ISR project
Threat intelligence
Proposal #2

Emese PADANYI

Ede KOMJATI

Date: 2019.12.11.

Programme: EIT Digital Cybersecurity M1

Source code: https://github.com/caspien6/isr_project

1. Data structure:

The data that we received is organized in the following structure:

- VirusTotal: analysis in JSON from VirusTotal.
- File: hash from the file.
- Path: the original place of the sample.
- Rabin2: meta information of the file.
- Strings: All the strings that the malware contains.
- Yara: All the rules that the file matched with.

2. Data cleansing

To be able to cluster the data, we must prepare our data for it: select features and transform them into processable entities. We have just a few options to choose from, since a couple of columns does not contain relevant information for clustering, or already contains the solution. For example:

- VirusTotal – Because it is an analysis already made, so it wouldn't add value to reuse it.
- File – Because it is just a dictionary contained hashes from the sample. Every sample has a different hash. It is meant for identification, not for clustering.
- Path – Because it is just a file path where the sample came from. It wouldn't help us to decide which sample goes where.

Our solution to cleanse the strings

We have chosen the **Strings** column to experiment with. But to be able to use those strings we should make them understandable to the clustering algorithms. The algorithms take matrixes of numbers, so we must aim to match that.

First, we cleaned up the strings a little bit. We implemented a function that gets rid of too short or too long strings from the lists. The user can change these parameters to fine-tune the algorithm. Also, we got rid of those strings which aren't represented in many samples. Meaning that if a string only exists in fewer files than value, we cut the string out of the data as well.

After this procedure, the remaining strings will represent the **relevant_strings_list**. We create a matrix, where the rows are the files, and the columns are all the relevant strings. Each value in the matrix is 1 if the file contains the string; or else it is 0.

Other solutions to process strings

Before implementing our own string filtering algorithm, we considered using already existing solutions, based on our research.

First, we just wanted to encode every string with an integer ID, but this means that there are strings that are further from each other than others. This would have flawed our clustering with the kmeans algorithm because related strings might get separated and thus unrecognized by the algorithm.

We were thinking of using onehot encoded strings because that would get rid of the above-mentioned problem. But the onehot algorithm raises another problem: memory inefficient usage. Within the context of this exercise: we have 120 samples, and we have ~220.000 unique strings. We filtered them to 7000 relevant strings. If we would use the onehot algorithm on them that would mean

that one word needs 7000 columns. One sample has a ~300-1000 unique general set of strings, that means we need at least 300 words for each sample. 300 words * 7000 columns. We don't have the infrastructure to experiment with this huge dataset right now, so that is why we chose a different method. It is a bad approach because the count of the strings can be different for each sample. That is why we chose the solution mentioned above.

New processable dataset

After the cleanse, the data looks like this:

```
In [129]: 1 res_1
```

Out[129]:

	.text	GetModuleHandleA	GetLastError	ExitProcess	.rsrc	KERNEL32.dll	GetProcAddress	GetCurrentProcess	LoadLibraryA	CloseHandle	...	Monday
0	0	1	1	1	0	0	1	1	1	1	...	0
1	1	1	1	1	1	1	1	1	1	1	...	0
2	1	1	1	1	1	1	1	1	0	1	...	1
3	1	1	1	1	1	1	1	1	1	1	...	0
4	1	1	1	1	1	1	1	1	1	1	...	1
5	1	1	1	1	1	1	1	1	0	1	...	1
6	0	0	1	0	0	0	0	0	0	1	...	0
7	0	1	0	0	0	0	0	0	0	0	...	0
8	1	1	1	1	1	1	1	1	1	1	...	0
9	1	1	1	1	1	1	1	1	1	1	...	0
10	1	1	1	1	1	1	1	0	1	1	...	0
...												
109	1	1	1	1	1	1	1	1	1	1	...	1
110	1	1	1	1	1	1	0	0	0	0	...	1
111	1	1	1	1	1	1	0	0	0	0	...	1
112	1	1	1	1	1	1	0	0	0	0	...	1
113	1	1	1	1	1	1	0	0	0	0	...	1
114	1	1	1	1	1	1	0	0	0	0	...	1
115	1	1	1	1	1	1	0	0	0	0	...	1
116	1	1	1	1	1	1	0	0	0	0	...	1
117	1	1	1	1	1	1	0	0	0	0	...	1
118	1	1	1	1	1	1	0	0	0	0	...	1
119	1	1	1	1	1	1	0	0	0	0	...	1

120 rows × 734 columns

Here as above mentioned earlier the columns come from the **relevant_strings_list.** If the sample (row) contains the string, then the value in the relevant column will be 1, else 0.

3. Clustering data

For the first tests, we used k-means++ algorithm from scikit-learn library with different n_cluster. After a few samples, we tried out the dbscan and the spectral clustering too to see what the difference between the clustering algorithms are. In the end, we also tried out the Agglomerative Clustering and the Mean Shift algorithms.

K-means principles

It follows a simple procedure of classifying a given data set into several clusters, defined by the letter "k," which is fixed beforehand. The clusters are then positioned as points and all observations or data

points are associated with the nearest cluster, computed, adjusted and then the process starts overusing the new adjustments until the desired result is reached.

The algorithm follows these general steps:

1. K points are placed into the object data space representing the initial group of centroids.
2. Each object or data point is assigned to the closest k.
3. After all the objects are assigned, the positions of the k centroids are recalculated.
4. Steps 2 and 3 are repeated until the positions of the centroids no longer move.

DBSCAN principles

Consider a set of points in some space to be clustered. Let ε be a parameter specifying the radius of a neighborhood with respect to some point. For the purpose of DBSCAN clustering, the points are classified as core points, (density-)reachable points and outliers, as follows:

1. A point p is a core point if at least minPts points are within distance ε of it (including p).
2. A point q is directly reachable from p if point q is within distance ε from core point p. Points are only said to be directly reachable from core points.
3. A point q is reachable from p if there is a path p1, ..., pn with p1 = p and pn = q, where each pi+1 is directly reachable from pi. Note that this implies that all points on the path must be core points, except for q.
4. All points not reachable from any other point are outliers or noise points.

Now if p is a core point, then it forms a cluster together with all points (core or non-core) that are reachable from it. Each cluster contains at least one core point; non-core points can be part of a cluster, but they form its "edge" since they cannot be used to reach more points.

Reachability is not symmetric relation since, by definition, no point may be reachable from a non-core point, regardless of distance (so a non-core point may be reachable, but nothing can be reached from it). Therefore, a further notion of connectedness is needed to formally define the extent of the clusters found by DBSCAN. Two points p and q are density-connected if there is a point of such that both p and q are reachable from o. Density-connectedness is symmetric.

Spectral clustering principles

Spectral clustering is a technique with roots in graph theory, where the approach is used to identify communities of nodes in a graph based on the edges connecting them. The method is flexible and allows us to cluster non-graph data as well.

Spectral clustering uses information from the eigenvalues (spectrum) of special matrices built from the graph or the data set. We'll learn how to construct these matrices, interpret their spectrum, and use the eigenvectors to assign our data to clusters.

Agglomerative Clustering

This is a subset of the hierarchical clustering. This is a "bottom-up" approach: each observation starts in its own cluster, and pairs of clusters are merged as one moves up the hierarchy.

Mean shift

Mean shift clustering using a flat kernel.

Mean shift clustering aims to discover “blobs” in a smooth density of samples. It is a centroid-based algorithm, which works by updating candidates for centroids to be the mean of the points within a given region. These candidates are then filtered in a post-processing stage to eliminate near-duplicates to form the final set of centroids.

Seeding is performed using a binning technique for scalability.

4. Validating results

We tried clustering with different algorithms and parameters. But to know which one produces valuable outcomes, we had to use some metrics and visualization.

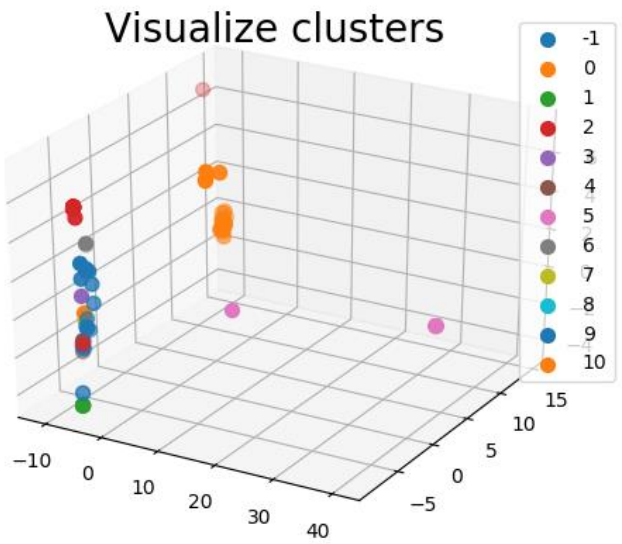
That is when we came up with the idea of tagging the samples. We needed evidence of the performance of the clustering. So, we chose a set of Yara rules for the malware families and tag each sample with one of them. After doing this we were able to check the clusters in which malware families are mapped to it.

The chosen yara rules for representing the families are:

```
malware_families = {
        'Empty': -1,        'Shohdi':0,                 'Bublik':1,
        'Mira':2,           'spyeye':3,                 'Njrat':4,
        'Shifu':5,          'sakula_v1_3':6,            'Wabot':7,
         'Warp':8,          'Nanocore_RAT_Gen_2':9,      'Wimmie':10, 'xRAT':11}
```

The Empty group means, that we couldn’t decide which malware family the sample belongs to.

The real dataset plotted in 3D with the help of PCA looks like this:



We implemented an algorithm, which automatically went through changing the parameters of the kmeans algorithm and saved the result. We looked at the outcomes and chose a couple of combinations, which seemed the best in terms of clustering the same malware samples together. It can be seen in the jupyter notebook file not in the python files. We used jupyter notebook because it was faster to try out things in Python.

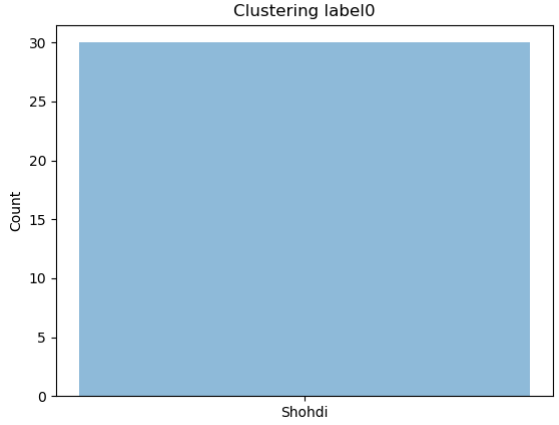
The best results came out with these string parameters:

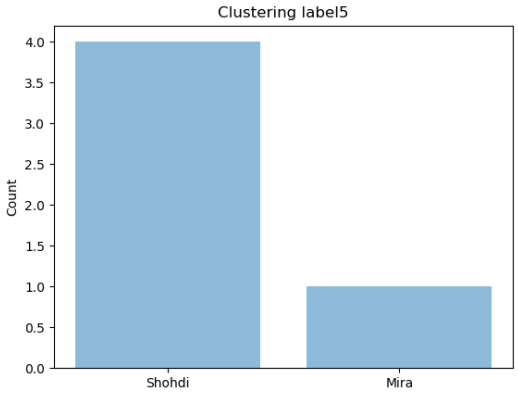
- Minimum length of string = 2
- Maximum length of string = 22
- Minimum count of sample which contains it = 17

Kmeans outcome summary

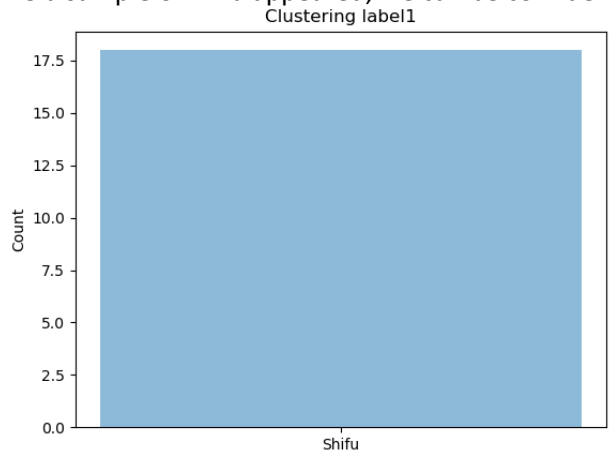
The best kmeans clustering result has 7 clusters. (labeled 0-6)

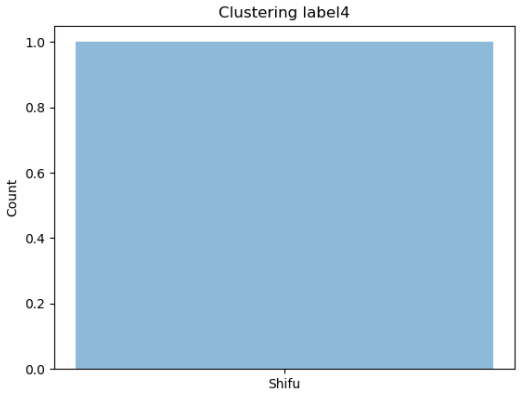
In the following diagrams, we can see the clusters one by one. The boxplots represent which malware families are included in each cluster. The plots are a little bit confusing because of the count scale changes in each.





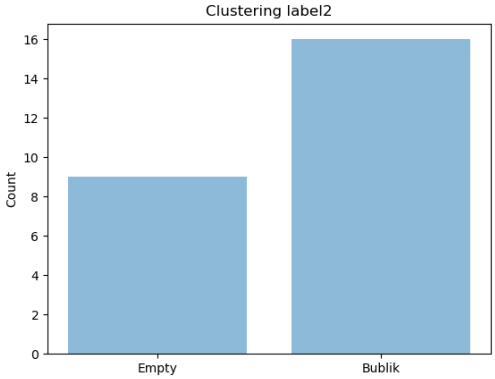
In clusters #0 and #5, almost all the samples are from Shodi malware family. Even though in cluster #5 a sample of Mira appeared, we can be confident that the clustering was successful.



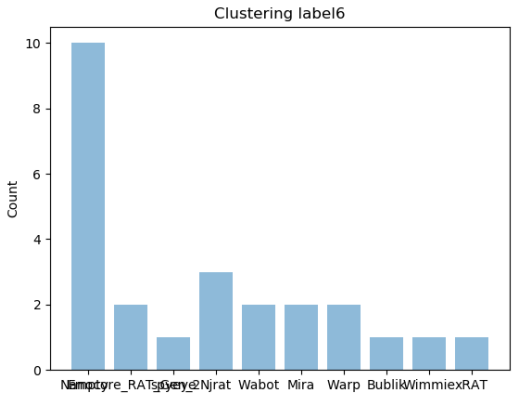


As you can see cluster #1 and #4 collected the Shifu family. Because the clustering managed to cluster them without any other type of malware, we can derive that this malware type uses some sort of string in the malware which is unique.

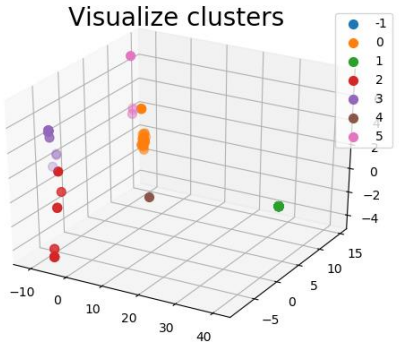
Mira family is also in almost only in one cluster: label #3. There are a couple of other malware samples which doesn't belong there but this isn't too much.



Cluster #2 is not as precise as previous clusters were. 'Empty' malware family contains malware samples that we couldn't categorize. It seems like some of them contain similar strings as Bubik does. Maybe some of the samples which we were uncertain about, should belong to Bubik as well. Maybe the Yara rules are not perfect, and it didn't pick up the files which should have been categorized as Bubik.



In cluster #6, there are many types together. It seems like the samples, which didn't fit to the rest of the clusters are clustered here. This might be the result of a lack of samples from each family. The interesting is that a couple of Mira and Bubik samples are included here, but none of the Shodi and Shifu samples are represented here.

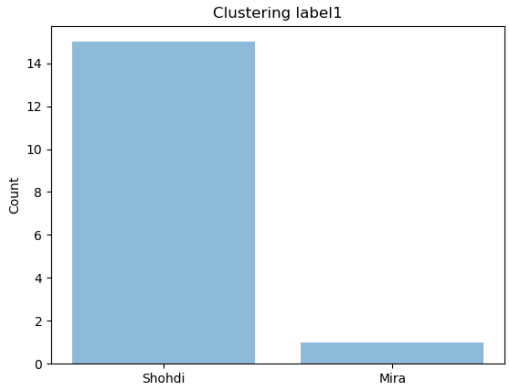


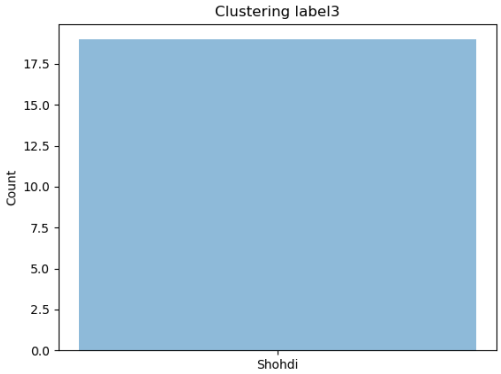
The last picture contains the 120 samples in 3D plotted, reduced the dimensions by the PCA algorithm.

In summary: this clustering did a perfect job clustering Shodi and Shifu samples. Mira and Bubik samples are also separated, but with less confidence. The rest of the samples are clustered together, so we don't get any information on them.

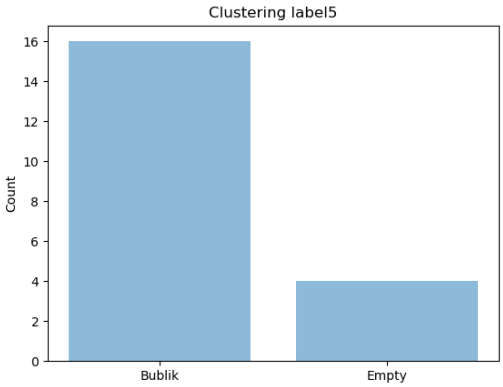
Spectral Clustering outcome summary

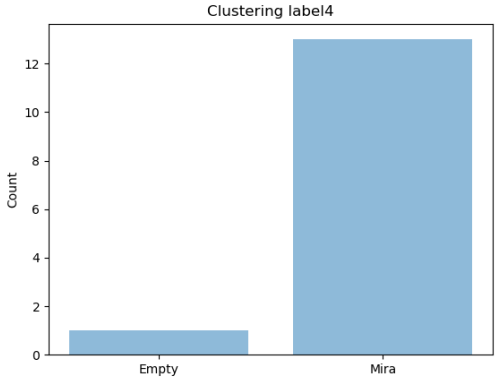
Another good result was given by the Spectral Clustering algorithm. The same type of bar charts is below, as previously.



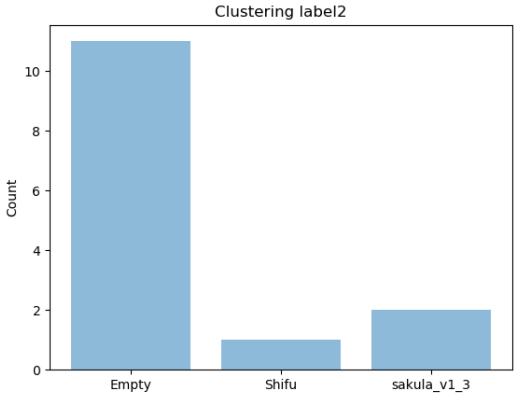


Cluster #1 and #3 are mainly containing only Shodi samples. So, this algorithm can also find the difference between the rest of the samples and the Shodi samples.

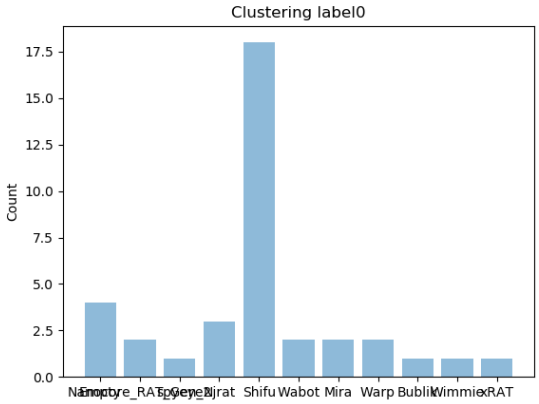




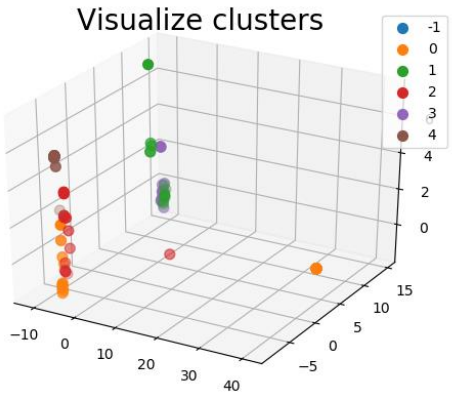
In cluster #5, we have most of the Bubik samples.
In cluster #4, we have most of the Mira samples.



Cluster #2 might suggest, that in the ‘Empty’ category, we have samples, which are more like each other, than anything else.



In cluster #0, we have almost all the Shifu samples, and everything else, which didn’t fit into any other cluster.



The last picture contains the 120 samples in 3D plotted, reduced the dimensions by the PCA algorithm.

In summary: this clustering did a perfect job clustering Shodi samples. Mira and Bubik samples are also separated, but with less confidence, and some other undefined samples are also mixed in.

Based on this clustering, we even might be able to perfect the initial tags of the data, because it seems like some undefined samples are very much resemble each other.

5. Summary

We mastered in practice the functions and the use of different clustering algorithms.

We also learned about the importance of data cleansing. The success of the clustering depends on the format of the input, so it is something that should be well considered beforehand.

Based on our findings, we can conclude which malware families can be more detectable based on the strings included in them.

However, it would have been interesting to include some files which were not malicious, to test if they are getting false positives based on this closeting, or not.

Our source code and documentation can be found in https://github.com/caspien6/isr_project.