UCLouvain

École polytechnique de Louvain

Unsupervised clustering machine learning on packed executable

Author: **Jolan Wathelet**
Supervisor: **Axel Legay**
Readers: **Charles-Henry Bertrand Van Ouytsel, Christophe Crochet, Benoît Macq**
Academic year 2021–2022
Master [120] in Cybersecurity

Abstract

In today's threat landscape, packers are increasingly used by malware creators to hinder static analysis. Using a specialised tool for each packer family to properly analyse them is becoming essential. In order to do this, properly classifying packed executable is important to apply the right unpacking tool. While supervised machine learning get good result with already known packers, they struggle with new packers.

In this report, we test two new features and two unsupervised clustering algorithms, DBSCAN and OPTICS. We start by creating a dataset of packed executable, with its ground truth. Using previous works, we extract fifty-six features and find which combination gives the best classification results. Using this result, we tested our two new features. Our first new feature is the semantic sequence, representing the semantic of the first 100 mnemonics executed by each sample. The semantic of a mnemonic is one of four categories: Data movement, Control, Arithmetic/logic and Other. We also experimented with the length of these mnemonic and semantic sequences. Our second feature was looking for hidden import of 16 API functions, given by Zakeri et al. [28]. And finally, we compared OPTICS with the results from DBSCAN. In our test, none of these new features improved our classification results. And OPTICS given us worse result than DBSCAN, all while being much slower.

Keywords: obfuscation, packing classification, clustering, feature extraction, unsupervised machine learning

Acknowledgement

I would like the express my gratitude to my supervisor, Prof. Axel Legay, for directing me toward this subject and answering my questions. As well as to his assistant, Charles-Henry Bertrand Van Ouytsel, for his continuous help and guidance. I also want to thanks the readers and member of the jury, Prof. Benoît Macq, Charles-Henry Bertrand Van Ouytsel and Christophe Crochet.

More personally, I would also like to express my gratitude to my family and friends for their unabated support during this year. Thanks again to Charles-Henry Bertrand Van Ouytsel, François and Tiffany, who have given me feedback to improve this report.

Table of Contents

List of Figures

List of Tables

List of Code Snippets

List of Abbreviations

AMI	Adjusted Mutual Information
ARI	Adjusted Rand Information
AV	Anti Virus
DBSCAN	Density-based spatial clustering of applications with noise
DIE	Detect It Easy
ESIL	Evaluable Strings Intermediate Language
ML	Machine learning
NMI	Normalized Mutual Information
OPTICS	Ordering points to identify the clustering structure

Chapter 1

Introduction

1.1 Context

Malware are an ever evolving threat, constantly changing their tactics to bypass new protections. To avoid being detected, they use obfuscation techniques. The most widespread obfuscation technique in use today is packing. Using this technique is making static analysis of executable hard, by compressing and/or encrypting the executable into a new one syntactically different from the original one. The best way to analyse a packed executable is dynamically, by executing it to force it to decompress itself.

Tools called unpackers exist to help analyse the packed executable, without having to run it. But such tools cannot always be generalised for all packing techniques, since packers use a wide variety of compression/encryption methods. In order to use specialised unpacker for specific packer families, we need to be able to accurately classify packed executable. Multiple papers worked on this classification problem. Biondi et al [16] worked with supervised machine learning methods, but this created a problem when encountering new packers not in the trained model.

To create a detection and classification method that could evolve over time with new packers, Noureddine et al. [23] worked on a self evolving packer classifier, using the DBSCAN unsupervised clustering algorithm. Using this unsupervised approach, they managed good results.

1.2 Research question

Expanding on the results obtained by Noureddine et al. [23] with an unsupervised clustering algorithm: We will try to improve their results by testing new features and by using the OPTICS clustering algorithm in place of DBSCAN.

1.3 Organisation of this document

This document is structured as follows. In Chapter 2 we describe in details every algorithm, and concept we use in the rest of the document. Chapter 3 look at the existing literature regarding packed executable and their classification using clustering algorithms. Chapter 4 explain how we built our dataset, and the challenge we encountered, and it explains the libraries we used to extract the features necessary for our experiments. Chapter 5 goes through the results of the experiments. Finally, we summarise our finding, give future improvement path and conclude in chapter 6.

Chapter 2

Background

2.1 Packed executable

Packing is a common technique to create smaller executable files, with the added benefit of making the inspection of its code harder. It can be classified in multiple sub-categories, from a simple compressor, to an encryptor, who will encrypt the executable and decrypt it at runtime. It can be used to make reverse engineering harder, both for proprietary code from a genuine organization, or malware using it to hide their identifying characteristics from AV engines.

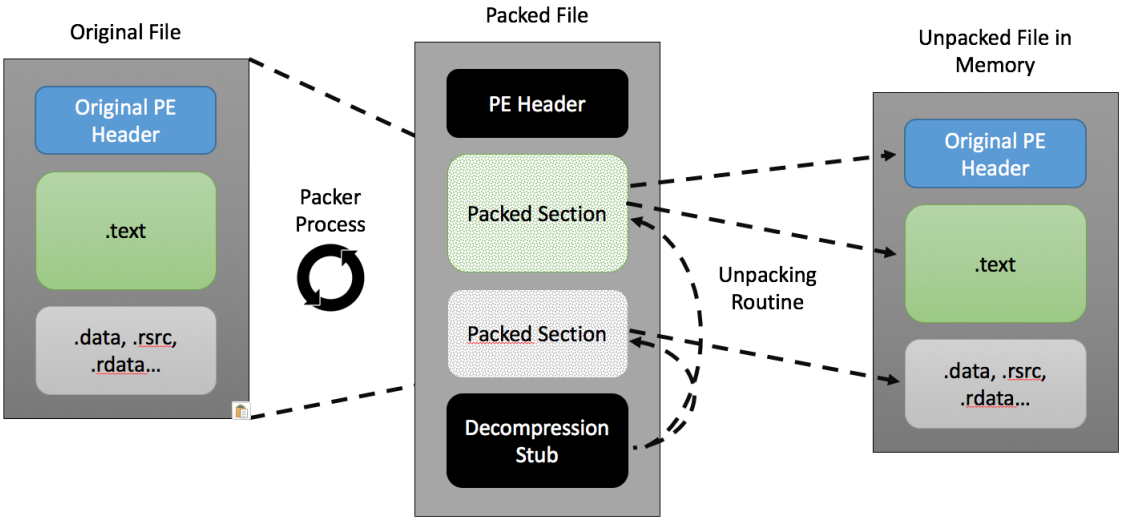

Figure 2.1: Packed executable life-cycle [26]

Figure 2.1 shows the lifecycle of an executable packed with UPX, one of the most widely used packers. This packer uses the UCL [11] algorithm to compress the executable. While not encrypted, the resulting compressed executable is complicated to analyze without using some form of emulation. As you can see the original executable is compressed, then the packer adds a new header and decompression sub, once executed, the packed executable will run the decompression stub, uncompress the original executable, and run it. Packers can use different strategies, Amber [2] allows in memory execution of its payload to avoid detection and Yoda's Protector [13] encrypt its payload.

2.2 Clustering algorithms

Clustering algorithms, as seen in figure 2.2, are an unsupervised form of machine learning, where the algorithm is given the samples as points in space, with a distance value between each other. The algorithm will then create clusters out of these points. We can then

compare the resulting clusters with the ground truth to evaluate the parameters we used.

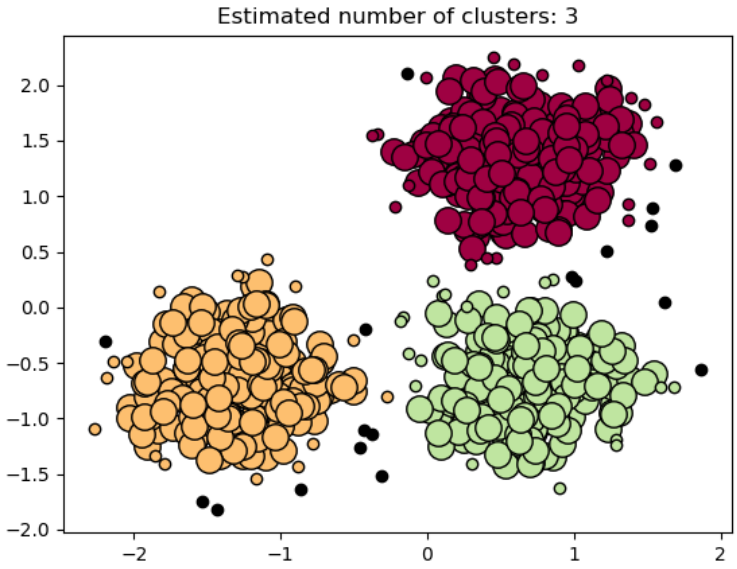

Figure 2.2: DBSCAN Clustering [10]

2.2.1 DBSCAN

Introduction

Density-based spatial clustering of applications with noise (DBSCAN) [19] is a density-based clustering non-parametric algorithm. Given a set of points in space, it creates clusters of points based on their density, and marks those alone in low density regions as outliers. It is called non-parametric because it does not make assumptions about the mapping function, when using DBSCAN to group data together, you are not expecting your cluster to take a given form, you are just letting the algorithm do its work and from there you will adjust the necessary parameter, to improve the result.

Inner working

DBSCAN define three sets of points: core, border and outlier; and uses two parameters, the distance radius ε and the minimum number of points $MinPts$. A core point is a point that has a minimum of $MinPts$ points in the ε radius around it. A border point is a point that is in the ε radius of a core point, but itself doesn't have the $MinPts$ in its ε radius to be a core point. An outlier is a point not in ε radius of any core point.

The algorithm steps are :

- We pick a point that hasn't been visited before.
- If the point is a border point, we pass.
- If it's a core point, we create a new cluster, with itself and the points in its ε radius.
- We then recursively go through each of these points. If these are also core points, we expand the cluster with the points they can reach in their ε radius, and also go through them. Once all the recursive steps have been done, we have a complete cluster.

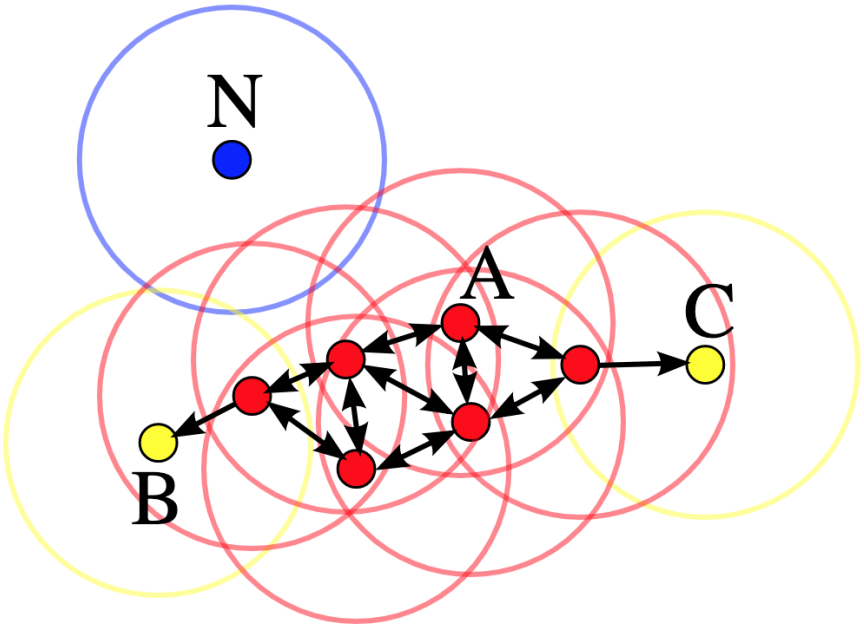

Figure 2.3: DBSCAN points types [17]
Cores points in red, borders in yellow and outliers in blue.

- Once we have gone through all the points, they are only points in clusters, in high-density areas, and points without clusters, outliers, in low-density areas.

The choice of ε and $MinPts$ values is essential to create good clusters. To determine what value to use, a good start is to use the density value of the cluster with the lowest density, since it will allow its creation, and any higher density cluster. From there it's mainly based on the experience we gather during our experiments, as there are no perfect values for all use cases.

One problem with DBSCAN is that the density is the same for all the clusters. It's defined by ε and $MinPts$, should the data you are working with be of varying density, it will be hard to get good results with it. This is why we will also use OPTICS, which can create clusters of varying density.

A second smaller problem is that DBSCAN is a nondeterministic algorithm. If you take two clusters, with a border point between them, in the ε radius of a core point from each cluster, the first cluster to be created will claim this border point. Since the original paper [19] does not give any instruction on how to order the points before looping through them, this leaves the possibility of slightly different clusters depending on the dataset order. In other words, DBSCAN is deterministic on core and noise points, and nondeterministic on border points.

2.2.2 OPTICS

Introduction

Ordering points to identify the clustering structure (OPTICS) [14] is defined, by its creators [14], as "an extended DBSCAN algorithm for an infinite number of distance parameters εi, which are smaller than a "generating distance" $\varepsilon(i.e.0 \leq \varepsilon i \leq \varepsilon)$". In other words, it allows for the creation of clusters of varying density.

Inner working

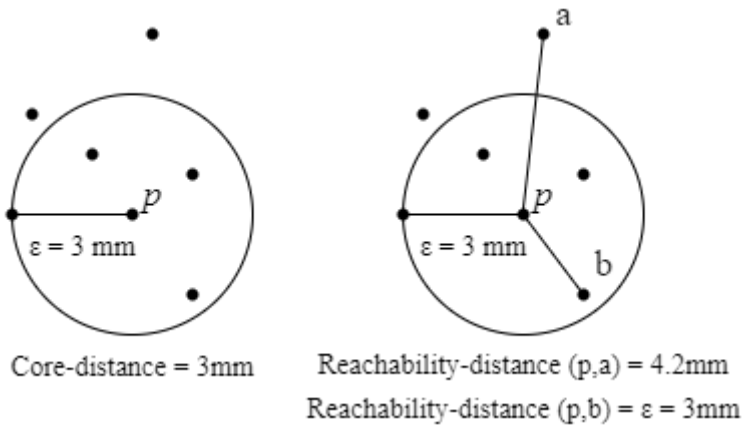

Figure 2.4: Reachability

Like DBSCAN, OPTICS use the parameter $MinPts$, but the parameter ε is optional this time. The first step in OPTICS is to calculate two new values for every point of a given dataset, the core and reachability distance: The core distance (symbol ε in figure 2.4) is the minimum value of the radius required to classify a given point as a core point, with regard to the parameter $MinPts$. If a ε parameter is provided, it will act as a maximum value for the core value, outside of which the point won't be considered a core point. The reachability distance is the distance between a point b and p, if the point p is a core point. If the it is less than the core distance of p, then the value of the reachability distance of b with respect to p, is the core distance of p. if p is not a core point, the reachability distance of b with respect to p is undefined (figure 2.4). To understand the next step, it is easier to explain what OPTICS does in its main loop :

- For each point p, set the reachability to undefined.
- For each unprocessed point we do the following :
  - We get the $neighbors$ of the point p.
  - We mark p as processed.
  - We push p into the ordered List.
  - If p is a core point :
    - We initialise a priority queue, which will be filled by the function update with p $neighbors$ with regard to their reachability distance.
    - For each point in the priority queue, we get their $neighbors$, we mark them as processed, we add them to the ordered list and finally apply the function update on them too.
- When finished, the algorithm will have expend until none of the unprocessed points have a core distance, those are the outlier.

Based on the OPTICS pseudo code A.1.

In the end, we are left with an ordered list, which, when plotted with regard to each

point's reachability distance, gives us the figure 2.5.

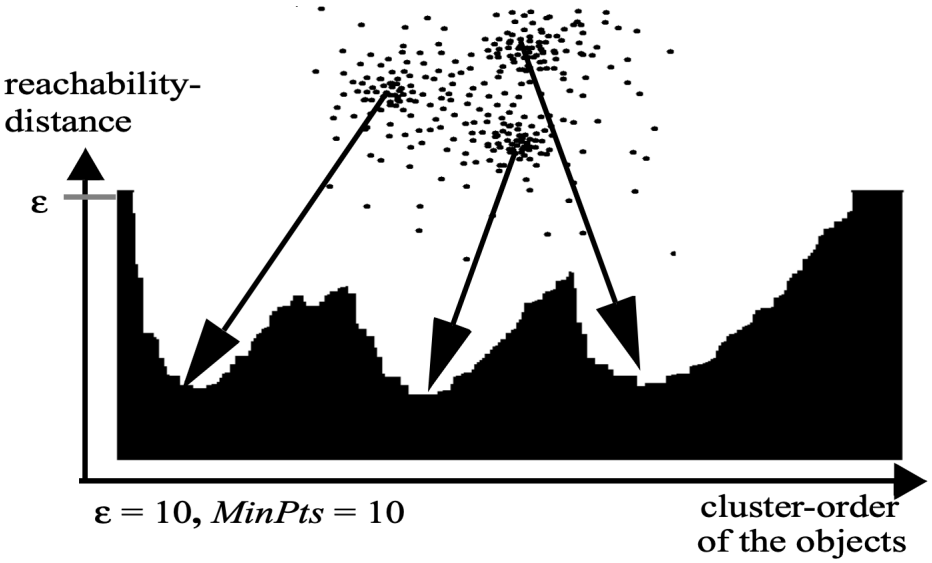

Figure 2.5: Reachability plot

In the reachability plot, the valleys represent the clusters, and the peaks represent the outliers. To extract the cluster from the reachability plot, we can set a threshold on the y axis, but this is equivalent to DBSCAN, or we can use a different algorithm to extract the clusters present in the plot. In stickitlearn [10], we use the default parameter XI, who use the steepness of the valleys in the plot to determine the clusters boundaries, and extract them.

2.3 Distance matrix algorithm

The DBSCAN and OPTICS implementations in Sklearn can both take our dataset as parameter and create a distance matrix by them-self, but since we have unusual features, like the mnemonic sequence, we decided to pre-compute the distance matrix ourselves. From the large array of algorithms to choose from, we decided to stick with the same choice Noureddine and al. [23] made.

2.3.1 Normalized Manhattan distance

Also known as city-block distance, the Manhattan distance is used for the numeric features of our dataset. If we take a numeric feature f of two sample of indices i and j, their distance is the absolute difference between observation $x_i^{(f)}$ and $x_j^{(f)}$:

$$Manh_{ij}^{(f)} = |x_i^{(f)} - x_j^{(f)}|$$

We want to work with the normalized version, so we divide by the absolute maximum range $R^{(f)}$ observed for f among all samples:

$$NormManh_{ij}^{(f)} = \frac{|x_i^{(f)} - x_j^{(f)}|}{|R^{(f)}|}$$

Implementation

In our code implementation we needed to take into account the possibility of a missing value for a feature, for one or multiple samples. We decided to just ignore this case, one could make the point that when comparing two samples, we should increase the distance if one of the samples is missing a value, but then the question of how much this distance should be increased arise. We ignore this problem by just passing this feature without altering the distance.

2.3.2 Normalized Tapered Levenshtein distance

The Normalized Tapered Levenshtein distance is used to calculate the distance between two mnemonic sequences. We start with the Levenshtein distance, who will compare two sequences S_1 and S_2 and simply add 1 to the value of the distance for each position in the sequence where S_1 and S_2 have the same value:

$$S_1 = \{push, mov, push, push, push, mov, push, mov, sub\}$$

$$S_2 = \{push, mov, mov, push, add, mov, add, mov, sub\}$$

The Levenshtein distance of S_1 and S_2 is 6. The tapered version decreases the weight of each value as they appear later in the sequence, giving more importance to the start of the sequence, where the unpacking logic is the most probable to be placed. The Normalized Tapered Levenshtein distance is defined with this equation:

$$NormTapLev_{ij}^{(f)} = \frac{\Sigma_{k=0}^{L_{(S_i,S_j)}^{(f)}-1} W_{(S_i,S_j)}^{(f)}(k)(1 - \frac{k}{L_{(S_i,S_j)}^{(f)}})}{C} \tag{2.1}$$

Where S_i and S_j are the mnemonic sequence compared, $L_{(S_i,S_j)}$ is the maximum length between S_i and S_j. $W_{(S_i,S_j)}^{(f)}(k)$ is worth 1 if the mnemonic at position k in S_i and S_j is the same, or 0 if not. Finally, C is the maximum length of the extraction, to normalise the distance into $[0, 1]$. If we take back the value S_1 and S_2 from the previous example, and give a value C of 50, meaning that we tried to extract 50 mnemonics, it gives us the following result:

$$NormTapLev_{ij}^{(f)} = 0+0+(1-(2/9))+0+(1-(4/9))+0+(1-(6/9))+0+0 = 1.67/50 = 0.033 \tag{2.2}$$

2.3.3 Composite distance

To get our final distance, we combine the different types of distances together. Both by just adding them together and dividing by the number of group combined, and by weighting them according the number of features in each group. In chapter 5 we display the best result of the two. We can see this composite distance as the same idea as the Gower [21] distance, used to compute distance with different types of features.

2.4 Metrics

There are a lot of available metrics to evaluate our result. We decided to settle on three: Homogeneity, Completeness and Adjusted Mutual Information.

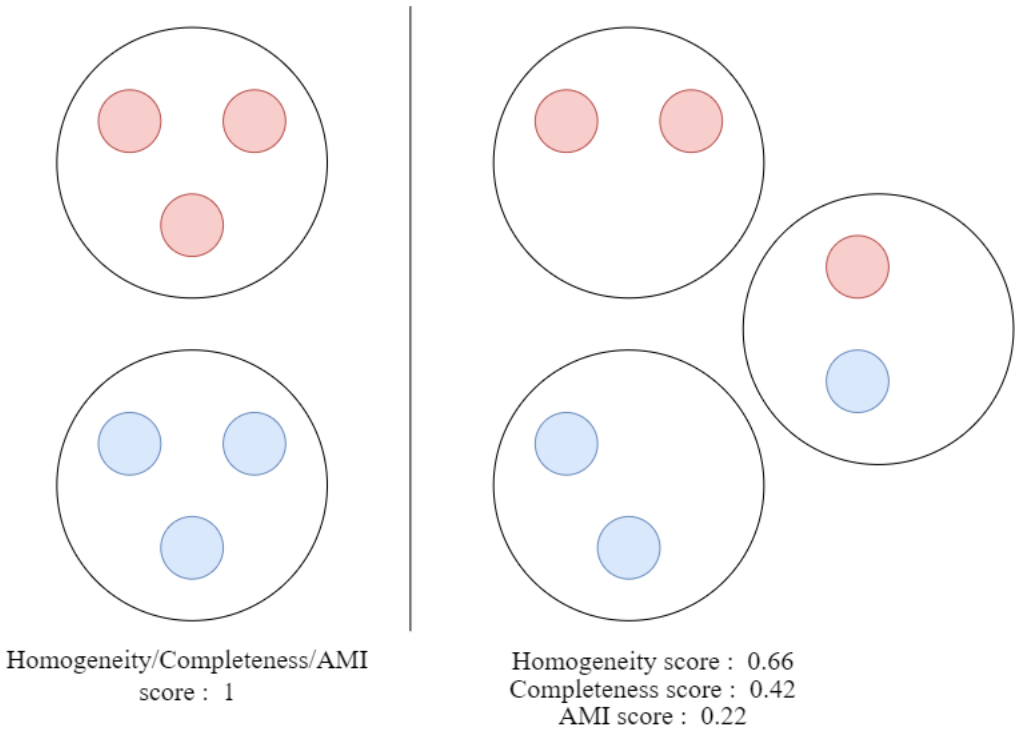

Figure 2.6: Example of the metrics we use

2.4.1 Homogeneity

When each cluster contains only members of a single class, we have a perfect homogeneity score of 1. The formal equation is :

$$h = 1 - \frac{H(C|K)}{H(C)} \tag{2.3}$$

Where $H(C|K)$ is the conditional entropy of the classes given the cluster assignments and $H(C)$ is the entropy of the classes [25].

2.4.2 Completeness

When all members of a given class are assigned to the same cluster, we have a perfect completeness score of 1. The formal equation is :

$$c = 1 - \frac{H(K|C)}{H(K)} \tag{2.4}$$

Where $H(K|C)$ is the conditional entropy of clusters given class and $H(K)$ is the entropy of clusters [25].

2.4.3 Adjusted Mutual Information Score (AMI)

The Mutual Information measures the agreement of the ground truth and the clustering result, ignoring permutations. We use one of the two normalized version of this metric, the Adjusted Mutual Information (AMI), it as the benefit of adjusting the score for chance [27], while the Normalized Mutual Information (NMI), does not. The formal equation is :

$$\text{AMI} = \frac{\text{MI} - E[\text{MI}]}{\text{mean}(H(U), H(V)) - E[\text{MI}]} \tag{2.5}$$

Where $E[\text{MI}]$ is the expected value of the mutual information (MI) and $H(U) and H(V)$ are the entropy of the labels set U and V.

Another metric we could have used instead of AMI is Adjusted Rand Information (ARI). While both choices are equally good, the literature [24] shows that AMI favors clusters of different sizes, while ARI favors clusters of equal sizes. As you will see in section 4.1, our dataset is composed of both small and big clusters, thus we decided to use AMI.

Chapter 3

State of the art

In this chapter we look at the works that have already been published on, and around the subject. Since our research question is relatively specialized, this paper can be considered as a direct follow-up on some of the questions they leave open.

Munkhbayar and al [15] used the entropy values of the unpacking process to detect and classify packers. To do that they used symbolic aggregate approximation (SAX) to change the entropy values into symbolic representation, allowing a reduction of complexity while maintaining accuracy. Using a dataset of 324 benign packed executables, and 326 packed malware, they tested the dataset using two classification methods. The first one is similarity classification methods, such as Cosine, Dice, Jaccard and fidelity. The second is supervised learning classification methods like naive Bayes and support vector machines. They got a 95% to 99% classification accuracy using the fidelity classification method.

Biondi et al [16] focused on the efficiency needed to detect and classify packers in a malware analysis workflow. Their solution is both effective, with a high true positive and a low-false positive rate, efficient, with a low computational cost, and robust, meaning it keeps its effectiveness with a different dataset than the one used to build it. To do that they built a dataset of 280,000 samples, labeling them using three detection tools. They created two ground truths, the first one, called 3cons, is the ground truth with the files whose packer is agreed on by all three detection tools, and 1cons ground truth with the files detected as packer by at least one of the three detection tools (including those already in 3cons). They then extracted a selection of 119 features, grouped into six categories, and applied features selections to only keep those useful in the decision process and with low extraction cost. The six categories of features are Byte Entropy, Entry Bytes, Import Functions, Metadata, Resource and Section. Using three machine learning algorithms, Naive Bayes, Decision Tree and Random Forests, they assessed the performance and usefulness of their dataset. After fine tuning the algorithms parameters, they tested every possible combination of features categories and analyzed their usefulness in detection and classification. They also tested an economical analysis, to figure out which one where optimal regarding the training time in relation to the result obtained. They got multiple conclusions from their experiments. In supervised ML, the size of the dataset is more relevant than its quality to achieve good results. This is due to the fact that malware creators often create custom packers, creating a difference between the result obtained in the lab and in real-life situations, because the trained models have a hard time identifying new packers. They also found that reducing the effectiveness of the algorithms by 1-2% could increase the efficiency by 17-42 times. Finally, looking at the uptime/training ratio, they find that simple algorithms with smaller feature set make more economical sense than more complex algorithms with bigger feature set.

Noureddine et al. [23] looked at the need to move past signature based detection for packed executable. This is needed to counter the increase in the creation of custom made

packers, making their detection by classic means harder. They worked toward creating a self-evolving packer classifier using a composite pairwise distance metric, combining different types of packet features, and the DBSCAN clustering algorithm. Their approach was to use an incremental clustering approach, to create a system able to identify both known and new packers, and to update the existing clusters automatically and efficiently. To test their system they created two datasets. The first one is composed of $16k$ samples from real-world packed malware, with 29 different packers. The second one is composed of $19k$ manually packed executables, with 31 different packers including custom ones. Their conclusion was that the proposed system was effective and reliable over time, since it's constantly able to adapt itself to new packers.

During the writing of this master thesis, Doctor Noureddine Lamine, finished his PhD thesis [22]. His work is the combination of the two last research papers we presented, on which he worked. While the conclusion didn't change, the longer format of a PhD thesis makes for a more in depth reading, where he goes deeper into each part of his solutions.

Chapter 4

Set-up

4.1 Dataset creation

Like all machine learning experiments, our first step was to create a dataset to run our code against. We decided to use a mix of packers found in the wild and executable we packed ourselves.

4.1.1 Packer detection

Packer name	Number of sample	Packer name	Number of sample
.NETZ	3	PECompact	165
ASPack	837	PEtite	198
BeRoEXEPacker	131	PKLITE32	1
DxPack	7	Packman	130
EXpressor	130	PyInstaller	265
Enigma Virtual Box	130	RLPack	130
Exe32pack	130	SerGreen Appacker	2
FSG	135	Spoon Studio	2
JDPack	130	TELock	130
MEW	133	Themida	130
MPRESS	506	UPX	5340
Molebox	131	WinUpack	128
NSPack	130	Yoda-Crypter	130
Neolite	122	Yoda-Protector	130
NsPacK	11	kkrunchy	25
		Total	9572

Table 4.2: Dataset from abuse.ch

We first decided to start our dataset by detecting packed executable in the abuse.ch malware database [1]. This database contains live malware samples added by the community and AV vendors. The fact that these samples are live is important for our experiment, since we intend to dynamically emulate the unpacking process to test new features. The abuse database [1] as over $500k$ samples, but offers only a limited option of labels, themselves of varying quality. We thus needed to scan the sample ourselves to create our dataset and ground truth label. To get those labels we intended to use the same idea as Noureddine et al. [23], building a ground truth using multiple detection tools. But the instability of some of these tools, and compute-time necessary to build the dataset like this, meant that it was out of reach for the scope of this work.

We resolved to use only one detection tool to build our dataset. From all the choices

available freely, we settled on Detect It Easy (DIE). While the perfect detection tool, one that could detect all packed executable without false positive, doesn't exist for now, DIE has three important points making it a good choice for us: It's widely used, up to date and you can use it through command line instruction. Most other detection tools are either too unstable, slow, or hard to incorporate into another workflow. Using a single detection tool brings an obvious bias problem. All bias in DIE will make it into the dataset ground truth, this is why we also add other sources of samples to our dataset.

Using DIE we got 30 different packers family and 9572 samples as you can see in table 4.2.

4.1.2 Packed executable creation

Packer name	**Number of sample**
Amber	9522
Eronana	9785
MPRESS	4352
PECompact	3401
PEtite	5605
Total	32665

Table 4.3: Generated dataset

To create our own packed executable, we used executable from abuse.ch that were not packed, and used 5 packers on them, Amber [2], Eronana [4], MPRESS [5], PECompact [6] and PEtite [8].

this gave us an additional $32k$ sample as you can see in figure 4.3

We then also added a dataset found on Github [18], this dataset contain 23 packer family, and $3k$ samples (figure 4.5).

Packer name	**Number of sample**	**Packer name**	**Number of sample**
ASPack	130	NSPack	130
Amber	130	Neolite	122
BeRoEXEPacker	130	PECompact	130
EXpressor	130	Packman	130
Enigma Virtual Box	130	RLPack	130
Eronana	130	TELock	130
Exe32pack	130	Themida	130
FSG	135	UPX	130
JDPack	130	WinUpack	128
MEW	133	Yoda-Crypter	130
MPRESS	130	Yoda-Protector	130
Molebox	130	Total	2988

Table 4.5: Github dataset

The resulting dataset was too big for our algorithms to process in a reasonable time, and the size of the different packers families where varying widely. To resolve both those

problems, we removed the families with too few samples, and reduce the maximum number of samples per family to $3k$, in order to avoid a over representation of some like UPX, in our dataset. You can see the final dataset we used in our experiment in figure 4.6

Packer name	**Number of sample**	**Packer name**	**Number of sample**
ASPack	967	Neolite	122
Amber	3000	PECompact	3000
BeRoEXEPacker	130	PEtite	3000
EXpressor	130	Packman	130
Enigma Virtual Box	130	PyInstaller	265
Eronana	3000	RLPack	130
Exe32pack	130	TELock	130
FSG	135	Themida	130
JDPack	130	UPX	3000
MEW	133	WinUpack	128
MPRESS	3000	Yoda-Crypter	130
Molebox	130	Yoda-Protector	130
NSPack	130	kkrunchy	25
		Total	21465

Table 4.6: Final dataset

4.2 Feature extraction

We choose our starting set of features according to the state of the art [23]. We have six groups of features: metadata (21 features), sections (21 features), byte entropy (6 features), resources (2 features), import functions (5 features), un-packing stub mnemonic sequence (1 feature). Making for a total of fifty-six features. The definition of each of these features can be found in the table B.

The first 55 features are extracted using the PEfile [7] library.

Mnemonics sequence extraction

The last feature, the unpacking mnemonic sequence, is more complex to extract since we need to get the unpacking code mnemonics dynamically. Simply taking the first operations after the entry point would be useless, since a simple jump operation in the second operation would mean that we are not extracting the unpacking logic, but potentially some random code. To extract valuable mnemonics, and not simply the *nth* mnemonic after the entry point, we need to use dynamic emulation.

Dynamic emulation would mean running the program, which is resource intensive and would be very time consuming given the size of the dataset, so we decided to settle on the next best thing, using "imprecise full emulation" [9].

To do this we used Radar2's Evaluable Strings Intermediate Language (ESIL). ESIL is both an intermediate language to represent the opcode semantic of a given CPU architecture, and an interpreter to emulate this intermediate language.

After opening the packed executable in radare2, We use the commands shown in

```
1       mov ecx, ebx -> ebx,ecx,=
```

Code snippet 1: ASM to ESIL

table 4.7 to extract the mnemonics.

radare2 Command	**Definition**
aaa	Analyse fuction
e asm.emu = true	Run ESIL emulation analysis on disasm
e emu.str = true	Show only strings if any in the asm.emu output
e emu.write = true	Allow asm.emu to modify memory (WARNING)
e io.cache = true	Change both of io.cache.read,write
e dbg.trace = true	Trace program execution (see asm.trace) (Show execution traces for each opcode)
aei	initialize ESIL VM state
aeim	initialize ESIL VM stack
aeip	initialize ESIL program counter to curseek
101aes	perform emulated debugger step (100 times)
dtd	List all traced disassembled

Table 4.7: Mnemonic sequence

```
1       0x00678a91 sub esp, -0x80
2       0x006788e0 pushal
3       0x006788e1 mov esi, 0x5a5000
4       0x006788e6 lea edi, [esi - 0x1a4000]
5       0x006788ec push edi
6       0x006788ed or ebp, 0xffffffff
7       0x006788f0 jmp 0x678902
8       0x00678902 mov ebx, dword ptr [esi]
9       0x00678904 sub esi, -4
10      0x00678907 adc ebx, ebx
11      0x00678909 jb 0x6788f8
12      0x006788f8 mov al, byte ptr [esi]
13      0x006788fa inc esi
14      0x006788fb mov byte ptr [edi], al
15      0x006788fd inc edi
```

Code snippet 2: ESIL emulation example

Ultimately we get a result like shown in the code snippet 2, in which we can see a jump operation in line 7. We only keep the address and the mnemonic from this result. The address will be useful for the semantic feature in the next section.

Semantic sequence

Up until now we have directly compared the extracted mnemonics, but one clear problem with that is that we compare them directly, without taking their meaning into account. There are around thirty different mnemonics with the meaning “Jump if condition is met”,

so one could argue that two of these mnemonics should be considered equal in our distance calculation.

So the first new feature we want to test is changing our mnemonic sequence into a sequence of "categories" of instruction. To define these categories, we used the University of Virginia cs216 X86 Assembly Guide [20], which divide instructions into three categories:

- Data movement: Mov, Push, Pop, etc.
- Arithmetic/logic : Add, Sub, And, Or, etc.
- Control-flow : Jmp, Call, Cmp, etc.

We also add the category "Other" for instructions like unk, nop and null.

Given that x86 has thousands of different instructions, translating the mnemonic sequence into a semantic sequence, by categorizing each instruction, would be a daunting task. thankfully, Radare2 has the beginning of a solution. By using the command *s* to point to a specific address, and then using the command *ao*, we get the information displayed in the code snippet 3.

```
address: 0x678907
opcode: adc ebx, ebx
esilcost: 0
disasm: adc ebx, ebx
pseudo: ebx += ebx
mnemonic: adc
description: add with carry
mask: ffff
prefix: 0
id: 6
bytes: 11db
refptr: 0
size: 2
sign: false
type: add
cycles: 1
esil: cf,ebx,+,ebx,+=,31,$o,of,:=,31,$s,sf,:=,$z,zf,:=,31,$c,cf,:=,$p,pf,:=,3,$c,af,:=
family: cpu
```

Code snippet 3: ao command output

As we can see, it's very complete, and seems to give us a form of categorisation with the "type" value, where the mnemonic *adc* is given the broader type *add*. To know what kind of types are defined by Radare2, we need to dive into the code, because the documentation does not have the information. We can find the type values in the file *radare2/libr/anal/op.c*, defined as the table *optypes* (code snippet 6).

Since Radare2 use Capstone [3] as its default disassembler, this type value is coming from capstone in some way. And indeed, in the function *anop* from the file *radare2/libr/anal/p/anal_x86_cs.c*, we can see Radare2 changing the mnemonic value it gets from Capstone into the much much smaller subset of types we can use.

We then classify the type defined by Radare2 into the 4 categories we explained before as displayed in the code snippet 4.

```
CONST_DATA_MOVE_OPCODES = ["mov","cmov","lea","pop","push","upush","rpush","load",
    "xchg","cast","store"]
CONST_LOGIC_OPCODES = ["add","sub","or","xor","div","mul","and","not","shl","shr",
    "crypto","cpl","sal","sar","rol","ror","mod"]
CONST_CONTROL_FLOW_OPCODES = ["acmp","cjmp","mjmp","cmp","jmp","ujmp","ucjmp","rjmp",
    "ijmp","irjmp","call","ccall","ucall","rcall","icall","ircall","uccall","ret",
    "cswi","swi","rep","ill","cret","switch","io","trap","new"]
CONST_OTHER_OPCODES = ["nop","unk","null","case","leave","sync"]
```

Code snippet 4: Semantic types

In the end we have the mnemonic, the type from Radare2 and the Semantic categories, as you can see in table 4.8.

Mnemonic	Radare2 Type	Semantic
sub	sub	logic
pushal	upush	data
mov	mov	data
lea	lea	data
push	rpush	data
or	or	logic
jmp	jmp	ctrl
mov	mov	data
sub	sub	logic
adc	add	logic
jb	cjmp	ctrl
mov	mov	data

Table 4.8: Mnemonic to semantic

Hidden import

If we look at feature number 51 in the original feature set (table B). The list of 16 malicious API functions come from Zakeri et al. [28]. The value is taken from the import table, without taking into account that a malicious actor could import API in some other way. Making use of Radare2 possibilities, we want to check if the packed executable isn't importing some of these malicious API without using the imports table.

Using the command *izz*, we can search for strings in the Whole binary. By comparing with the string in the import table, we can easily get the number of malicious API import hidden.

```
[0x006788e0]> izz~GetProcAddress
21906 0x000dd80e 0x0068280e 14  15   .rsrc   ascii   GetProcAddress
```

Code snippet 5: izz example

Chapter 5

Experiments

Experiments were done on a 24 core machine using 32go of ram, Running a Debian linux. We used the following libraries :

- SQLite3 to store the dataset. While the idea of having an SQL table with at least fifty-six columns is, by our own admission, barbaric, the ease of transfer and use of an SQLite database far outweighs the few unorthodox inserts needed in our code.
- Pandas to manipulate the data. The distance matrix uses pandas dataframes, allowing us to easily combine them.
- Scipy to help with the creation of the distance matrix.
- Radare2 to extract the features

5.1 Features selection and testing for DBSCAN

The results displayed in this section depend on two parameters, $MinPts$ and ε. In order to show them in the most understandable way possible, they are divided in two graphs. On the left, we will show the value of $MinPts$ on the X axis and the score on the Y axis, with the ε fixed to the value that gave the best AMI score. On the right is the opposite, with ε on the X axis and $MinPts$ fixed to the best value. You can find the number of clusters for each figure in appendix C.1.

5.1.1 Feature selection

To get a general idea on how each group of features stands on its own, we ran each of them with a range of values for ε and $MinPts$, to find the best possible AMI score.

The import and mnemonic sequence features group didn't achieve any results on their own. The resource feature group stayed at a constant AMI of around 0.06, which is insignificant. Since its score was static, it gave us an example of the nondeterministic nature of DBSCAN, with slightly different AMI, Homogeneity and Completeness scores for the same parameters. Their AMI score has a difference of 0.002.

The metadata features, in figure 5.1, give a promising score of 0.574, with a $MinPts$ of 4 and a very small ε of 4×10^{-4}.

The section features, in figure 5.2, give the best score at 0.8, with a $MinPts$ of 23 and a ε of 0.23.

The entropy features, in figure 5.3, also give a promising score at 0.8, with a $MinPts$ of 30 and a ε of 0.04.

A first observation we can make is that $MinPts$ does not seems to influence the AMI score much, compared to ε.

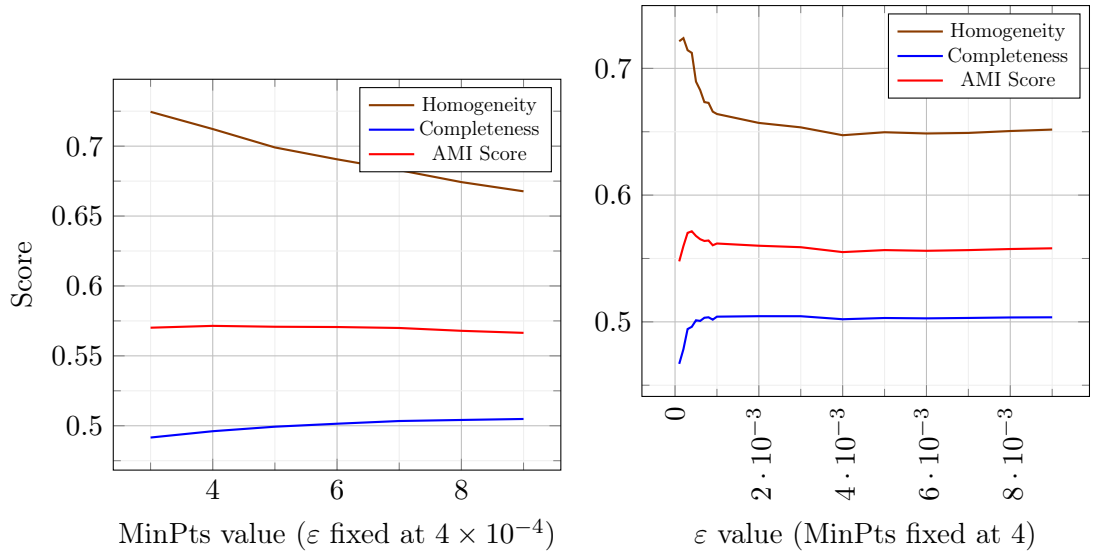

Figure 5.1: DBSCAN Metadata feature group

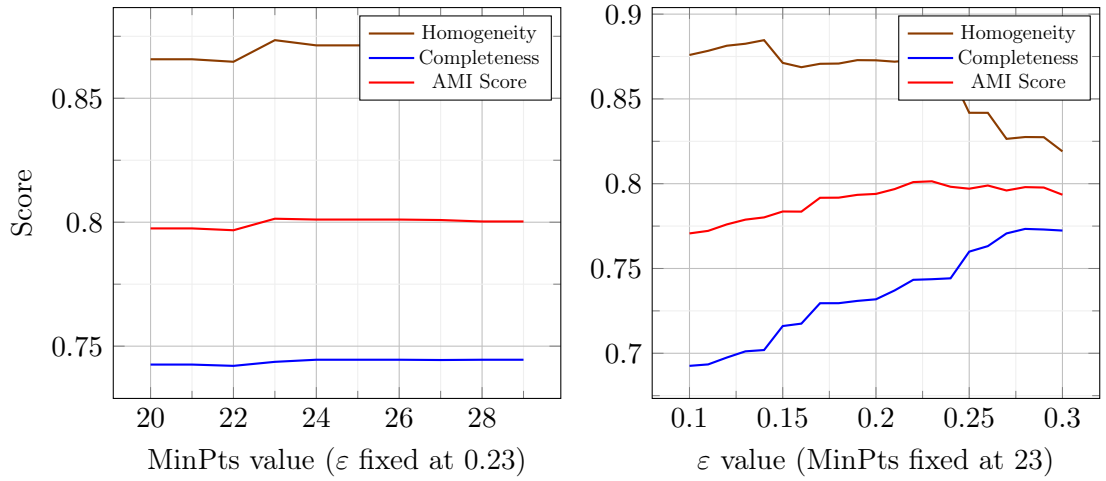

Figure 5.2: DBSCAN Section feature group

5.1.2 Combination of features

Now that we have an idea of the individual strength of each feature group, we will combine them. Each combination of features is tested both weighted per the number of features in each group, or not. We only show the combination that gets the higher AMI score between the weighted and not weighted.

The weighted combination of the section features and the metadata features, in figure 5.4, gets a lower score than section alone, with an AMI score of 0.73.

The weighted combination of the entropy features and the metadata features, in figure 5.5, gets a better score than both alone, with an AMI score of 0.63.

The non weighted combination of the section features and the mnemonic sequence, in figure 5.6, gets a better score than section alone, with an AMI score of 0.83. The weighted combination of the section features and the mnemonic sequence, in figure C.1, only give an AMI score of 0.81. Showing that giving the mnemonic sequence equal importance to the section feature in the distance calculation yield a better result. Something to note here is

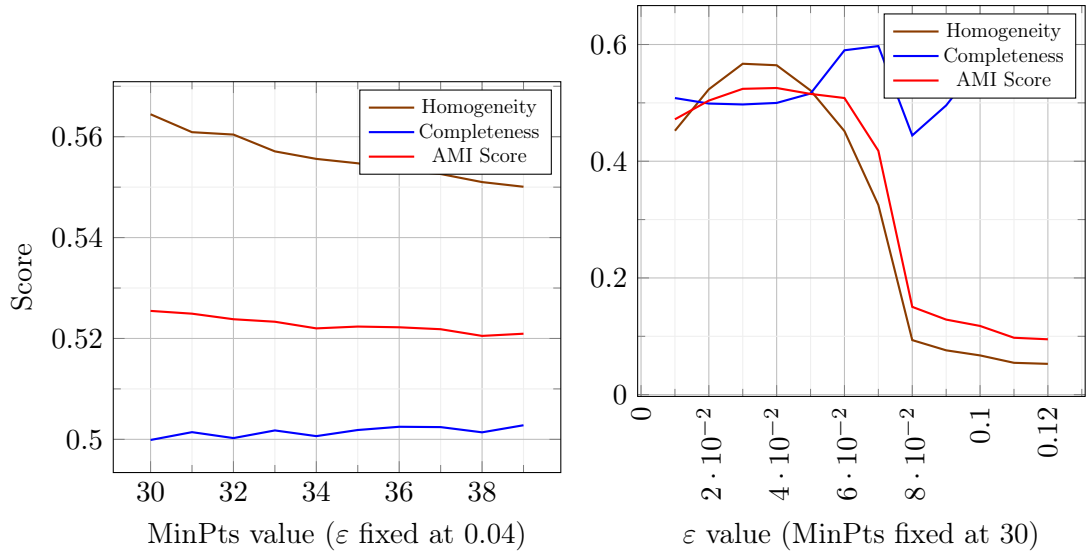

Figure 5.3: DBSCAN Entropy feature group

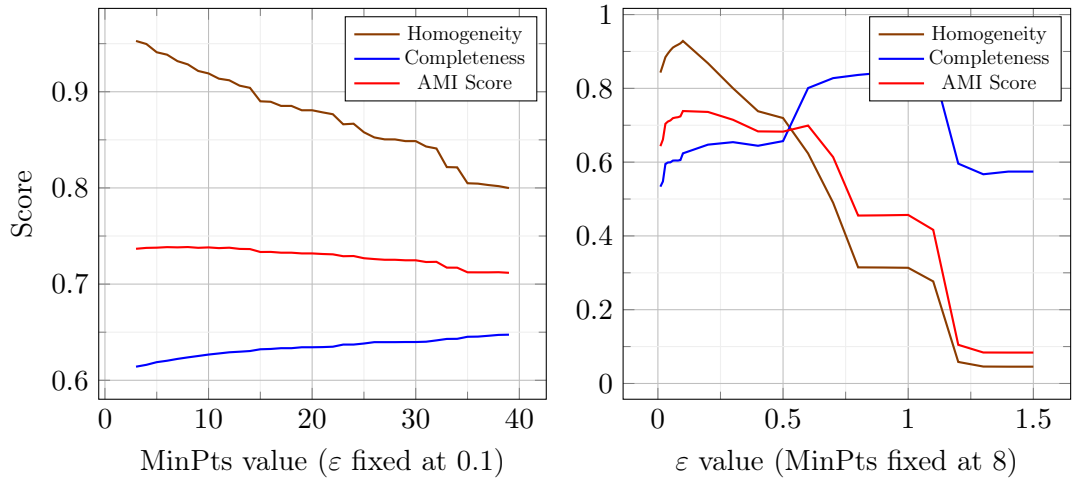

Figure 5.4: DBSCAN section and metadata feature group (weighted)

the *MinPts* value for the best AMI score, which is 1. Having *MinPts* at 1 means that every point is by default a core point. While it doesn't change the creation of clusters, it does consider every noise point as its own cluster. While DBSCAN doesn't technically treat a *MinPts* of 1 as a wrong parameter, we prefer to consider 2 as the optimal *MinPts* value in these cases, as to not make the noise disappear and the number of clusters potentially explode.

The combination of the section features and the mnemonic sequence gives the best result, and is the one we will use to test the rest of our features. This combination was also found by Noureddine et al. [23] to be the best one. However with a far better AMI score of up to 0.98 with their synthetic dataset. This shows that our dataset is not as good as what has been done previously, but it also shows, by arriving at the same conclusion, that it is still valid to test our new features.

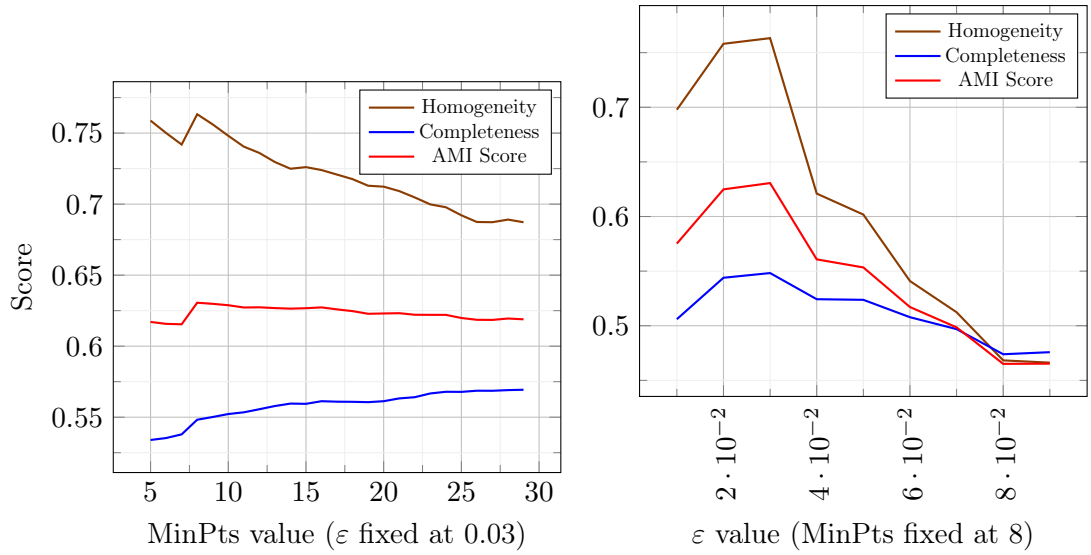

Figure 5.5: DBSCAN entropy and metadata feature group (weighted)

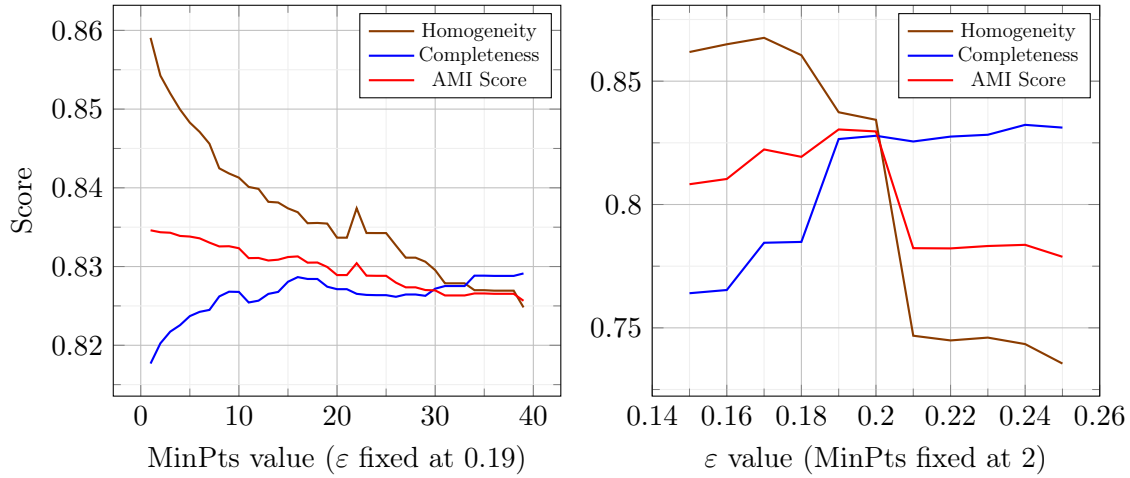

Figure 5.6: DBSCAN section and mnemonic feature group (not weighted)

5.1.3 Semantic sequence

The semantic sequence is a new feature, explained in section 4.2. By combining it with the section features, in figure 5.7, we can see that it performs slightly worse than with the mnemonic sequence in figure 5.6.

5.1.4 Length of the sequences

Up until now, the distance matrix of the mnemonic and semantic sequences we calculated with up to 100 values in the sequences. As we explained in section 2.3.2, the importance of each element in the sequence is decreasing the deeper you go in the sequence. So a longer sequence will give more importance to more elements than a shorter one.

To test if it has any impact on the final result. We tested with a combination of the section features and the mnemonic sequence, or the semantic sequence, at lengths of 30, 50, 70 and 100. As you can see in figure C.2, the result shows hardly any difference between the different lengths, with the mnemonic sequences doing slightly better than the

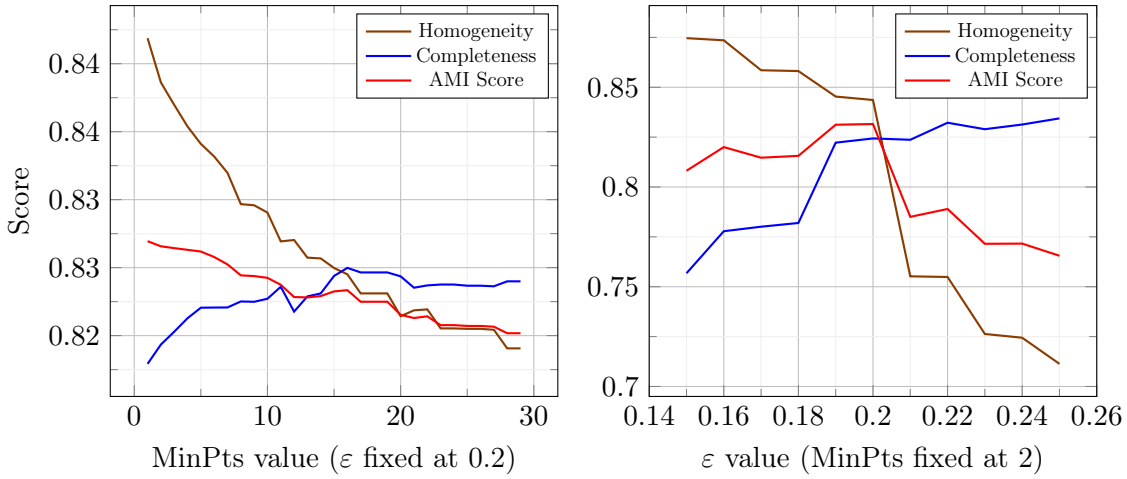

Figure 5.7: DBSCAN section and semantic feature group (not weighted)

semantic sequences. This difference in the score is at most 0.004, which is in the margin of error of DBSCAN nondeterministic nature.

5.1.5 Hidden imports

The hidden imports sequence is a new feature, explained in section 4.2. By combining it with the section features, in figure 5.8, we can see that it performs slightly worse than the section features alone in figure 5.2.

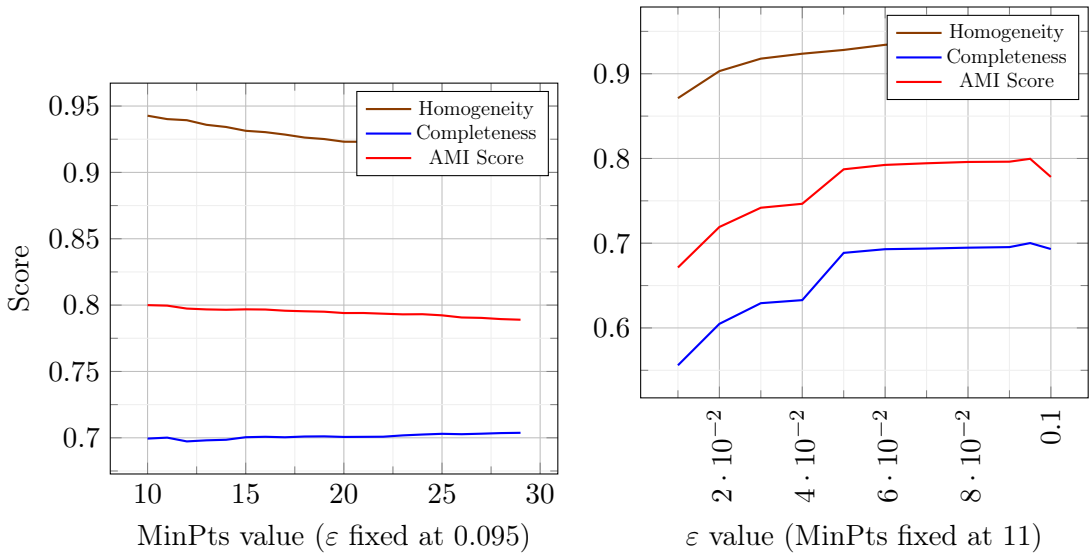

Figure 5.8: DBSCAN section and hidden import feature group (weighted)

5.2 Features selection and testing for OPTICS

The original goal of this section was to run the same test as we did with DBSCAN, But OPTICS takes far more time to execute than DBSCAN. To give an idea of the magnitude of the difference, OPTICS takes an average of 45 minutes where DBSCAN takes 2.2 seconds.

This means that we can't just test all possible values for $MinPts$ with all possible combinations of features. To find the ideal value for $MinPts$, we used the combination

of the section features and the mnemonic sequence of length 100, and tested a range of values. The result is worse than DBSCAN, with a max AMI score of 0.74. It was obtained with a $MinPts$ of 208, as you can see in figure 5.9

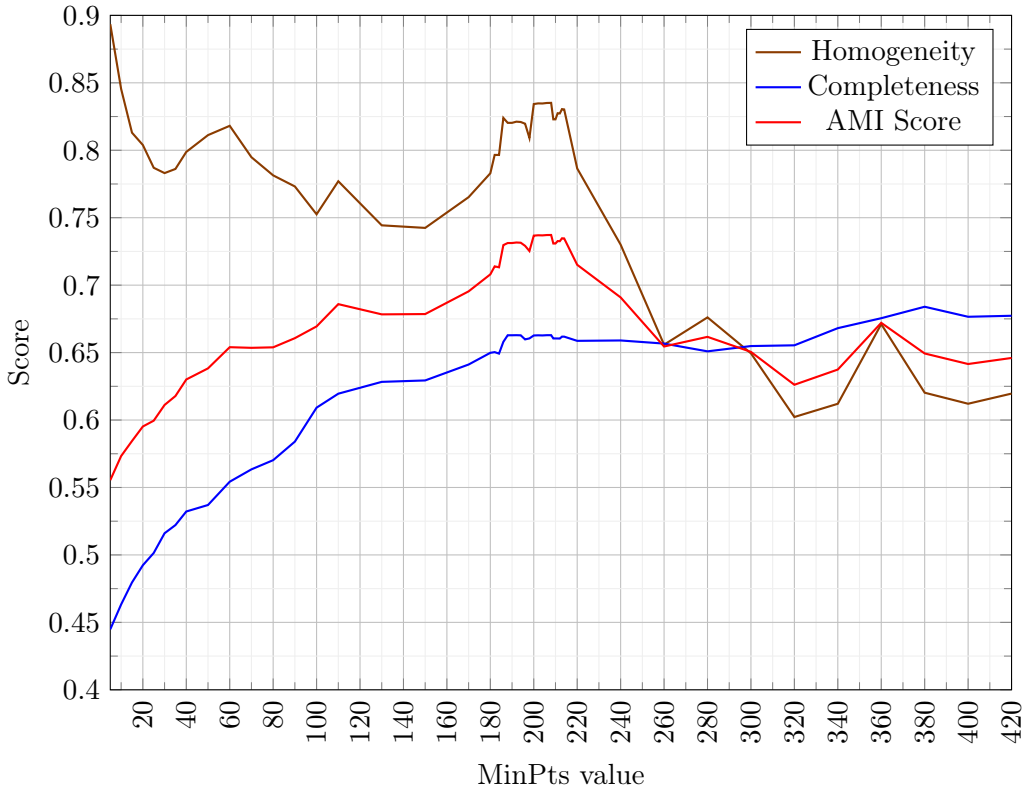

Figure 5.9: OPTICS section and mnemonic(Length 100) feature group (weighted)

5.2.1 Length of the sequences

As show in figure 5.10, testing the sequences length give the same result as with DBSCAN, it is not a relevant factor.

5.2.2 hidden imports

Hidden imports, like with DBSCAN, is not a relevant feature.

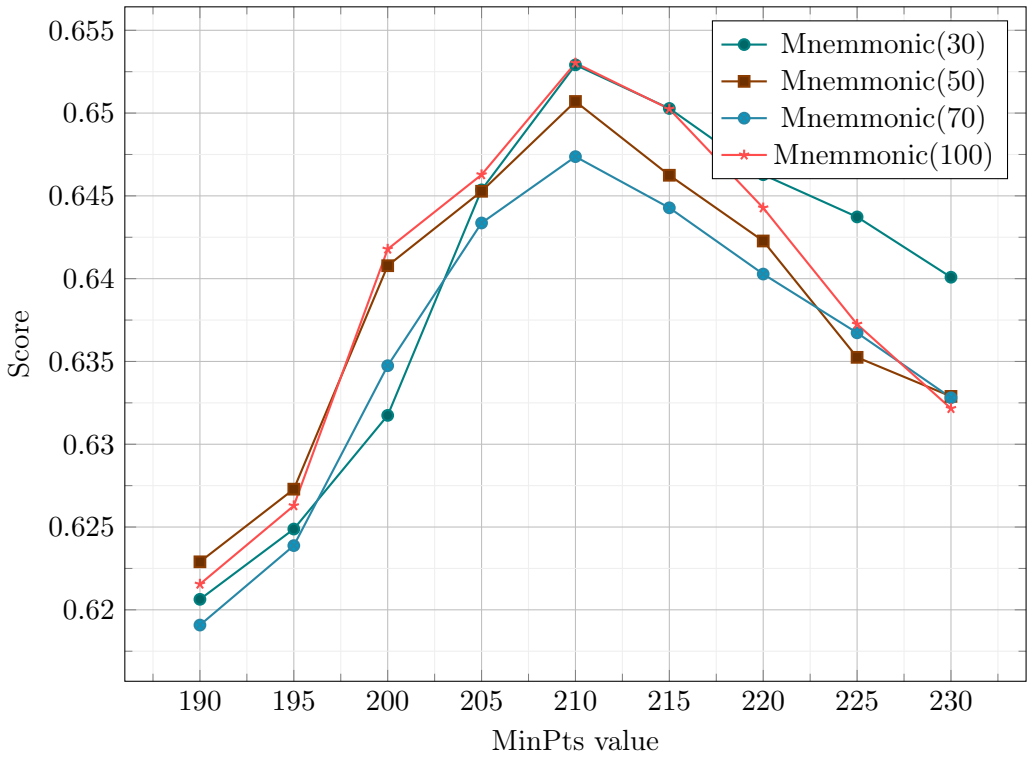

Figure 5.10: OPTICS AMI score for different length of mnemonic with the section feature group (not weighted)

Chapter 6

Conclusions

The objective of this work was to find new features and try a new clustering algorithm. With the hope of improving the quality of classification of packed executable using unsupervised clustering algorithms. We started by creating our dataset and ground truth using malware from abuse.ch [1], Github [18] and executable we packed ourselves. We checked our dataset quality by comparing it with results from previous work [23], and achieved a dataset that, while not on par with the ones from previous work, was good enough to provide our experiment with meaningful results.

6.1 Findings

Our first new feature was the semantic sequence, representing the semantic of the first 100 mnemonics executed by each sample when run. While testing it, we found out that it gave slightly worse results than the mnemonic sequence.

Our second experiment was testing the impact of both the mnemonic and semantic sequence lengths on the final score. And again the results were inconclusive, with the mnemonics sequence of length 100 slightly above the rest, but still in the marge of error coming from DBSCAN nondeterministic nature.

The last new feature was looking for hidden import of 16 API functions, given by Zakeri et al. [28]. Using Radare2 we searched for their name in each sample, and counted the number of them not present in the import table. The test shows that this feature is not relevant and lowers the results when included.

Finally, we tested the clustering algorithm OPTICS in place of DBSCAN. Again, the result was negative. With score around 10% worse than what could be achieved with DBSCAN. While testing the efficiency of the algorithms was not the goal, we can clearly say that OPTICS is ten of thousands of times slower than DBSCAN. Making it practically unusable for real time use in an AV solution.

6.2 Further Works

In this work we analysed the features by groups. While this allowed us to quickly identify which groups and combinations were relevant, it is not perfect. Take the mnemonic sequence for example, who is unable to produce any useful result on is own, but can improve the result of the section features when combined with them. With 56 features in our original feature list (table B), we can probably still improve our result by combining and testing them outside of their groups.

Although we got an idea of the results OPTICS can give us, there is still room for improvement. Firstly, we did not change the default value of the steepness parameter, used in the clustering phase of the algorithm. Secondly, our dataset was not ideal, while

we could compare our early result with previous work for DBSCAN, there is a possibility that OPTICS was more impacted by its quality.

Bibliography

[1] Abuse.ch malware database, bazaar.abuse.ch/browse/, accessed May 2022. [Cited on pages 12 and 25.]

[2] Amber packer, github.com/EgeBalci/amber, accessed May 2022. [Cited on pages 2 and 13.]

[3] Capstone, capstone-engine.org, accessed May 2022. [Cited on page 16.]

[4] Eronana packer, github.com/Eronana/packer, accessed May 2022. [Cited on page 13.]

[5] Mpress packer, autohotkey.com/mpress/mpress_web.htm, accessed May 2022. [Cited on page 13.]

[6] Pecompact packer, bitsum.com/portfolio/pecompact/, accessed May 2022. [Cited on page 13.]

[7] Pefile, github.com/erocarrera/pefile, accessed May 2022. [Cited on page 14.]

[8] Petite packer, un4seen.com/petite/, accessed May 2022. [Cited on page 13.]

[9] Radare2, rada.re/n/radare2.html, accessed April 2022. [Cited on page 14.]

[10] scikit-learn documentation, scikit-learn.org/stable, accessed May 2022. [Cited on pages IV, 3, and 6.]

[11] Ucl compression library, oberhumer.com/opensource/ucl/, accessed May 2022. [Cited on page 2.]

[12] Wikipedia optics algorithm, en.wikipedia.org/wiki/OPTICS_algorithm, accessed May 2022. [Cited on pages V and 29.]

[13] Yoda's protector, sourceforge.net/projects/yodap/, accessed May 2022. [Cited on page 2.]

[14] Ankerst, M., Breunig, M.M., Kriegel, H.P., Sander, J.: Optics: Ordering points to identify the clustering structure. In: Proceedings International Conference on Management of Data. pp. 49–60. Munich, Germany (1999) [Cited on page 4.]

[15] Bat-Erdene, M., Park, H., Li, H., Choi, H.L.S.: Entropy analysis to classify unknown packing algorithms for malware detection. International Journal of Information Security 16, 227–248 (2017) [Cited on page 10.]

[16] Biondi, F., Enescu, M.A., Given-Wilson, T., Legay, A., Noureddine, L., Verma, V.: Effective, efficient, and robust packing detection and classification. Computers and Security 85, 436–451 (2019) [Cited on pages 1 and 10.]

[17] Chire: Dbscan illustration, travail personnel, commons.wikimedia.org/w/index.php?curid=17045963, license: Creative Commons BY-SA 3.0 [Cited on pages IV and 4.]

[18] D'Hondt, A.: Github dataset, github.com/dhondta/dataset-packed-pe, accessed May 2022. [Cited on pages 13 and 25.]

[19] Ester, M., Kriegel, H.P., Sander, J., Xu, X.: A density-based algorithm for discovering clusters in large spatial databases with noise. In: Proceedings of 2nd International Conference on Knowledge Discovery and Data Mining. pp. 226–231. Munich, Germany (1996) [Cited on pages 3 and 4.]

[20] Evans, D.: University of virginia cs216: x86 assembly guide, `cs.virginia.edu/~evans/cs216/guides/x86.html`, based on materials originally created by Adam Ferrari many years ago, and since updated by Alan Batson, Mike Lack, and Anita Jones. Accessed May 2022. [Cited on page 16.]

[21] Gower, J.C.: A general coefficient of similarity and some of its properties. In: Biometrics. vol. 27, pp. 857–74 (1971) [Cited on page 7.]

[22] Lamine, N.: Packing detection and classification relying on machine learning to stop malware propagation. Phd thesis, Cryptography and Security [cs.CR]. Université Rennes 1 (2021), nNT: 2021REN1S091 . tel-03522278v3 [Cited on page 11.]

[23] Noureddine, L., Heuser, A., Puodzius, C., Zendra, O.: Se-pac: A self-evolving packer classifier against rapid packers evolution. In: Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy. pp. 1–12. Virtual Event, USA (2021), hal-03149211 [Cited on pages 1, 6, 10, 12, 14, 20, and 25.]

[24] Romano, S., Vinh, N.X., Bailey, J., Verspoor, K.: Adjusting for chance clustering comparison measures. In: Nowozin, S. (ed.) Journal of Machine Learning Research 17. pp. 1–32. Montreal, Canada (2016) [Cited on page 9.]

[25] Rosenberg, A., Hirschberg, J.: V-Measure: A Conditional Entropy-Based External Cluster Evaluation Measure. In: Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning. pp. 410–420. Prague, Czech Republic (2007) [Cited on page 8.]

[26] Thomas Roccia: Malware packers use tricks to avoid analysis, detection. (2017-05-16), `www.mcafee.com/blogs/enterprise/malware-packers-use-tricks-avoid-analysis-detection/`, accessed May 2022. [Cited on pages IV and 2.]

[27] Vinh, N.X., Epps, J., Bailey, J.: Information theoretic measures for clusterings comparison: is a correction for chance necessary? In: Proceedings of the 26th Annual International Conference on Machine Learning. pp. 1073–1080. Montreal, Canada (2009) [Cited on page 9.]

[28] Zakeri, M., Daneshgar, F.F., Abbaspour, M.: A static heuristic approach to detecting malwaretargets. Security and Communication Networks 8(17), 3015–3027 (2015) [Cited on pages I, 17, and 25.]

Appendix A

Source code

A.1 OPTICS source code

```
function OPTICS(DB, eps, MinPts) is
    for each point p of DB do
        p.reachability-distance = UNDEFINED
    for each unprocessed point p of DB do
        N = getNeighbors(p, eps)
        mark p as processed
        output p to the ordered list
        if core-distance(p, eps, MinPts) != UNDEFINED then
            Seeds = empty priority queue
            update(N, p, Seeds, eps, MinPts)
            for each next q in Seeds do
                N' = getNeighbors(q, eps)
                mark q as processed
                output q to the ordered list
                if core-distance(q, eps, MinPts) != UNDEFINED do
                    update(N', q, Seeds, eps, MinPts)
```

Listing A.1: Main loop [12]

```
function update(N, p, Seeds, eps, MinPts) is
    coredist = core-distance(p, eps, MinPts)
    for each o in N
        if o is not processed then
            new-reach-dist = max(coredist, dist(p,o))
            if o.reachability-distance == UNDEFINED then
                o.reachability-distance = new-reach-dist
                Seeds.insert(o, new-reach-dist)
            else
                if new-reach-dist < o.reachability-distance then
                    o.reachability-distance = new-reach-dist
                    Seeds.move-up(o, new-reach-dist)
```

Listing A.2: Update function [12]

A.2 Radare2 source code

```
static struct optype {
        int type;
        const char *name;
} optypes[] = {
        { R_ANAL_OP_TYPE_IO, "io" },
        { R_ANAL_OP_TYPE_ACMP, "acmp" },
        { R_ANAL_OP_TYPE_ADD, "add" },
        { R_ANAL_OP_TYPE_SYNC, "sync" },
        { R_ANAL_OP_TYPE_AND, "and" },
        { R_ANAL_OP_TYPE_CALL, "call" },
        { R_ANAL_OP_TYPE_CCALL, "ccall" },
        { R_ANAL_OP_TYPE_CJMP, "cjmp" },
        { R_ANAL_OP_TYPE_MJMP, "mjmp" },
        { R_ANAL_OP_TYPE_CMP, "cmp" },
        { R_ANAL_OP_TYPE_IO, "cret" },
        { R_ANAL_OP_TYPE_ILL, "ill" },
        { R_ANAL_OP_TYPE_JMP, "jmp" },
        { R_ANAL_OP_TYPE_LEA, "lea" },
        { R_ANAL_OP_TYPE_LEAVE, "leave" },
        { R_ANAL_OP_TYPE_LOAD, "load" },
        { R_ANAL_OP_TYPE_NEW, "new" },
        { R_ANAL_OP_TYPE_MOD, "mod" },
        { R_ANAL_OP_TYPE_CMOV, "cmov" },
        { R_ANAL_OP_TYPE_MOV, "mov" },
        { R_ANAL_OP_TYPE_CAST, "cast" },
        { R_ANAL_OP_TYPE_MUL, "mul" },
        { R_ANAL_OP_TYPE_DIV, "div" },
        { R_ANAL_OP_TYPE_NOP, "nop" },
        { R_ANAL_OP_TYPE_NOT, "not" },
        { R_ANAL_OP_TYPE_NULL  , "null" },
        { R_ANAL_OP_TYPE_OR    , "or" },
        { R_ANAL_OP_TYPE_POP   , "pop" },
        { R_ANAL_OP_TYPE_PUSH  , "push" },
        { R_ANAL_OP_TYPE_REP   , "rep" },
        { R_ANAL_OP_TYPE_RET   , "ret" },
        { R_ANAL_OP_TYPE_ROL   , "rol" },
```

...

```
        { R_ANAL_OP_TYPE_UNK   , "unk" },
        { R_ANAL_OP_TYPE_UPUSH , "upush" },
        { R_ANAL_OP_TYPE_RPUSH , "rpush" },
        { R_ANAL_OP_TYPE_XCHG  , "xchg" },
        { R_ANAL_OP_TYPE_XOR   , "xor" },
        { R_ANAL_OP_TYPE_CASE  , "case" },
        { R_ANAL_OP_TYPE_CPL   , "cpl" },
        { R_ANAL_OP_TYPE_CRYPTO, "crypto" },
        {0,NULL}
};
```

Code snippet 6: Radare2 type list

```
static void anop(RAnal *a, RAnalOp *op, ut64 addr, const ut8 *buf, int len, csh *handle,
cs_insn *insn) {
        int bits = a->config->bits;
        struct Getarg gop = {
                .handle = *handle,
                .insn = insn,
                .bits = a->config->bits
        };
        int regsz = 4;
        switch (bits) {
        case 64: regsz = 8; break;
        case 16: regsz = 2; break;
        default: regsz = 4; break; // 32
        }
        switch (insn->id) {
        case X86_INS_FNOP:
                op->family = R_ANAL_OP_FAMILY_FPU;
                /* fallthru */
        case X86_INS_NOP:
        case X86_INS_PAUSE:
                op->type = R_ANAL_OP_TYPE_NOP;
                break;
        case X86_INS_HLT:
                op->type = R_ANAL_OP_TYPE_TRAP;
                break;
        case X86_INS_FBLD:
        case X86_INS_FBSTP:
        case X86_INS_FCOMPP:
        case X86_INS_FDECSTP:
```

Code snippet 7: Capstone mnemonic to Radare2 type

Appendix B

Tables

Feature id	Description	Category
1	the DLLs characteristics 1	Metadata
2	the DLLs characteristics 2	Metadata
3	the DLLs characteristics 3	Metadata
4	the DLLs characteristics 4	Metadata
5	the DLLs characteristics 5	Metadata
6	the DLLs characteristics 6	Metadata
7	the DLLs characteristics 7	Metadata
8	the DLLs characteristics 8	Metadata
9	the Checksum	Metadata
10	the Image Base	Metadata
11	the Base of Code	Metadata
12	the OS Major version	Metadata
13	the OS Minor version	Metadata
14	the Size of Image	Metadata
15	the Size of Code	Metadata
16	the Headers	Metadata
17	the Size Of InitializedData	Metadata
18	the Size Of UninitializedData	Metadata
19	the Size Of StackReserve	Metadata
20	the Size of Stack Commit	Metadata
21	the chapter Alignment (Size of memory pages)	Metadata
22	the number of standards chapters the PE holds	Section
23	the number of non-standards chapters the PE holds	Section
24	the ratio between the number of standards chapters found and the number of all chapters found in the PE under analysis	Section
25	the number of Executable chapters the PE holds	Section
26	the number of Writable chapters the PE holds	Section
27	the number of Writable and Executable chapters the PE holds	Section
28	the number of readable and executable chapters	Section
29	the number of readable and writable chapters	Section
30	the number of Writable and Readable and Executable chapters the PE holds	Section
31	the code chapter is not executable	Section
32	the executable chapter is not a code chapter	Section

33	the code chapter is not present in the PE under analysis	Section
34	the EP is not in the code chapter	Section
35	the EP is not in a standard chapter	Section
36	the EP is not in an executable chapter	Section
37	the EP ratio between raw data and virtual size for the chapter of entry point	Section
38	the number of chapters having their physical size = 0 (size on disk)	Section
39	the number of chapters having their virtual size greater than their raw data size	Section
40	the maximum ratio raw data per virtual size among all the chapters	Section
41	the minimum ratio raw data per virtual size among all the chapters	Section
42	the address pointing to raw data on disk is not conforming with the file alignement	Section
43	the entropy of Code/text chapters	Byte entropy
44	the entropy of data chapter	Byte entropy
45	the entropy of resource chapter	Byte entropy
46	the entropy of PE header	Byte entropy
47	the entropy of the entire PE file	Byte entropy
48	the entropy of chapter holding the Entry point (EP) of the PE under analysis	Byte entropy
49	the number of DLLs imported	Import functions
50	the number of functions imported found in the import table directory (IDT)	Import functions
51	the number of malicious APIs imported	Import functions
52	the ratio between the number of malicious APIs imported to the number of all functions imported by the PE	Import functions
53	the number of addresses (corresponds to functions) found in the import address table (IAT)	Import functions
54	the debug directory is present or not	resource
55	the number of resources the PE holds	resource
56	the sequence of the first 100 mnemonics executed	mnemonic

Table B.1: Base features

Appendix C

Figures

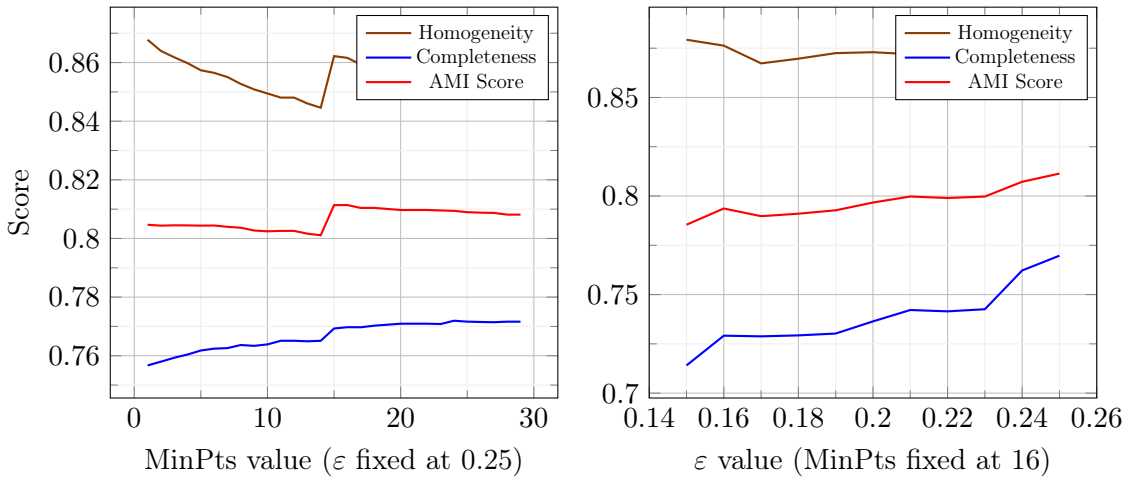

Figure C.1: DBSCAN section and mnemonic feature group (weighted)

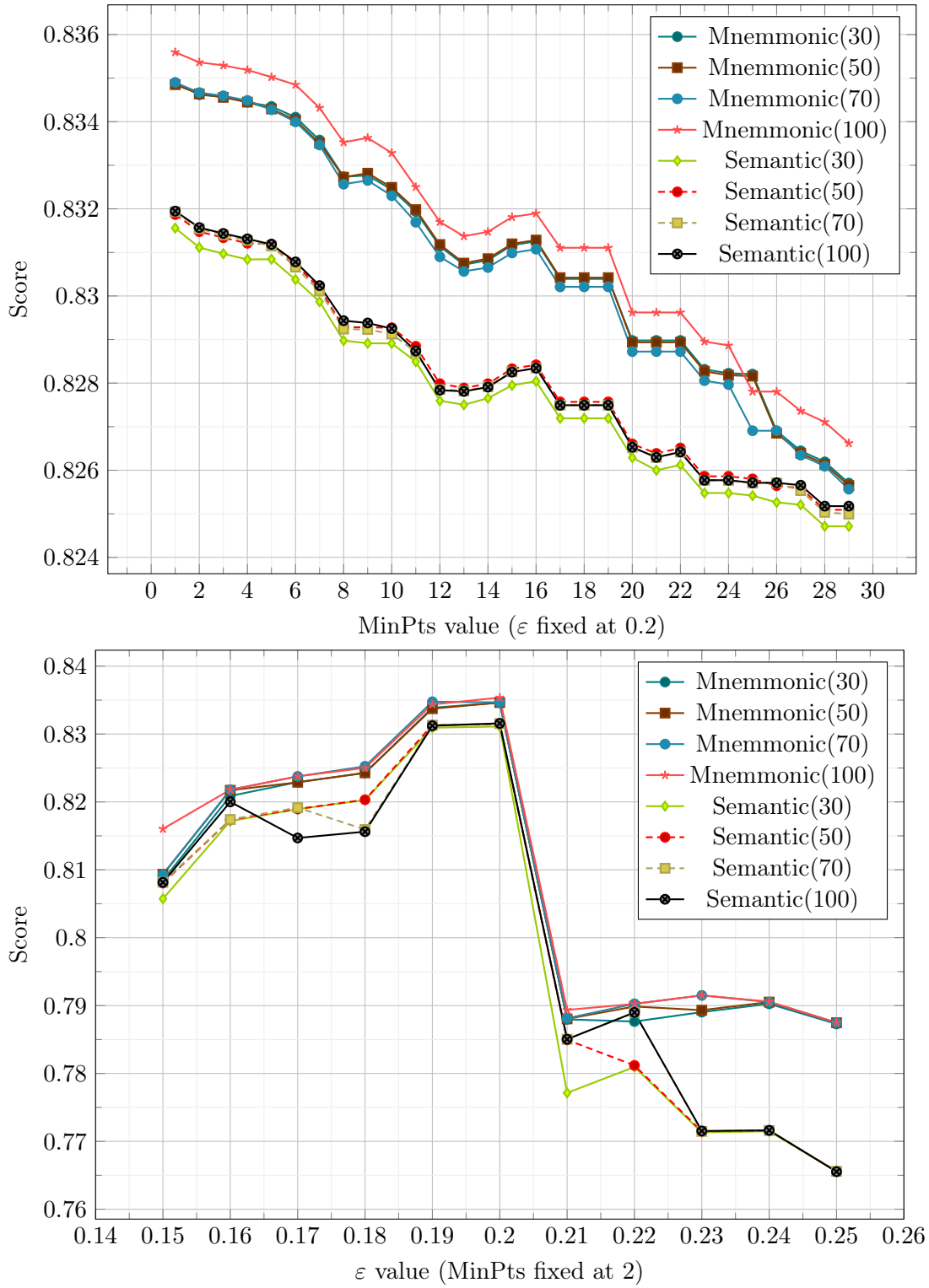

Figure C.2: DBSCAN AMI score for different length of mnemonic/semantic with the section feature group (not weighted)

C.1 Number of clusters

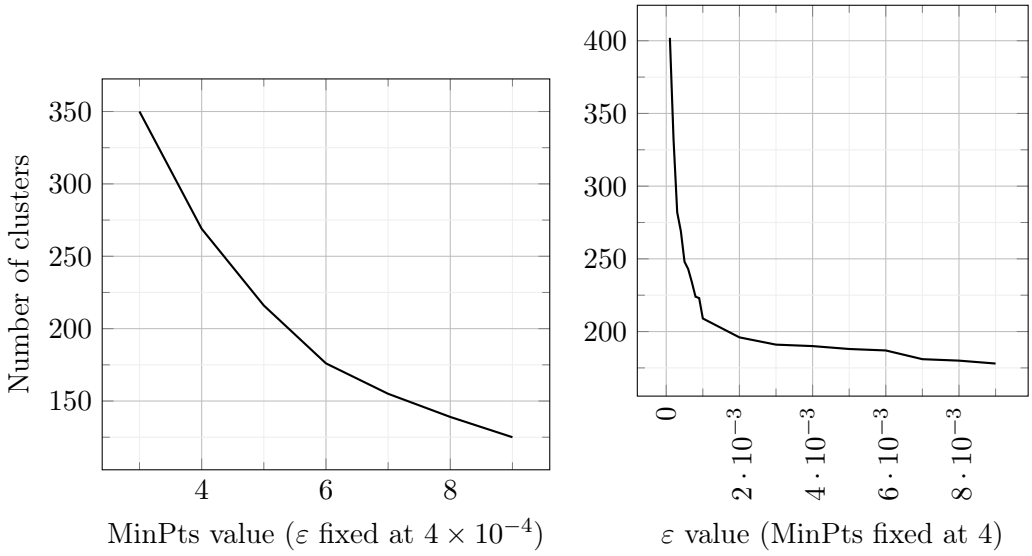

Figure C.3: DBSCAN Metadata feature group: cluster numbers

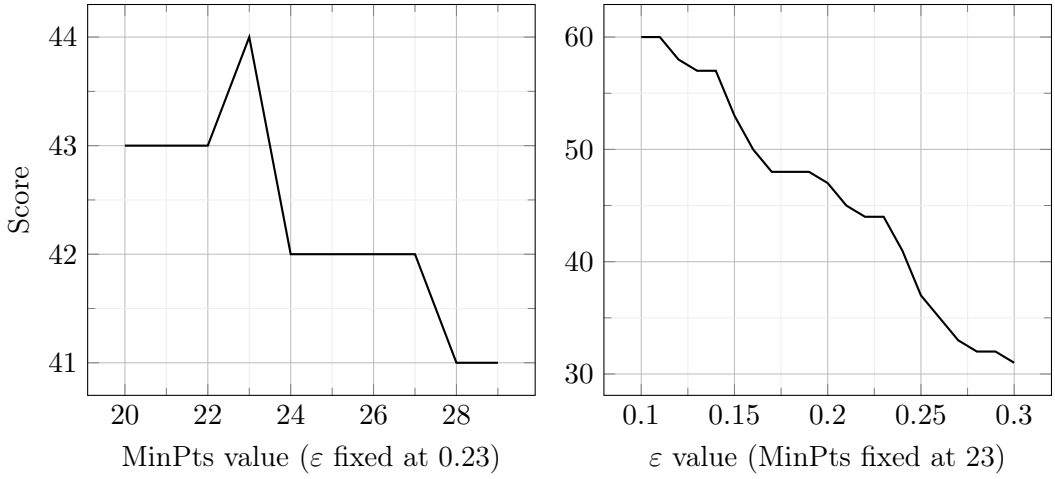

Figure C.4: DBSCAN Section feature group: cluster numbers

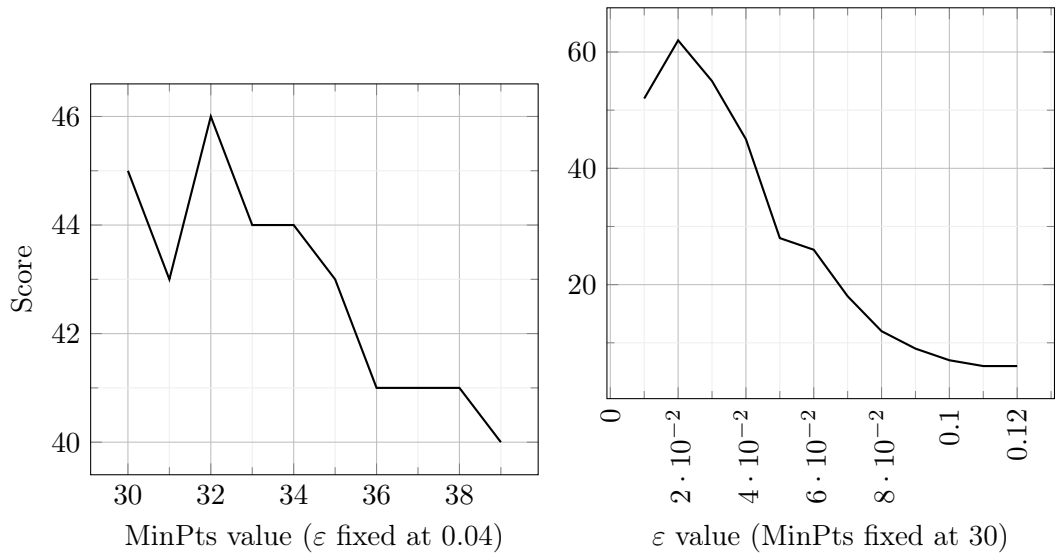

Figure C.5: DBSCAN Entropy feature group: cluster numbers

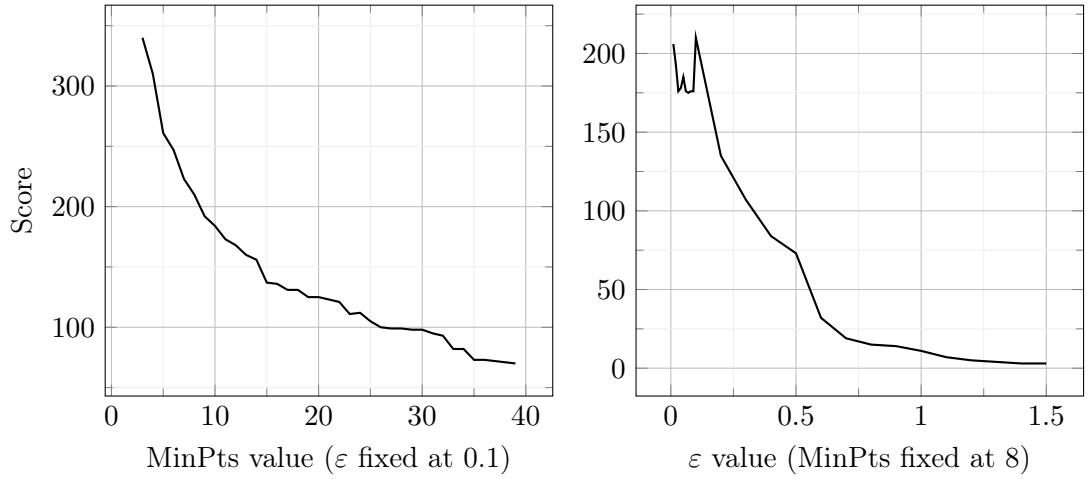

Figure C.6: DBSCAN section and metadata feature group (weighted): cluster numbers

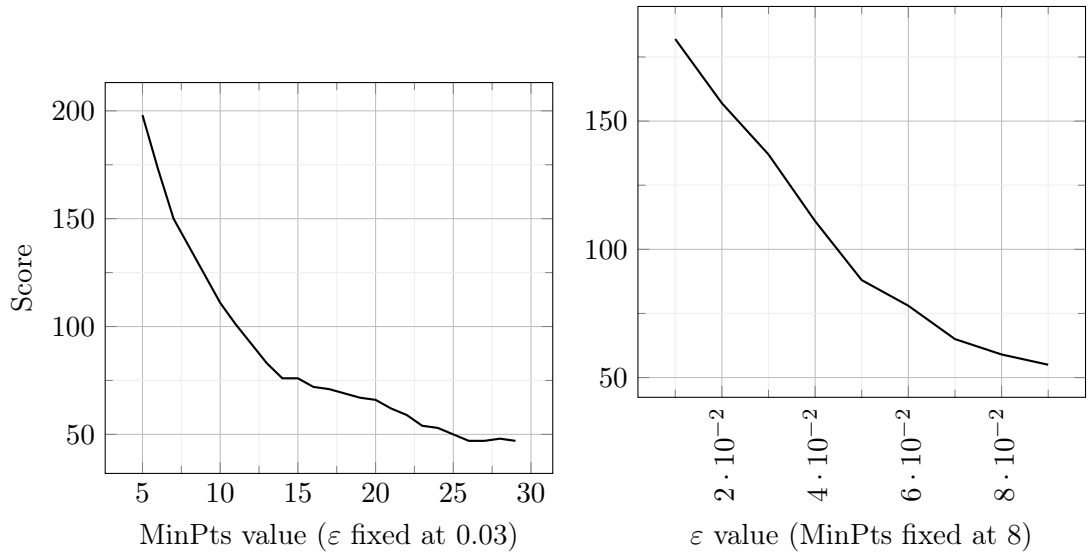

Figure C.7: DBSCAN entropy and metadata feature group (weighted): cluster numbers

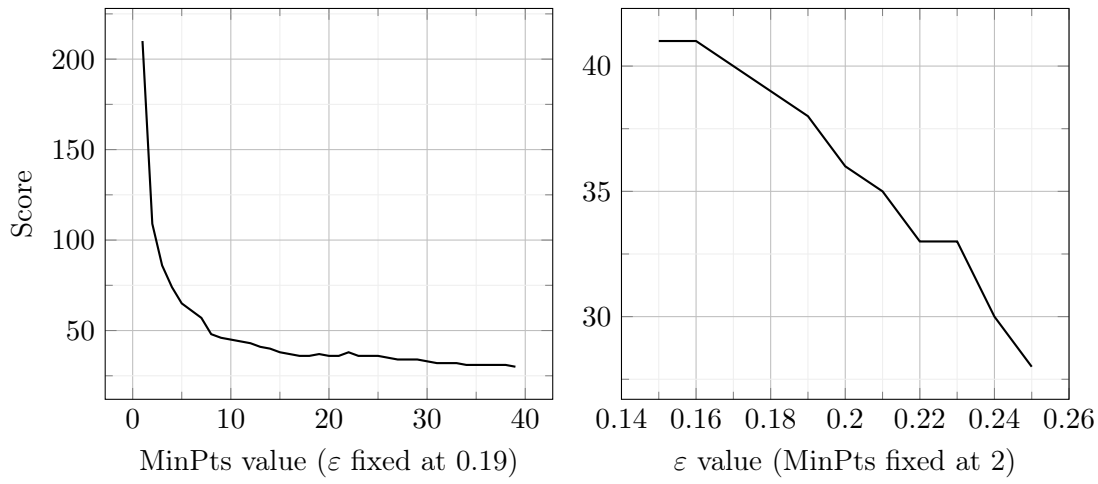

Figure C.8: DBSCAN section and mnemonic feature group (not weighted): cluster numbers

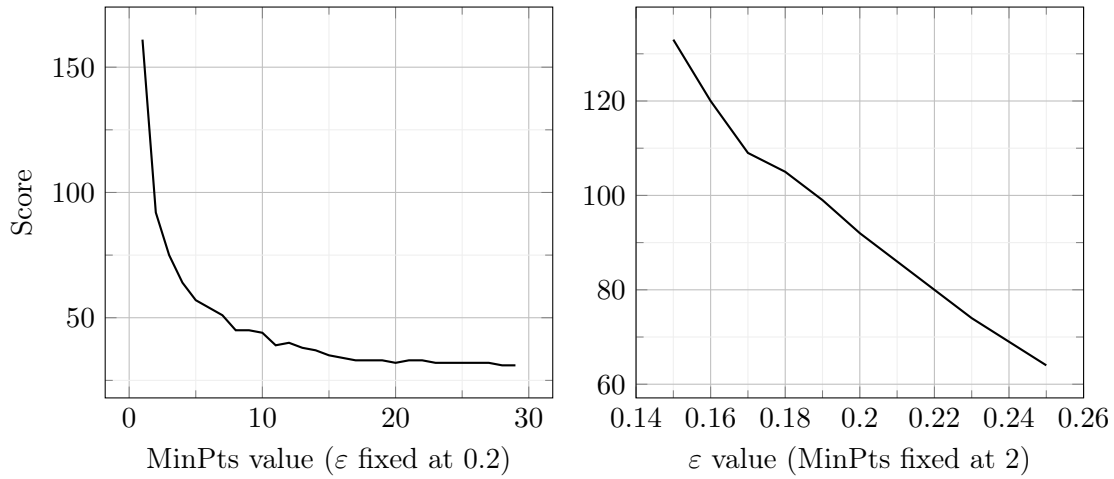

Figure C.9: DBSCAN section and semantic feature group (not weighted): cluster numbers

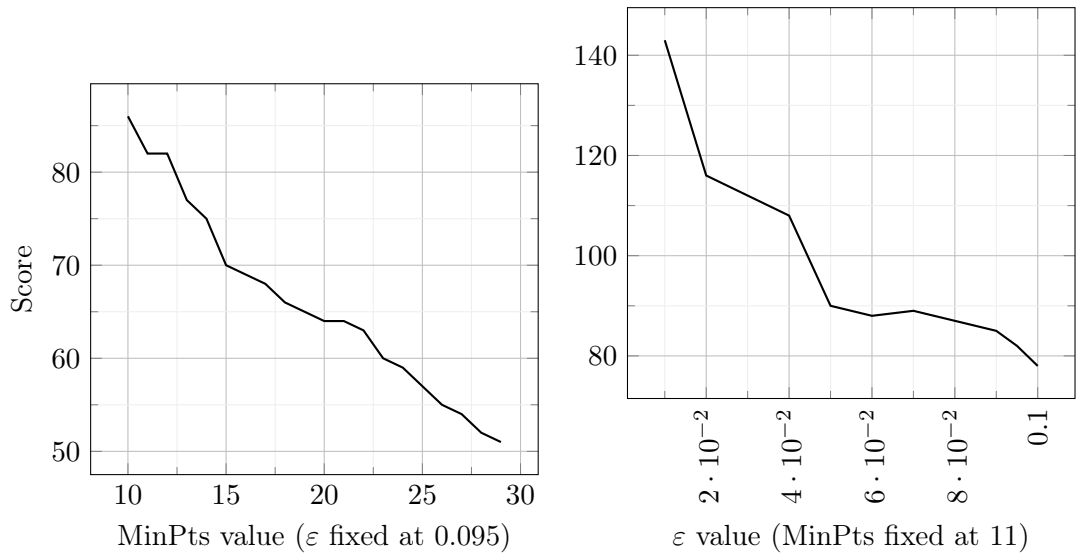

Figure C.10: DBSCAN section and hidden import feature group (weighted): cluster numbers

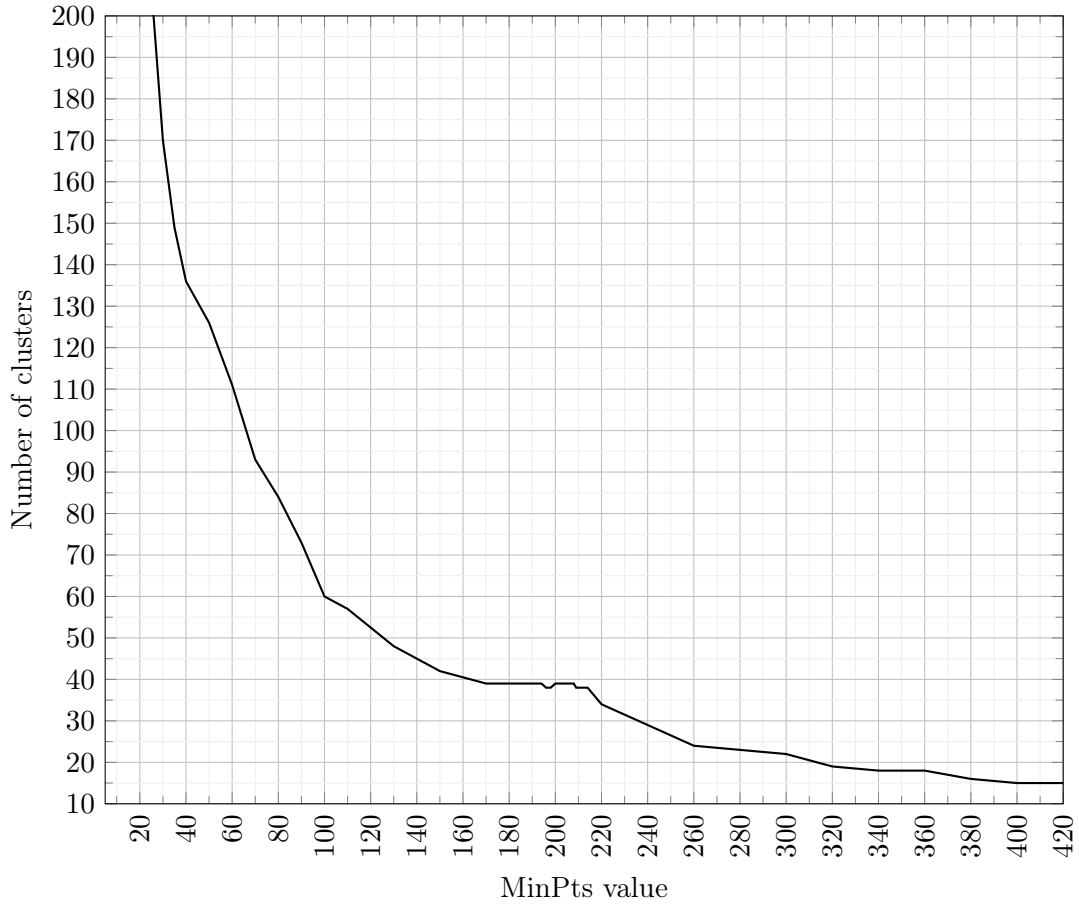

Figure C.11: OPTICS section and mnemonic(Length 100) feature group (not weighted): cluster numbers

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**