_n\}$. A cluster was associated to an object z_i only if one of the objects it contained had a distance of 1 or less to z_i. Then *Purity* and APV were calculated.

The first experiment follows the procedure as above, by keeping fixed the significance level $\varepsilon = 0.2$ and using the two non-conformity measures we described: *k-NN* and *KDE* (with Gaussian kernel). In the case of k-NN we trimmed the parameter k, for KDE we instead considered the bandwidth h. Table 4 and Table 5 show the results of this experiment.

From these results we can see that both non-conformity measures were able to achieve best Purity. However, k-NN non-conformity measure did quite better than KDE in terms of efficiency, here measured by APV. Best results of Purity and APV are in this case on the same row for both the non-conformity measures. In order to obtain the same the reader may, from the directory above `code/` , launch:

```
python -m code.scripts.evaluate_clustering --dataset [preprocessed
 dataset] --labels [labels]
```

where *labels* should be the labels file produced by **create_clusters()**, `data/dataset/dataset-labels.txt` . Please note that this script requires much time to be executed.

For the second experiment we focused on ε, to see how different significance levels may infer with *Purity*. Note that since APV criterion doesn't

h	APV	Purity
0.1	**0.092544**	**1.000000**
0.2	0.093504	0.992537
0.3	0.103475	0.955224
0.4	0.114118	0.925373
0.5	0.124189	0.888060
0.6	0.133151	0.888060
0.7	0.142152	0.902985
0.8	0.151735	0.947761
0.9	0.160588	0.947761
1.0	0.167546	0.910448

Table 5: KDE with Gaussian kernel non-conformity measure for Conformal Clustering, $\varepsilon = 0.2$

change by trimming ε, because it is an ε-independent criterion, we don't report its value here. In this experiment we use the kNN and KDE parameters who achieved best APV in the previous one. From them, respect to different values of ε, we construct the grid, create clusters and predict our objects by using them. Follows a listing showing these steps. The reader may obtain our same results by executing what follows from the directory above `code/` and within a *Python* interactive session:

```
import pylab as pl
import numpy as np
from scipy.spatial.distance import pdist, squareform
from code.scripts import load
from code.conformal_prediction import clustering as cls
from sklearn.manifold import TSNE

# Settings
DATASET = 'data/dataset/dataset-preproc.csv'
# Settings after the experiments
PERPLEXITY = 50
METRIC = 'euclidean'
LEARNINGR = 100
SEED = 2
L = 35
```

```
NCM = 'knn'
NCM_ARGS = {'k': 1}

# Load dataset
x = load.load_dataset(DATASET)
# Conformal clustering
dist = squareform(pdist(x, METRIC))
t = TSNE(perplexity=PERPLEXITY, learning_rate=LEARNINGR,
random_state=SEED,
          metric='precomputed')
z = t.fit_transform(dist)
grid = cls.pvalues_grid(z, L, NCM, **NCM_ARGS)
# Round to nearest value in set [0.05, 0.1, 0.2, ...]
nearest = lambda x,s: s[np.argmin(np.abs(s-x))]
f = np.arange(0, 0.6, 0.1)
grid[:,2] = [nearest(x,f) for x in grid[:,2]]
# Different transparency for each confidence interval.
colors = np.array([(0,0,1,a/max(f)*10) for a in grid[:,2]])
# Plot data with different confidence levels
cm = pl.cm.get_cmap('Reds')
pl.scatter(grid[:,0], grid[:,1], c=grid[:,2], marker='x',
cmap=cm)
pl.colorbar()
pl.show()
```

This code will produce a plot as in Figure 7, where the prediction set for our best scoring k-NN Conformal Clustering was coloured differently with the respect to ε. This allows us to intuitively understand how objects within our t-SNE space will be predicted and clustered for different significance levels.

We can do the same for KDE non-conformity measure, by substituting in the previous listing of code `NCM='knn'` with `NCM='kde'` and `NCM_ARGS={'k': 1}` with `NCM_ARGS={'kernel':'gaussian', 'h': 0.1}`. This brings to Figure 8.

As expected from the previous results in *APV*, KDE shows many points with low p-value all over the map. This makes the prediction set bigger than k-NN one.

So far we have considered all the parts of our system (*preprocessing*, *clustering*) as separate entities. This approach is good for understanding how each parameter influences the results within its context. However, we need

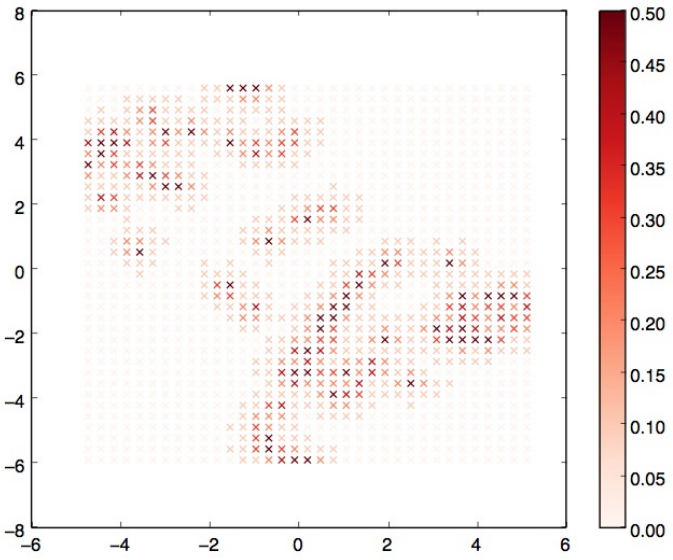

Figure 7: Prediction set for various ε for k-NN k=1

to know how different part integrate in the system, in order to achieve best general performances. Next section investigates this aspect, and introduces a *test* set to evaluate the experiments.

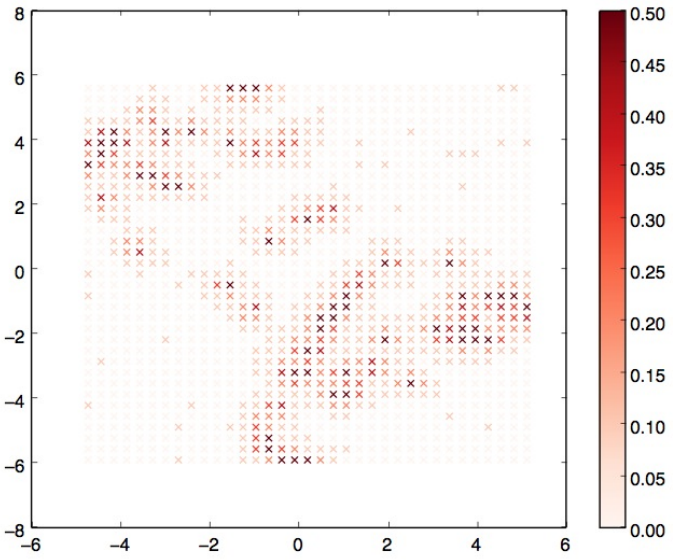

Figure 8: Prediction set for various ε for KDE (gaussian) bandwidth=0.1

6 System Evaluation

Experiments conducted in Section 3, Section 4 and Section 5 had the goal of evaluating only parts of the system, in order to understand how parameters influenced them. However, since previous experiments did not lead us to a general understanding of the system, it seemed necessary to have a global evaluation taking into account how the parameters of the different parts analysed so far influence the final results. Another aspect that was not considered in the previous sections is validation on a *test* set. Using $LOOCV$ in this particular experiment was not feasible in our context, so we opted for simply splitting our dataset into *train* and *test* set as explained later.

6.1 Evaluation Criteria

For evaluating the experiments in this section we decided to consider three of the criteria we analysed in this report. In particular we chose to use *Purity*, *Average P-Value (APV)* and *Kullback-Leibler divergence*. *Purity*, as explained in Section 5, is here used as an accuracy criterion which expresses how many differently labelled objects one cluster contains. APV is

an efficiency measure, which indicates how large is our prediction set, and the smaller it is, the bigger the efficiency is. *Kullback-Leibler divergence* is here considered in order to have an idea of the t-SNE embedding. However, this parameter cannot be used to evaluate the system, since as experiments in Section 4.3 show, by incrementing *perplexity*, KL divergence keeps going toward 0.

6.2 Description of the experiment

The experiments performed in this section will follow these steps:

1. A *dataset* is generated as in Section 3.
2. The *dataset* is preprocessed as in Section 4.1 and Section 4.2, to generate what we will here refer to as *dataset-preproc*.
3. *dataset-preproc* is projected into a 2-D map z by using t-SNE.
4. From z is created a *train_set* with 75% of its objects and a *test_set* with the remaining elements.
5. *train_set* is used to generate a p-values grid, clusters are created from it as in Section 5, by using a certain significance level ε.
6. *test_set* is clustered respect to the clusters generated before as in Section 5.
7. *Purity* is calculated over *test_set* predicted clusters, and APV over the p-values grid created in step 5.

We make notice that we must split between *train* and *test* set only after using t-SNE, and not directly on the dataset. This because we need both the *train* and *test* objects to be described by the same map. Clusters are extracted from p-values grid after applying a significance level $\varepsilon = 0.2$.

Previous experiments on t-SNE suggested that by increasing *perplexity* we get better and better values for KL divergence. In our experiment we analyse the two non-conformity measures k-NN and KDE, by using the parameters which gave best results in the previous section, and then trim *perplexity* to see how it influences the performances.

6.3 Results and Analysis

We first analyse k-NN based Conformal Clustering. Perplexity is considered for values between 5 and 109. Figure 9 shows that, as hypothesised in Section 4, KL divergence keeps decreasing toward 0 as *perplexity grows*. However, from Figure 10 we can see that APV and Purity are not influenced so much by this parameter on the long run. In fact, Purity rapidly increases around *perplexity*= 20, but it then stays on average on 1.0, with some down-peaks. Similarly, Average P-Value has many peaks, and in the long run looks to raise, but it doesn't seem to be completely correlated to it. Since Purity looks to be stable respect to *perplexity*, we choose to look at the best APV. A minimum for APV is in our experiment at 0.0618, when *perplexity*= 48 and *Purity*= 0.9706. These fluctuations in both APV and *Purity*, which are not present in KDE case, as we will show later, suggest that we should run the experiments for many PRNG seed values, for both t-SNE and *train* and *test* set separation, and then average the results. Because of limited time we could not do these experiments, and they may be a good start point for future research. Figure 12 shows t-SNE using *perplexity*= 48, which obtained best for k-NN Conformal Clustering. From this figure we can observe that our four classes are easily separable. However, we also notice that 4 clusters are not enough. Conformal Clustering, in our case, produced around 20 clusters for this experiment. Future research should understand if this is acceptable for this clustering method, or if instead a different *accuracy* or *validity* criterion should be chosen to evaluate it. In fact, as we mentioned before, *Purity* does not take into account the number of clusters produced.

Quite different results were noticed for KDE Conformal Clustering. Here, APV assumes a clear trend which quickly goes down until around 30, and then gradually starts going up after 60. Its behaviour, described in Figure 11, in closer to what we actually expected: after a certain value of *perplexity*, the size of the prediction sets start raising. This phenomenon can be also seen in kNN case, as in Figure 10, but it is not clearly visible, and further experiments with different seeds values should understand this. In terms of Purity, k-NN and KDE behaves similarly. We suggest the peaks may be due to random sampling, but this should be further investigated. The best APV obtained by experiment with k-NN was at *perplexity*= 52, where $APV = 0.0861$ and *Purity*= 1.0000.

In order to run the two experiments, run, from the directory above `code/`:

```
python -m code.scripts.evaluate_system --dataset [preprocessed
```

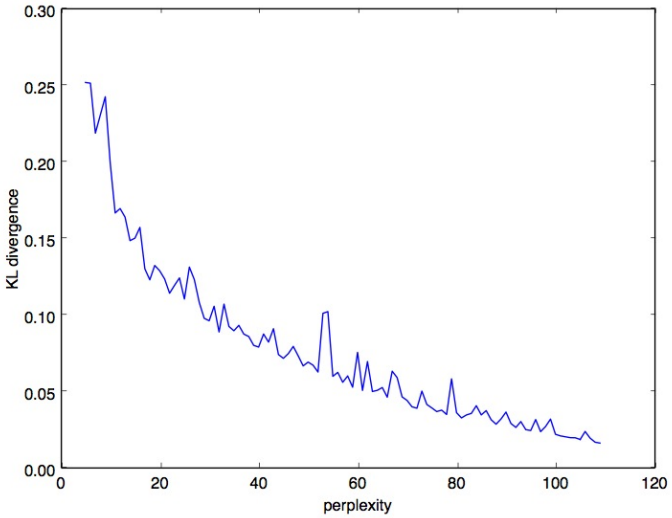

Figure 9: Kullback-Leibler divergence in k-NN Conformal Clustering for many values of *perplexity*

`dataset] --labels [labels] --ncm knn --k 1` ,
to evaluate the system for k-NN non-conformity measure. Here, k is the number of neighbours.

```
python -m code.scripts.evaluate_system --dataset [preprocessed
dataset] --labels [labels] --ncm kde --kernel gaussian --bandwidth
0.1
```

where *kernel* is the KDE kernel, and *bandwidth* is the kernel bandwidth. These two scripts will return an output in 5 rows, indicating: *perplexity*, *significance level*, *KL divergence*, APV, *Purity*.

The experiments indicate that k-NN non-conformity measure performs better for our Conformal Clustering respect to KDE in terms of prediction set size (APV). This result is quite unexpected, because KDE would suggest a function able to smooth the prediction set tight as we want. However, we were able to obtain perfect scores in terms of *Purity* on the test set. This fact shows that our system works well, but also that maybe an *accuracy* criterion considering also the number of clusters should be experimented.

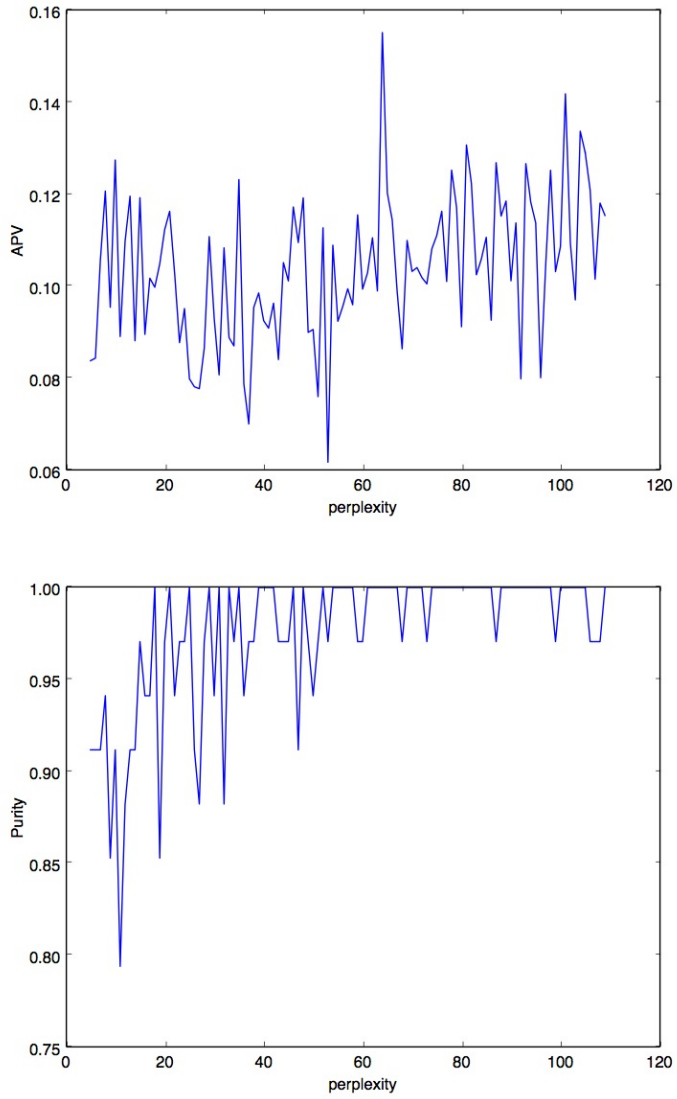

Figure 10: Average P-Value and Purity in k-NN Conformal Clustering for many values of *perplexity*

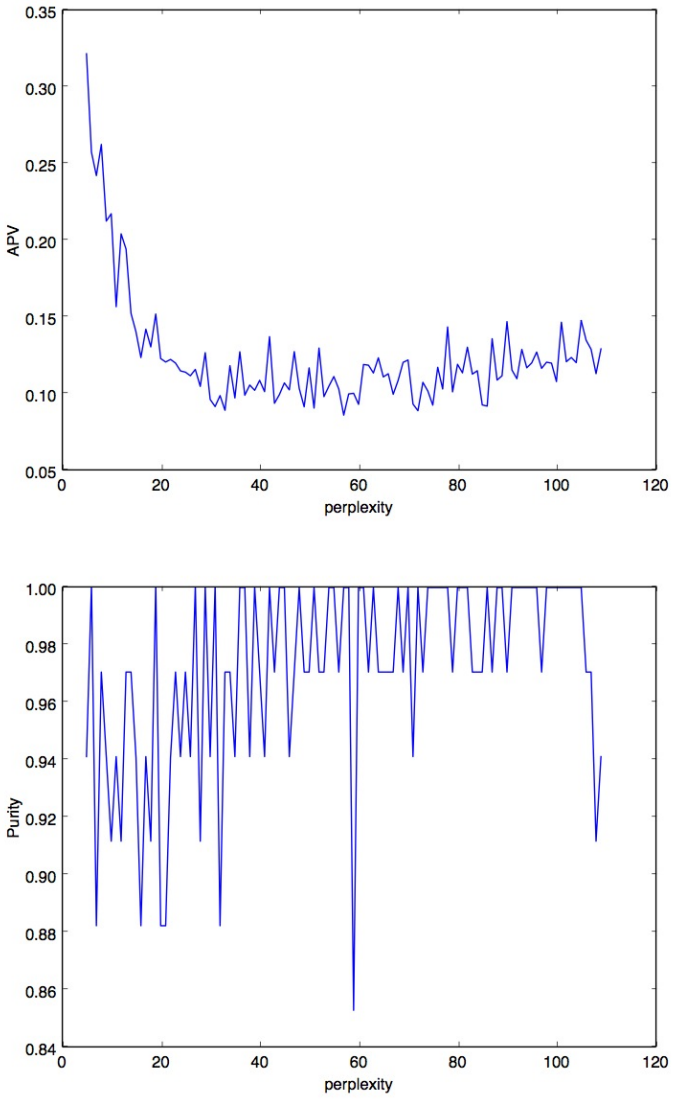

Figure 11: Average P-Value and Purity in KDE Conformal Clustering for many values of *perplexity*

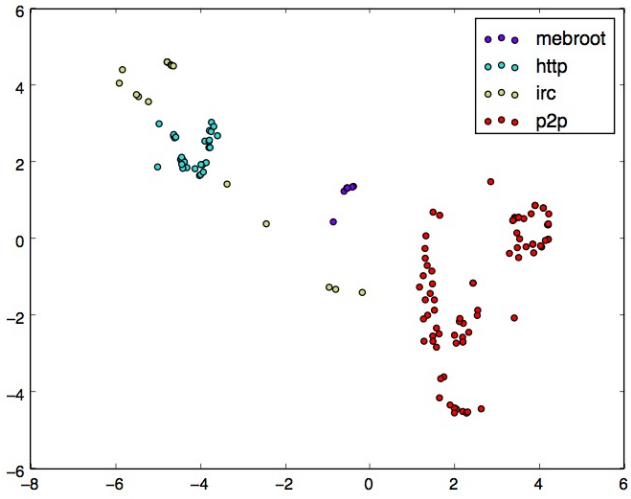

Figure 12: Best t-SNE for our system with labelled classes.

7 Code structure

This Section presents the effort spent in creating a consistent code structure. The code was mainly written in *Python*. The choice of other languages for specific tasks is explained throughout this section. The developed software is a prototype, with the goal of helping research and easily produce interpretable results and evaluations. The code is hierarchically divided into packages, which are presented below. A *Python module* is a file containing *Python* definitions and statements. Modules can be logically grouped into *packages*, which are the collection of modules contained in a directory[8]. In general, a module **xxx** in directory `code/aaa/` will be referred to as **code.aaa.xxx**. Figure 13 shows the hierarchy of our code. Follows an explanation the packages in our code. Note that in our structure **code** is itself a package, which contains the packages: **scripts**, **features**, **preprocessing**, **conformal_prediction**, **evaluate**.

7.1 Package code.scripts

A logical separation can be made between **code.scripts** and all the other modules from **code**. This module contains all the code of the experiments. If a user wants to create a new experiment, he can create a python file in `code/scripts/` folder and insert the code. In order to import the modules within **code** he must (antepone) a ".." to the module name. For example, if the user wants to use the module **code.conformal_prediction.clustering** from a script in **code.scripts**, he will insert:

```
from ..conformal_prediction import clustering
```

and then refer to the module's functions as **clustering.foo()**. In order to execute a script from this directory the user should, as we did previously, run:

```
python -m code.script.[module without .py extension]
```

from the directory above `code/` .

In this report we located all the dataset creation and performance evaluation scripts in `code/scripts` . Module **code.scripts** also contains the module **load**, which exposes the functions required to load a network trace file and store/load a dataset, labels, and class ids in files.

[8] A directory, in order to be considered a package, must also contain, even if empty, the file `__init__.py` , to initialize the package.

```
code/
|-- __init__.py
|-- conformal_prediction
| |-- README
| |-- __init__.py
| |-- clustering.py
| `-- predictor.py
|-- evaluate
| |-- README
| |-- __init__.py
| `-- evaluate.py
|-- features
| |-- README
| |-- __init__.py
| `-- select_features.py
|-- preprocessing
| |-- README
| |-- __init__.py
| `-- preprocessing.py
`-- scripts
    README
    __init__.py
    best_tsne.py
    create_dataset.py
    dataset_blacklist.txt
    evaluate_clustering.py
    evaluate_features.py
    evaluate_system.py
    evaluate_tsne.py
    load.py
    load.pyc
    preprocess_dataset.py
    test-pacf-artificial.R
```

Figure 13: Hierarchy of `code/` directory

7.2 Package code.features

The package **code.features** only contains module **select_features**. This module has all the functions needed to extract features from a vector of data. In our context this module was used by **code.scripts.create_dataset** to elaborate the informations in a network trace file, and to output features for creating the dataset. Appendix will show a peculiarity of the function **get_frequency()** from this package.

7.3 Package code.preprocessing

This package contains the module needed to preprocess our dataset when created. The module **code.preprocessing.preprocessing** contains the functions to normalize a vector in [0,1] and to take the log of every element in a vector. This module is used by **code.scripts.preprocess_dataset** to preprocess the dataset before other analysis.

7.4 Package code.conformal_prediction

In this package functions are logically divided into two modules: **predictor** and **clustering**.

Module **predictor** contains functions which are used for potentially every application of Conformal Prediction. It comprises functions such as the calculation of p-value by using a defined or a custom non-conformity measure, and the prediction of a new example with the respect to a significance level. For how the module is designed it will be easy to extend the module with new non-conformity measures.

The second module, **clustering** contains functions specific to Conformal Clustering, as explained in Section 5. These functions allow to create p-values grids, creating clusters, and predicting clusters for z_i objects.

7.5 Package code.evaluation

This package contains all the functions needed to evaluate our system. Since the evaluation functions were only a few, they where all collected in one module **evaluate**. However it is possible to create a division between these functions in case it will be needed, by separating them in modules under **code.evaluate**. Functions in **code.evaluation** comprise calculation of *SIMM-DIFF* coefficients, as in Section 3, calculation Kullback-Leibler divergence when computing TSNE, and calculation of Average P-Value (APV) and clusters *Purity* as described in Section 5.

8 Future directions, Conclusions

We presented a system for detecting botnet infected computers (bots) within a network, by using high level traffic features. Our goal was to cluster together botnet threats of the same type, with the claim that by doing this we will be able to detect previously known botnets in a real world setting. Respect to previous research [16, 5, 7] we introduced a larger set of meaningful features, a novel algorithm for periodicity detection in botnets communication by using PACF, and conducted many experiments by using t-SNE and Conformal Clustering to generate clusters. We experimented with both *KDE* and *k-NN* non-conformity measures, and quite surprisingly found out that *k-NN* performed better. Since however the results of the two measures are quite similar, we suggest this may be due to approximation errors, and more specific studies are required on this. The method we proposed to evaluate the contribution of features, *SIM-DIFF*, helped us to decide between possible preprocessing choices and to spot errors in creating the dataset. We also suggested *Purity* as an accuracy criterion for clusters, and we were able to obtain perfect accuracy in our experiments, while keeping a good value of efficiency. Efficiency criterion APV was used as in Smith et al. [14].

Further studies may focus on a different criterion for evaluating clustering accuracy or validity. In fact, as explained in this report, *Purity* does not take into account the number of clusters generated, and this may be a problem when evaluating performances. Future studies should then either propose a different accuracy criterion, or show that *Purity* is valid for evaluating Conformal Clustering. A second important aspect would be to perform the experiments by considering every specific kind of botnet as a class itself. Here in fact we grouped botnets in four major classes, but we afterwards noticed that there may be some evident distinctions within themselves (see Figure 12). One set of features which may be really important for our scope is "most frequent destination port", or "k most frequent destination ports". This information can be useful because some botnets make use of a precise set of port numbers [1], but we did not use it. We also did not use features $\{14, 15, ..., 18\}$ of Table 1, and further studies may try to understand if they are useful for our problem.

Finally, further experiments should be done on the system, as suggested by Section 6, to remove the high variability in results, and to have a better idea about how *perplexity* relates to APV and Purity in k-NN based Conformal Clustering. Noise reduction methods may also be experimented on network traces before creating the dataset.

References

[1] Paul Bacher, Thorsten Holz, Markus Kotter, and Georg Wicherski. Know your enemy: Tracking botnets, 2005.

[2] Michael Bailey, Evan Cooke, Farnam Jahanian, Yunjing Xu, and Manish Karir. A survey of botnet technology and defenses. In *Conference For Homeland Security, 2009. CATCH'09. Cybersecurity Applications & Technology*, pages 299–304. IEEE, 2009.

[3] James Cook, Ilya Sutskever, Andriy Mnih, and Geoffrey E Hinton. Visualizing similarity data with a mixture of maps. In *International Conference on Artificial Intelligence and Statistics*, pages 67–74, 2007.

[4] Alexander Gammerman and Vladimir Vovk. Hedging predictions in machine learning the second computer journal lecture. *The Computer Journal*, 50(2):151–163, 2007.

[5] Jan Goebel and Thorsten Holz. Rishi: Identify bot contaminated hosts by irc nickname evaluation. In *Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, pages 8–8. Cambridge, MA, 2007.

[6] Guofei Gu, Roberto Perdisci, Junjie Zhang, Wenke Lee, et al. Botminer: Clustering analysis of network traffic for protocol-and structure-independent botnet detection. In *USENIX Security Symposium*, pages 139–154, 2008.

[7] Guofei Gu, Phillip A Porras, Vinod Yegneswaran, Martin W Fong, and Wenke Lee. Bothunter: Detecting malware infection through ids-driven dialog correlation. In *USENIX Security*, volume 7, pages 1–16, 2007.

[8] Geoffrey E Hinton and Sam T Roweis. Stochastic neighbor embedding. In *Advances in neural information processing systems*, pages 833–840, 2002.

[9] RC Joshi and Anjali Sardana. *Honeypots: A New Paradigm to Information Security*. Science Publishers, 2011.

[10] Rikard Laxhammar and Göran Falkman. Sequential conformal anomaly detection in trajectories based on hausdorff distance. In *Information Fusion (FUSION), 2011 Proceedings of the 14th International Conference on*, pages 1–8. IEEE, 2011.

[11] Jing Lei, Alessandro Rinaldo, and Larry Wasserman. A conformal prediction approach to explore functional data. *Annals of Mathematics and Artificial Intelligence*, pages 1–15, 2013.

[12] Jing Lei and Larry Wasserman. Distribution-free prediction bands for non-parametric regression. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 76(1):71–96, 2014.

[13] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[14] James Smith, Ilia Nouretdinov, Rachel Craddock, Charles Offer, and Alexander Gammerman. Anomaly detection of trajectories with kernel density estimation by conformal prediction. Accepted for COPA'2014.

[15] Robin Sommer and Vern Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 305–316. IEEE, 2010.

[16] Florian Tegeler, Xiaoming Fu, Giovanni Vigna, and Christopher Kruegel. Botfinder: Finding bots in network traffic without deep packet inspection. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 349–360. ACM, 2012.

[17] Toni. Calculating the size of the downadup outbreak. `http://www.f-secure.com/weblog/archives/00001584.html`.

[18] Laurens van der Maaten. t-distributed stochastic neighbor embedding. `http://homepage.tudelft.nl/19j49/t-SNE.html`.

[19] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(2579-2605):85, 2008.

[20] Vladimir Vovk, Alex Gammerman, and Glenn Shafer. *Algorithmic learning in a random world*. Springer, 2005.

A Installing and using the code

The code we produced is all in directory `code/` , which constitutes the **code** *Python* package. **code** depends on the following *Python* packages: *Numpy*, *Scipy*, *Scikit-learn*, *IPy*, *patsy*, *rpy2*, *Matplotlib*. The code was tested on OSX 10.9.4 and Linux openSUSE 13.1 using *CPython* version `2.7`. Follow the instructions to install each of the requirements if needed. To check if some of them is already installed, from a command line:

```
$ python
>>> import numpy
>>> import scipy
>>> import sklearn
>>> import IPy
>>> import patsy
>>> import rpy2
>>> import pylab
>>>
```

Each line will raise an exception only if the package is not installed. After all the requirements are installed, the reader can follow the examples, as introduced in Section 1.4.

NOTE: *Scikit-learn* was modified by the author of this report for the purpose of the project. A `pull request`[1] has been sent by the author, but it has not been merged to the master yet. For this reason it *must* be installed from a branch on the author's fork as below.

pip

It is a useful tool to easily install *Python* packages without the need of visiting package website or manually cloning remote repositories. It can be installed by following the instructions at `http://pip.readthedocs.org/en/latest/installing.html#install-pip`. Following installation instructions will also propose the *pip* command. The flag `--user` we will use in the following *pip* directives indicate the installation to happen locally for a single user.

[1] `https://github.com/scikit-learn/scikit-learn/pull/3422`

numpy, scipy

Can be installed by downloading binaries from `http://www.scipy.org/scipylib/download.html`. Otherwise, by using *pip*, from a command line:

`pip install --user numpy`

and

`pip install --user scipy`

sklearn

Scikit-learn must be installed as follows. From a command line, type:

```
cd some/tmp/directory/
git clone https://github.com/joker0x5F5F/scikit-learn
cd scikit-learn
git checkout tsne-kldivergence
python setup.py install --user --record files.txt
```

IPy

Can be installed by following instructions on the website `https://pypi.python.org/pypi/IPy/`, or by typing from a command line:

`pip install --user IPy`

patsy

As before, it can be installed by following instructions on the website `https://pypi.python.org/pypi/patsy/`, or by typing from a command line:

`pip install --user patsy`

rpy2

Its website (`http://rpy.sourceforge.net/`) suggests to directly use *pip*:

`pip install --user rpy2`

As alternative, `https://pypi.python.org/pypi/rpy2` can be used.

matplotlib

Matplotlib can be installed either by its download page `http://matplotlib.org/downloads.html` or as before by using *pip*:

`pip install --user matplotlib`

B Calling *R* from *Python*

The function **code.featutes.select_features.get_frequency()** was presented extensively in Section 3, when talking about periodicity detection. Its code although presents a peculiarity, which is worth to mention. The algorithm we suggested requires to compute the Partial Autocorrelation Function (PACF). However, while for other functionalities we relied on external *Python* packages (e.g. **sklearn.manifold.TSNE**) or created our implementation where needed (e.g. **code.conformal_prediction**), in this case we decided not to use the most known *Python* implementation of it: **pacf_ols()** from **statsmodels.tsa.stattools**. This choice is due to the fact that the code of this implementation is poor, very slow, and gives unreliable results. Due to timing constraints we decided to rely on some code we knew to work well: the *R*'s function **pacf()**. This function source code is clean and well written, and it uses *C* compiled code when performing the most computationally expensive tasks. This was also an opportunity to learn how to call *R* code from *Python*.

For interfacing *R* from *Python* we opted for **rpy2**[2]. This package provides an easy to use interface to *R*'s functions. Using some *R* code from within a *Python* environment is in fact as simple as:

```
>>> from rpy2.robjects import r as R
>>> # run R's seq()
>>> s = R.seq(1, 10, 1)
>>> # convert it to a python list
>>> s = list(s)
>>> print s
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

where we execute *R*'s **seq()** function, convert the result to *Python* list and print its result. The reason we need to convert the result to a list object is that **rpy2** returns as output objects of custom types. This is not a big issue, as it's easy to cast objects from/to the functions of this module. For example if, as in our specific case, we want our *R* code to accept a **numpy.array** and compute on it **pacf()**, we need to do something like:

```
from rpy2.robjects import r as R, FloatVector
y = range(10)
```

[2] http://rpy.sourceforge.net/

```
pacf = R.pacf(FloatVector(y), plot=False)[0]
pacf = list(pacf)
print pacf
# Output:
[0.7000000000000001, -0.15270350564468238,
-0.15490666705224027,
-0.154749118543953, -0.1491847053787029,
-0.13323879180867937,
-0.09926169022868174, -0.036962247141936716,
0.06477815793402909]
```

In this interactive session we create a vector y of numbers from 0 to 9, cast it to be of type **rpy2.robjects.FloatVector** and pass it to **pacf()** function. The object returned by this function is a **rpy2.robjects.vectors.ListVector**, from which we consider the first entry: the PACF coefficients. These are of type **rpy2.robjects.vectors.Array**, which we may want to convert to list as shown.

In general **rpy2** is a great interface to *R* language. It has many other interesting functionalities, such as importing external library as in *R* environment, but they are not presented in this document. For what concerns this project, the choice of replacing **pacf_ols()** from **statsmodels.tsa.stattools** with **pacf()** from *R* considerably improved the timing performances. In fact this task improved from taking several minutes for computing PACF to just a few seconds. We believe this solution also to be elegant, since it was done in pure *Python*. However, in the future it should be created a stable *Python* library to compute ACF and PACF.

C Code listings

All the working code produced and presented should be provided together with this document. Otherwise, the code can be found on *bitbucket* private repository `https://bitbucket.org/joker__/bdcc`. To get access to this repository, please send an email to:

`g.chers :at: gmail.com`.