# Multiple AI solutions to Malware detection classification

**Victor Coatalem**

1

## Contents

## 1. Preamble

This report purpose is to give a broader look at the AI project done for ESLR recruitment session 2019. I will go over design choices, results and my understanding of the different parts of the projects. If you are more interested in the nitty-gritty I would suggest you consult the documentation I wrote in the repository.

## 2. Project Overview

This part is there to give some context to the report, feel free to jump to Part 3 if you already know what is going on there.

### 2.1. Problem

Malware detection is an important problem to solve, and a challenging one at that since there does not seem to be one magical approach that solves all parts of the problem. If one was to find a good classification method, it is likely malware programmers can easily find a workaround anyway.

### 2.2. State of the Art

The most basic malware classification technique is signature-based, meaning each file is assigned a hash based on various features extracted from the file and stored in a table. Files that find themselves hashed similarly as known malwares will be classified as such. This technique, while effective when sorting most malwares, has serious shortcomings when it comes to new, unidentified threats as the features of the binary worth extracting will likely have changed, and most advanced malwares use polymorphism, changing their signature every iteration. In facts, the technology evolved so much that 97%[4] of the modern malwares use these mutation techniques and therefore signature based algorithm seems pretty lacking.

### 2.3. AI solutions

Evergoing learning is one of the core concepts of Machine Learning. It turns out this fits perfectly the needs of malware classification, which is too an ever evolving process. Analyzing a **lot** of program binaries with large arrays of features can establish a model of what a well-intentioned program looks like, and therefore deduct ill-intentioned programs[5]. There are multiple AI approaches to this problem; in this work we will be experimenting with basic KMeans algorithm, classifier, deep neural network and another KMeans approach using a brand new feature vector.

### 2.4. Dataset

We will be using EMBER[1] dataset in this experimentation. EMBER has been a staple in malware classification since its first iteration in 2017 and provides really good documentation, insights on its structure, as well as constant support. It also tries to handle the issue of constant malware evolution by having its training set composed of older binaries, and evaluation set focus on newer binaries to make sure the features extracted in older binaries are still relevant in newer ones.

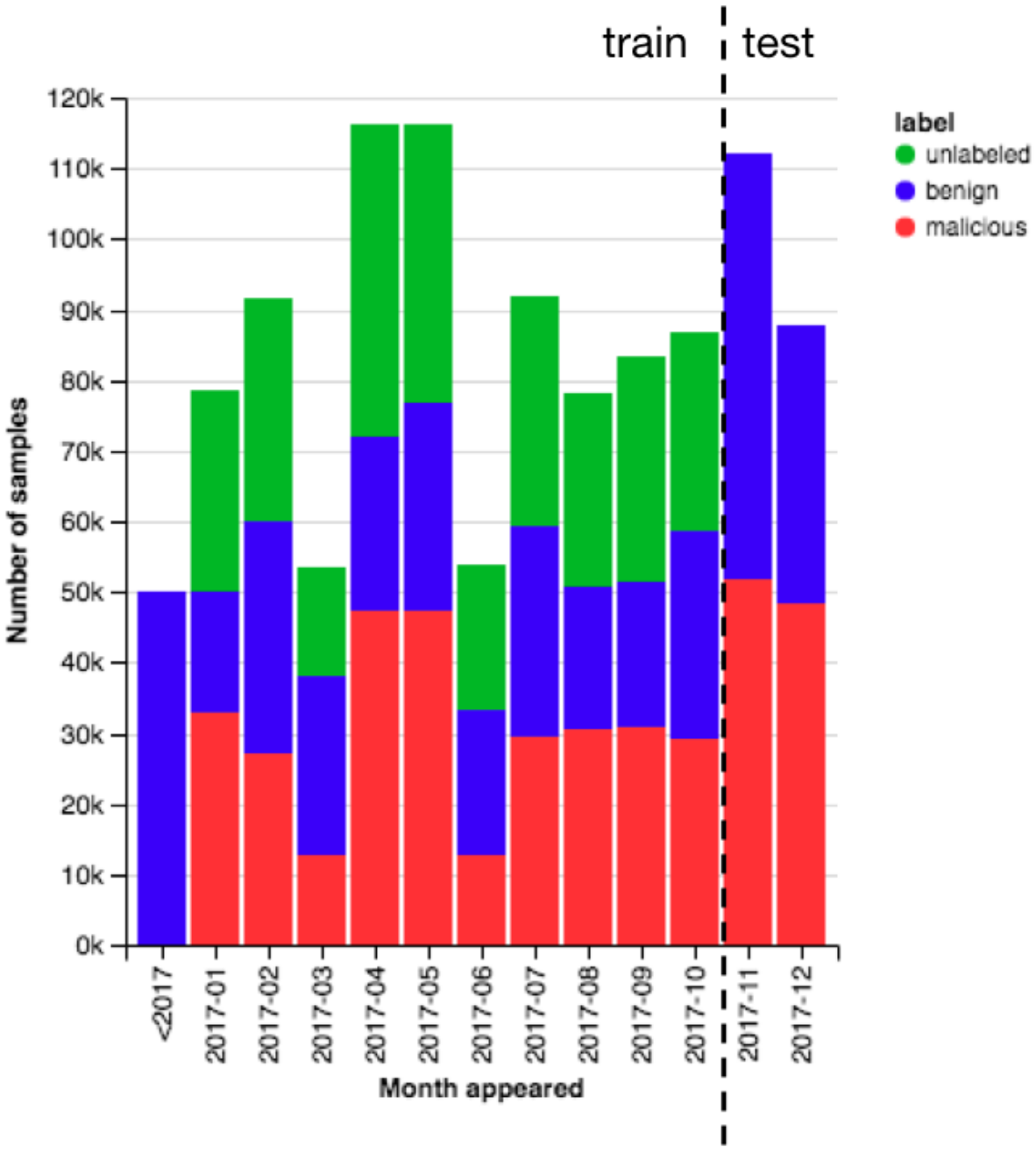


**Figure 1. Constitution of the EMBER dataset (2017 version)**

## 3. KMeans

The EMBER dataset provides us with unlabelled data, in an attempt to foster unsupervised approaches to machine learning which might have good results given the amount of different features to take into account. In this exercise we will focus on creating a KMeans with excellent optimization and at least decent accuracy.

### 3.1. Reference Examination

Our goal is to improve an already existing (poor) implementation of KMeans. For reference, here are the results obtained by launching the kmeans and its evaluation as stated in the README:

```
(env) xarang@saturn:~/try/ESLR_PROJECT_2019/kmeans$ ./kmeans 3 20 1.0 2351 900000 ~/ESLR_PROJECT_2019/ember/Xtrain.dat out.dat
Start Kmeans on /home/xarang/ESLR_PROJECT_2019/ember/Xtrain.dat datafile [K = 3, dim = 2351, nbVec = 900000]
Iteration: 1, Time: 10.401899, Error: 306825568.000000
Iteration: 2, Time: 10.302044, Error: 185975616.000000
Iteration: 3, Time: 10.178803, Error: 199004704.000000
Iteration: 4, Time: 9.764349, Error: 209908720.000000
Iteration: 5, Time: 10.396807, Error: 200758960.000000
Iteration: 6, Time: 10.414660, Error: 164408256.000000
Iteration: 7, Time: 10.313691, Error: 161070960.000000
Iteration: 8, Time: 10.386433, Error: 160309600.000000
Iteration: 9, Time: 9.829622, Error: 160116560.000000
Iteration: 10, Time: 9.780211, Error: 160062496.000000
Iteration: 11, Time: 9.778750, Error: 160051200.000000
Iteration: 12, Time: 9.614625, Error: 160045984.000000
Iteration: 13, Time: 9.691567, Error: 160045216.000000
Iteration: 14, Time: 9.829819, Error: 160044944.000000
Iteration: 15, Time: 9.770797, Error: 160044944.000000
(env) xarang@saturn:~/try/ESLR_PROJECT_2019/kmeans$ python3 ../python/eval.py out.dat ~/ESLR_PROJECT_2019/ember/Ytrain.dat
[ 0 ,  -1 ,  1 ]
Accuracy:  0.35383888888888887
Precision:  0.4204136782927333
Recall:  0.35383888888888887
Confusion Matrix:  [[ 44293 248730   6977]
 [ 35305 259923   4772]
 [ 40591 245170  14239]]
(env) xarang@saturn:~/try/ESLR_PROJECT_2019/kmeans$
```

**Figure 2. KMeans evaluation of reference code**

We can see that the process takes way too much time. This is both due to the algorithm having a lot of redundant computations, space for multi-threading as well as vectorization, and the fact that even though the error started converging at Iteration 6, the process kept going for 10 more iterations for very little gain. As far as the accuracy results are concerned, we can see on the confusion matrix that results for labels 0 and -1 have a pretty even distribution among the 3 labels (meaning vectors of labels 0 and -1 were predicted to be part of labels 0, 1 and 2 and about even amount of times - *about what you would get on average if you assigned a cluster randomly, so not a great success* - On the other hand we can see that data vectors of label 1 - *malicious* - are assigned to their right cluster at a rate of $14239/6977 \approx 2.04$, which is a pretty good result for an algorithm as basic as KMeans.

### 3.2. Optimization Axes

#### 3.2.1. Limiting unnecessary computations

The first way I found to reduce computations is to mark some vectors as 'rightly placed' when their error is sufficiently small, so that we do not compute them afterwards.

**Algorithm 1** KMeans Loop with Mark

**Require:** $E > 0$ (tolerated error threshold)
**Require:** $mark[nbVectors]$ (array of vectors to skip, memset'ed to 0)
**for** each Vector **do**
  **if** $mark[Vector] \neq 1$ **then**
    assign Vector to its closest cluster
    **if** distance to closest cluster $< E$ **then**
      $mark[Vector] = 1$
    **end if**
  **end if**
**end for**

This simple trick allows us to greatly limit the amount of computations in later iterations where only a few vectors would change cluster anyway. Of course some vectors will be wrongfully placed in a cluster, but the time gain is so extreme that we deem it worth the little loss in accuracy.

Setting some vectors centroids in stone like that also allows us to simplify mean vector computation. This computation is indeed pretty expensive as it require us to go through all our vectors ($N = 900000$). In the reference code, I found that every instance of the Mean Vector computation took about 2 seconds of time, which over multiple iterations becomes a hefty amount. Therefore I decided to change the way mean vectors are computed. To go back to our previous algorithm; Marked vectors are now added to the mean vector, which we simply divide by the amount of marked vectors in concerned cluster.

This basically removed Mean Vector computation as we only need to make a copy and a division by $card[centroid]$ to get the mean vector of each centroid (no more $nbVector$ traversal). As a result The time it takes to compute Mean Vector values dropped from $\approx 2.0$ seconds to $\approx 0.05$ seconds, which easily nets a $10$ seconds increase in most KMeans executions.

**Algorithm 2** KMEANS Loop with Mark and Mean Vector computation

**Require:** $E > 0$ (tolerated error threshold)
**Require:** $mark[nbVectors]$ (array of vectors to skip, memset'ed to 0)
**Require:** $meanVector[nbCentroids]$ (mean vector initial values for each centroid)
**Require:** $card[nbCentroids]$ (amount of vector marked in each centroid)

```
for each Vector do
  if mark[Vector] ≠ 1 then
    assign Vector to its closest cluster
    if distance to closest cluster < E then
      mark[Vector] = 1
      meanVector[closestCluster] += Vector
      card[closestCluster] += 1
    end if
  end if
end for
```

### 3.2.2. Paralleling the main loops

The loops in our program that benefit paralellisation the most are loops that iterate through our 900000 vectors. As a result the Algorithm 2 seen in previous part runs on multiple threads thanks to OpenMP pragma directives, resulting in an obvious gain in performances.

### 3.2.3. Optimizing the main computations

Our heuristic function for computing distance between 2 vectors is the euclidian distance, therefore this function needs to be as optimized as possible. On our first iteration it is called $\approx 1758000$ times, and this number only starts dropping when the amount of marked vectors goes up. We therefore use vectorization function of Intel Intrisics (AVX and AVX2) to quicken each computation of distance as much as possible

```
/*
** euclidian distance between 2 vectors of dimension dim
*/
inline double distance(float *vec1, float *vec2, unsigned dim)
{
    double dist = 0;
    unsigned vector_size = 8;
    __m256i index = _mm256_set_epi32(0, 1, 2, 3, 4, 5, 6, 7);
    for (unsigned i = 0; i < dim / vector_size; i++)
    {
        unsigned base_index = i * vector_size;
        __m256 arr1 = _mm256_i32gather_ps(vec1 + base_index, index,
            sizeof(float));
        __m256 arr2 = _mm256_i32gather_ps(vec2 + base_index, index,
            sizeof(float));

        //vectorization of double d = vec[i] - vec2[i]
        __m256 sub_arr = _mm256_sub_ps(arr1, arr2);
        //vectorization of double d = vec[i] * vec[i]
        __m256 mul_arr = _mm256_mul_ps(sub_arr, sub_arr);
        double sum = mul_arr[0] + mul_arr[1]
            +   mul_arr[2] + mul_arr[3]
            +   mul_arr[4] + mul_arr[5]
            +   mul_arr[6] + mul_arr[7];
        dist += sum;
    }
    return sqrt(dist);
}
```

### 3.2.4. Better Initialisation of centroids

**KMeans++** Finally, I thought a large part of the amelioration I could bring to the given source code was a proper initialisation method. Basically, given the way I mark vectors that are sufficiently close to a centroid, the base position of these centroids is very important in order for my results not to be skewed by bad initialisation. Hence, I utilized a fast version of the centroid initialisation algorithm known as *K-means++*[2]. In short: we select a few candidates to be our centroids and evaluate their potential, which we evaluate as their squared distances to a subset of points. Once we have all our candidates potentials found, we take one with the median potential (meaning it is at a medium distance to all other points) and proceed finding other candidates, now adding our found candidate to the subset of points to compute distances with. This method will foster a disposition of initial centroids with an even distance to other points, thus limiting risks of one centroids absorbing all the points to itself.

**Another solution** After implementing this initialisation algorithm, I found it simply was not fit four our 2-cluster problem. Indeed, we using KMeans++, the 2 first centroids are more or less placed randomly, and only the further centroids placed benefit from the algorithm. As a result this complex and usually performant algorithm simply is not adapted to our case.

I therefore came up with another solution that is more adapted to a $k = 2$ KMeans algorithm. The idea is to still select a small subset of points and select a few potential candidates for being a cluster. We then find pairs of candidates which are at about equal distance to all the points in the subset, and evaluate their potential by looking at the difference between their distance to the subset and the distance between the two of them. The idea is to find pairs that are at equal distance of all points in a subset, while not being too close to eachother.

**Algorithm 3** Initialisation Loop

**Require:** $k\ candidates$
**Require:** $subset\ size = n$
**Require:** $distVector[n]$ (mean distances between candidates and subset of points)
**Require:** $distMatrix[k][k]$ (distances between each candidate)
  $pairs \leftarrow pairs(i, j)$ with $distVector[i] \approx distVector[j]$
  **for** each pair (a, b) **do**
    $potential \leftarrow |distVector[a] - distMatrix[a][b]|$
  **end for**
  choose pair of centroids with the lowest potential

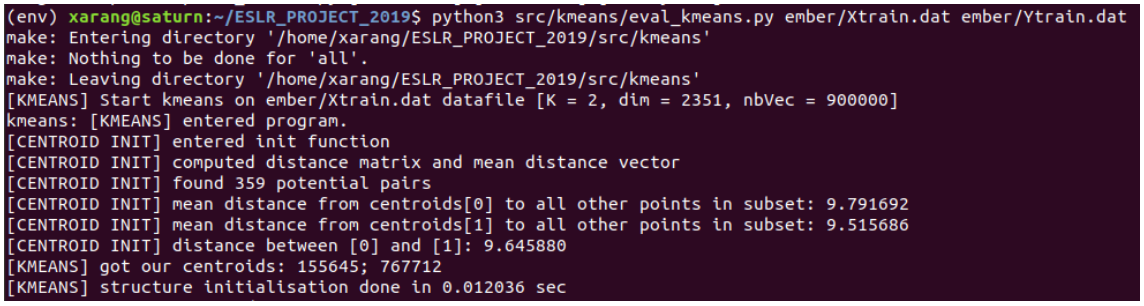


**Figure 3. KMeans init output**

### 3.3. Results

Unfortunately our results with all these ameliorations did not match our expectations. If from an optimization point of view, we reached better computation time ($\approx$ 6 seconds), our results in term of accuracy were too close to those of the references (not great, to say the least). Later on in this report we will see another attempt at KMeans, with better results this time around.

## 4. SKlearn Classifier

In this section, we will propose an implementation of a Classifier using low computational power as well as low RAM. Our Classifier will assign to each input data vector a probability distribution vector $[p0, p1]$ where $p0$ is the probability of being *Benign*, and $p1$ the probability of being *Malicious*.

**Unlabelled Output** We could use our probability vector so that our classifier gives us a probability of certainty in its predictions, and vectors that could not be classified with enough confidence in a category. **While in the end we did not need to use this in our computations - since our results were already great -** this is a possibility to explore for further improvements of our classifier. For instance, a $[0.4, 0.6]$ prediction should not have the same weight as a $[0.0, 1.0]$ prediction.

For now though, we will simply strip unlabelled data prior to execution (as our DNN will too need its unlabelled data removed) and a $[0.4, 0.6]$ distribution will be treated same as a $[0.0, 1.0]$.

**Objectives** We will be aiming at a compromise between RAM/CPU usage and computation time.

### 4.1. Data Pre-Treatment

The main objectives of our data pre-treatment are:

- Remove unlabelled data *(done higher up in the execution pipe)*
- Reduce vector dimension

**PCA** To reduce the amount of features to keep only the most important ones. For that purposes, we choose a subset of vectors in our training data holding a similar amount of benign and malicious data. We then incrementally compute our PCA batch by batch to keep RAM usage as limited as possible. Once our PCA is computed on our subset, we apply it to our entire training data and validation data. Separation of unlabelled data from labelled data is done batch by batch to keep the numpy arrays mapped in memory (not all allocated at once).

```
# Returns a generator that yields chunks of size 'chunk_size'
def chunks(data: np.array, labels: np.array, chunk_size: int):
    for i in range(0, len(data), chunk_size):
        yield ( data[i:i + chunk_size, :], labels[i:i + chunk_size] )
# Computes PCA for our training data, reducing the dimensions of our
    data
# PCA_NB_COMPONENTS as the amount of dimensions to scale our vector
    into.
# The PCA is computed by increments to avoid RAM overusage
# Returns all the data, unlabelled data removed and PCA-reduced
def data_pre_treatment(training_data, validation_data, training_labels,
    validation_labels):

    # subset of data to fit our pca on
    sample_indexes = np.random.choice(range(len(training_data)),
        100000, replace=False)
    subset = training_data[sample_indexes]
    subset_indexes = training_labels[sample_indexes]
    subset, subset_indexes = remove_unlabelled_data(subset,
        subset_indexes)
    pca = IncrementalPCA(n_components=PCA_NB_COMPONENTS, batch_size =
        100)
    pca = pca.fit(subset)

    # transform all validation data with the pca we just compute
    new_validation_data = pca.transform(validation_data)
    # We process our training data by smaller chunks to not overload
       RAM usage
    new_training_data = np.vstack([ \
            pca.transform(remove_unlabelled_data(chunk_tuple[0],
                chunk_tuple[1])[0]) \
            for chunk_tuple in chunks(training_data, training_labels,
                VECTOR_CHUNK_SIZE) \
            ])
    new_training_labels = training_labels[training_labels != -1]

    return new_training_data, new_validation_data, new_training_labels,
        validation_labels
```

### 4.2. Classification Algorithm

Our classifier will use the **K-Nearest Neighbors classifier** provided by SKlearn, which yields good results when operating on 2 labels datasets.

### 4.3. Results

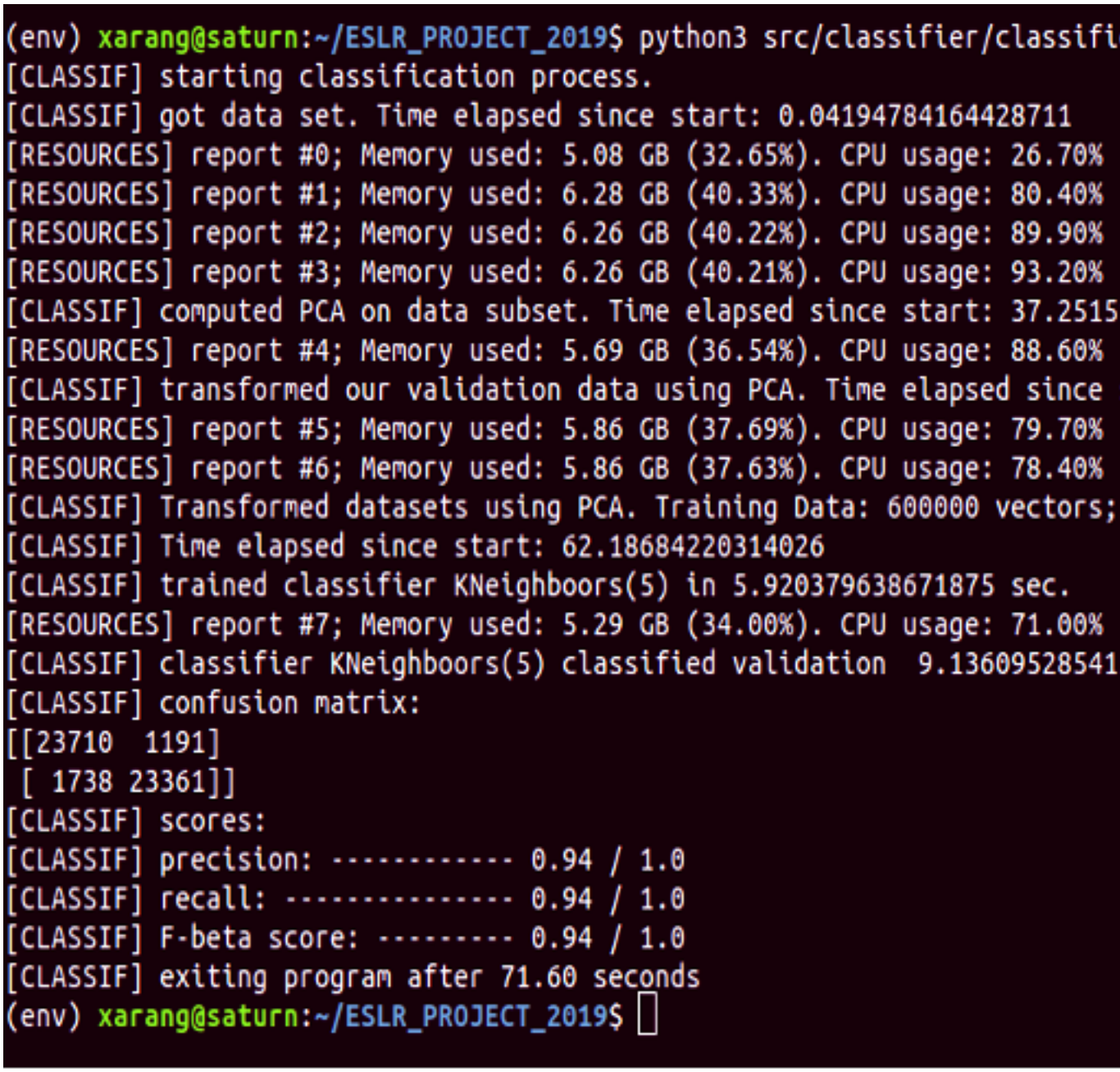


**Figure 4. Classifier Output. While running our classifier, we run a background process that outputs every 10 seconds informations about RAM and CPU usage.**

As we can see in the figure, our classifier make use of limited ressources yet manages to train in decent time and outputs satisfying evaluation scores; this method seems

successful.

## 5. Deep Neural Network

In this section, we will propose a DNN structure to classify our data using Keras.

### 5.1. DNN structure

- Input layer $n = 2351$
- Normalization layer $n = 1024$
- Hidden layers $n = 1024$ to $n = 64$
- Output layer $n = 1$

Our units are activated through Rectified Linear Unit which is generally a good activation function when dealing with classification problems [6]. The output is a single, sigmoid-activated unit that translates the label of processed data: $0$ for benign, $1$ for malicious.

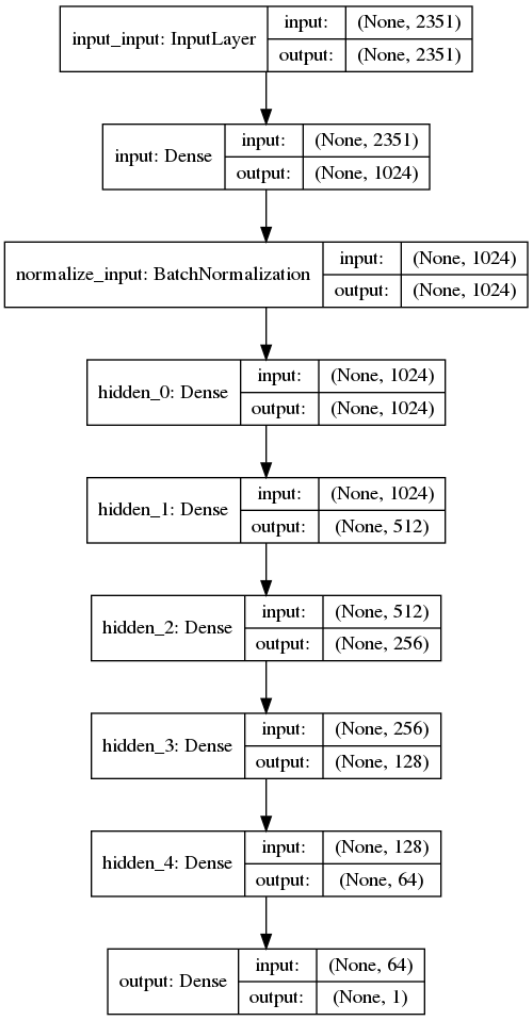


**Figure 5. Network Model. I try to keep a good compromise between computation time and result quality.**

### 5.2. Training method

#### 5.2.1. Loss function

Most binary classification problems use **binary crossentropy** as their loss function [3]. As our problem falls in this category *(benign / malicious)*, we decide to use it too.

#### 5.2.2. Optimizer

Our loss is minimised through **AdaGrad**, an improved stochastic gradient method that adapts its learning rate to each parameter.

### 5.3. Results

#### 5.3.1. v1

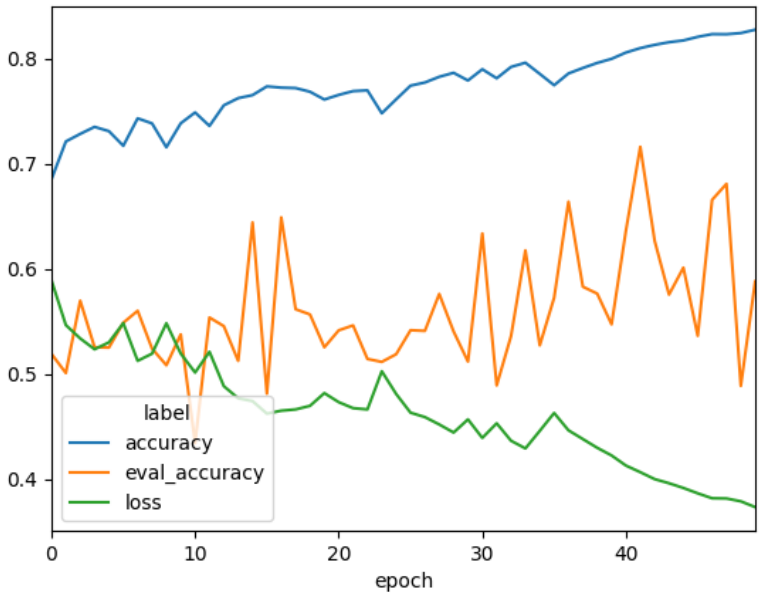


**Figure 6. epochs:** 50**; learning rate:** 0.1

Even though the loss function is minimized pretty well, evaluation accuracy remains pretty randomly valued $\approx 0.5$.

### 5.3.2. v3

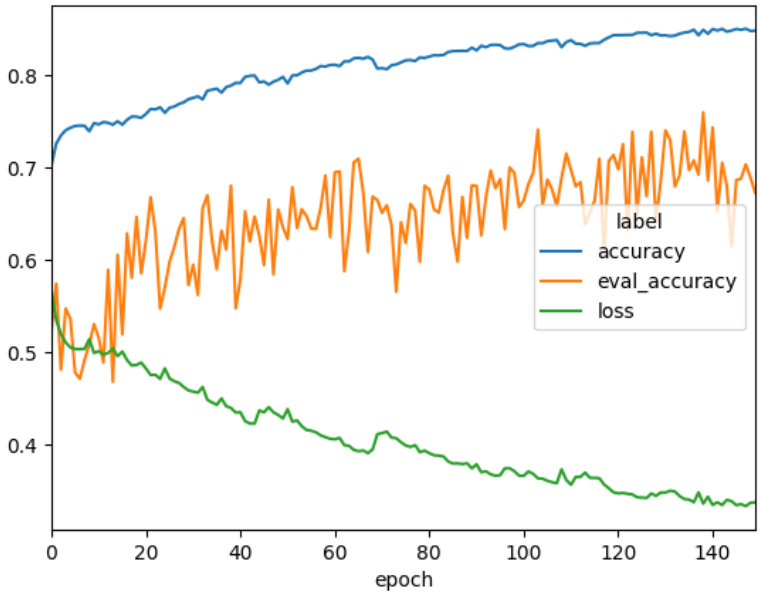


**Figure 7. epochs:** $150$**; learning rate:** $0.001$

Increasing the amount of epochs as well as diving learning rate by $10$ allowed evaluation accuracy to start getting better on average, now consistently over the $0.5$ threshold of viability, and averaging $\approx 0.65$.

### 5.3.3. v4

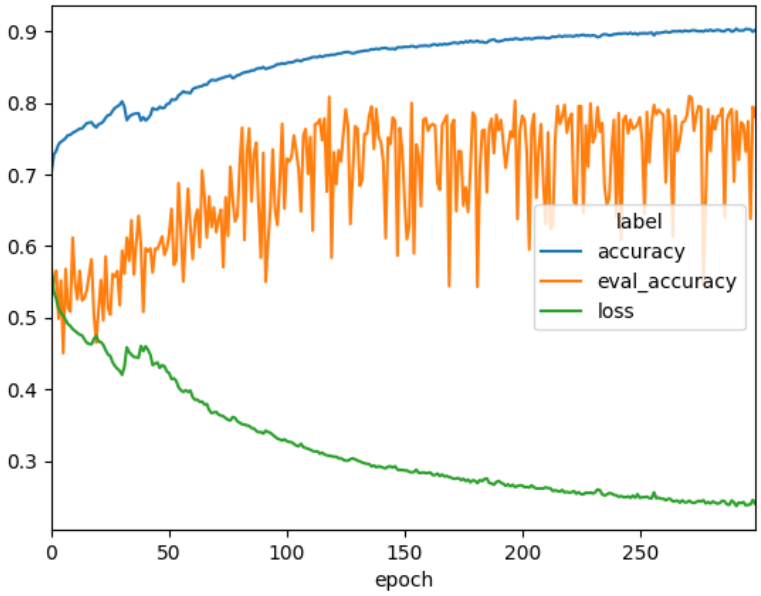


**Figure 8. epochs:** 300**; learning rate:** 0.0007

We can see that even though accuracy on training set quickly converges in high value, it takes more time for evaluation accuracy to get good values reliably. In the end we obtain an evaluation accuracy of $\approx 0.78$.

## 6. KMeans 2

### 6.1. Preamble

While trying to optimize KMeans, one thing that occured to me was the fact that our heuristic method - *euclidean distance* - was simply not adapted to our dataset. To check this I tried to visualize our dataset in 2 dimensions only. For that purpose, I computed a PCA with $ncomponents = 2$, thus bringing our data down to 2 features through combination of features.

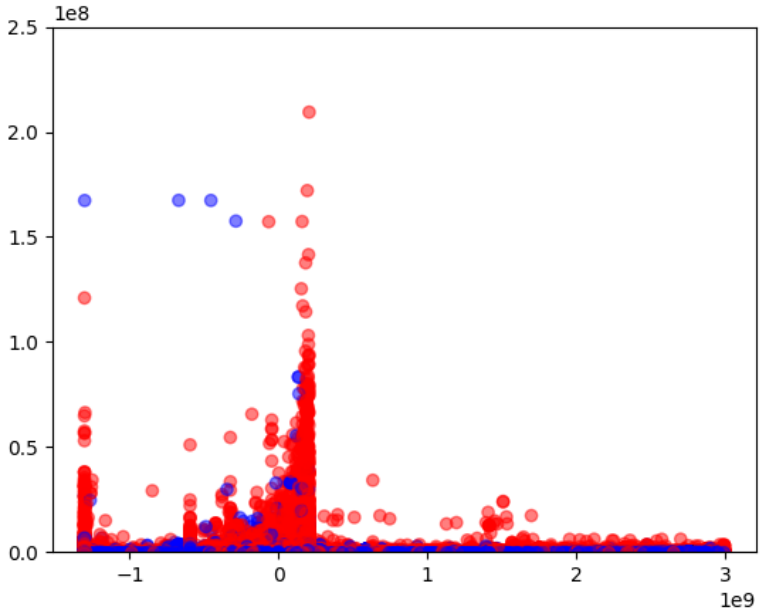


**Figure 9. 2D visualization of our dataset**

In this graph, we can see that all blue plots are stacked on the $y = 0$ axis, whereas the red dots are for the most part aligned on $x = 0$. Hence, euclidean distance simply can not be the best way to cluster our data.

### 6.2. Solution

**To tackle this problem, I decided to try to create a new feature vector, that is more fitted to euclidean distance**. The idea was to identify features in our vector with the largest standard deviation between data of different labels. By using Sklearn libraries as well as calculus spreadsheets, I obtained the list of the 128 features in our vector that are on average, the most different between vectors of label 0 and vectors of label 1. These 128 features hence allow us to build a new feature vector, thus:

- Removing dimensions that could confuse our KMeans in wrong directions
- **Greatly** improving performances by reducing the amount of computations for each $distance$ call from 2351 to 128

$Distance$ being a cheaper function overhaul allows us to be not as restricted in its use. The tricks detailed in part 3.2, which improved performance at the cost of *some* accuracy are no longer needed. The distance function now only adds up dimensions in our vector that are part of the 128 selected indexes. For each of these features, I also obtained the absolute maximum value for that feature over our entire dataset, **thus allowing normalization of our features**. Normalization allows us to put all dimensions on a same level of importance, whereas before features that naturally took larger values overshadowed any smaller-valued features. By bringing all features to a $[-1, +1]$ range we are free to apply our own weights, thus giving some key features more weight in euclidean distance computation.

Our feature vector is sorted by order or importance (vector[0] is the more important feature, vector[128] the least) and have a weight vector associated. Our weight distribution is given by the function $f(x) = log(128) - log(x)$, which gives a greater importance to the few first features in our vector, and then gradually less to other features.

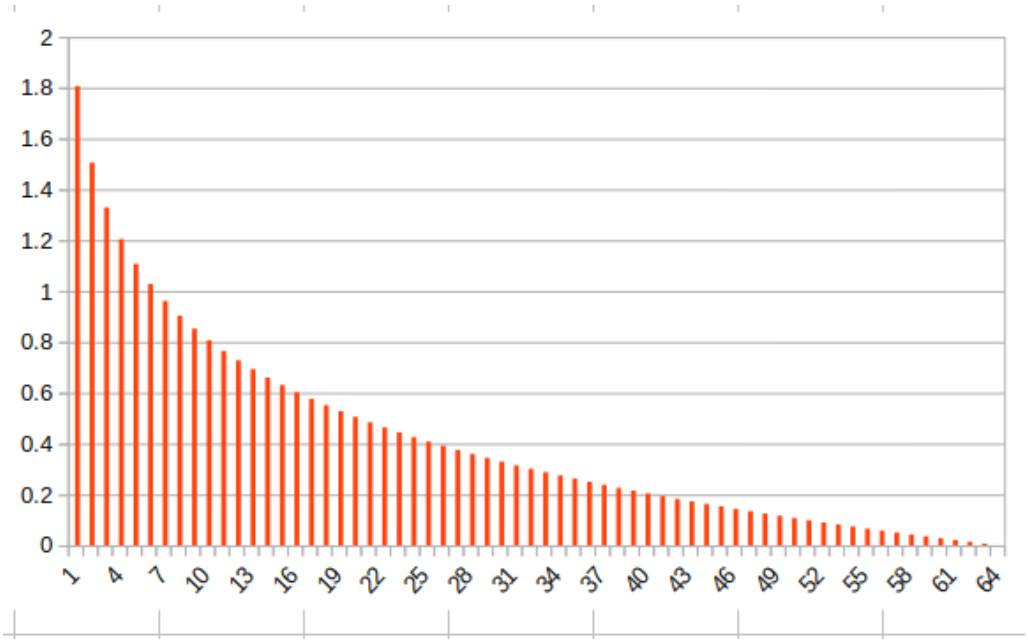


**Figure 10. Our Weight distribution function for** $nbFeatures = 64$

### 6.2.1. New Heuristic

```
#include <immintrin.h>

#define MASK_SIZE 64
//mask: indexes of our vector worth computing (features)
size_t mask[MASK_SIZE] = { 637,621,618,655, ... ,680,860,375 };
//max: max absolute values associated with each feature
// (used for normalization of features)
double max[MASK_SIZE] = { 1,1,1,1,1,1,2,1, ... ,7,0.206387,2,255,4,0.323169,0.224523,0.70557 };
//weights: values to multiply important features
// with (weight[x] = log(MASK_SIZE) - log(x))
double weights[MASK_SIZE] = { 1.80617997398389,1.50514997831991, ... ,0.013788284485633,0.006839424530305 };

//our new feature vector enhanced, computation lighter, kmeans higher scoring heuristic function
double distance(float *vec1, float *vec2)
{
    double dist = 0;
    for (unsigned i = 0; i < MASK_SIZE / 8; i++)
    {
        __m256 arr1 = _mm256_set_ps(
            vec1[mask[i * 8 + 0]], vec1[mask[i * 8 + 1]],
            vec1[mask[i * 8 + 2]], vec1[mask[i * 8 + 3]],
            vec1[mask[i * 8 + 4]], vec1[mask[i * 8 + 5]],
            vec1[mask[i * 8 + 6]], vec1[mask[i * 8 + 7]]);
        __m256 arr2 = _mm256_set_ps(
            vec2[mask[i * 8 + 0]], vec2[mask[i * 8 + 1]],
            vec2[mask[i * 8 + 2]], vec2[mask[i * 8 + 3]],
            vec2[mask[i * 8 + 4]], vec2[mask[i * 8 + 5]],
            vec2[mask[i * 8 + 6]], vec2[mask[i * 8 + 7]]);
        __m256 max_mask = _mm256_set_ps(
            max[i * 8 + 0], max[i * 8 + 1],
            max[i * 8 + 2], max[i * 8 + 3],
            max[i * 8 + 4], max[i * 8 + 5],
            max[i * 8 + 6], max[i * 8 + 7]);
        __m256 weights_mask = _mm256_set_ps(
            weights[i * 8 + 0], weights[i * 8 + 1],
            weights[i * 8 + 2], weights[i * 8 + 3],
            weights[i * 8 + 4], weights[i * 8 + 5],
            weights[i * 8 + 6], weights[i * 8 + 7]);
        // normalize data
        arr1 = _mm256_div_ps(arr1, max_mask);
        arr2 = _mm256_div_ps(arr2, max_mask);
        // multiply by weight
        arr1 = _mm256_mul_ps(arr1, weights_mask);
        arr2 = _mm256_mul_ps(arr2, weights_mask);
        // get differences
        __m256 sub_arr = _mm256_sub_ps(arr1, arr2);
        // square the differences
        __m256 mul_arr = _mm256_mul_ps(sub_arr, sub_arr);
        // sum up the differences
        double sum = mul_arr[0] + mul_arr[1]
            +   mul_arr[2] + mul_arr[3]
            +   mul_arr[4] + mul_arr[5]
            +   mul_arr[6] + mul_arr[7];
        dist += sum;
    }
    return dist;
}
```

### 6.3. Results

#### 6.3.1. Evaluation Method

Our KMeans evaluation function focuses on 2 main aspects:

- Cluster Repartition
- Cluster Uniformity

**Cluster Repartition** represents the fact that a similar amount of vectors are outputted in each cluster. As our dataset has $300000$ malicious and $300000$ benign values, both cluster *ideally* have 300000 vectors each. Hence, our **Cluster Repartition** score is computed as such:

$$ClusterRepartition = min(card[0], card[1])/max(card[0], card[1]) \quad (1)$$

$card[0]$ *and* $card[1]$ *being the amount of vectors outputted in clusters* $0$ *and* $1$

**Cluster Uniformity** represents the fact that a cluster holds values of same labels. Ideally, all values in each cluster should be of the same label. If in cluster $1$ we output $15000$ vectors of label $0$ and $30000$ vectors of label $1$, we have a vector of repartition $[0.33 : 0.66]$, and the cluster uniformity score for this cluster is based on the minimal euclidean distance between $[0.33 : 0.66]$ and one of the 2 *ideal* vectors $v1 = [0 : 1]$ and $v2 = [1 : 0]$. More specifically, the formula we use is as follow:

$$ClusterUniformity(C) = 1 - min(euclid(card[C], v1), euclid(card[C], v2)) \quad (2)$$

Our KMeans is computed $10000$ times for each parameter setup we wish to try out. The values specified above are averaged out, and collected at the end, as well as mean computation time which is another really important metric to keep track of.

### 6.4. Observation

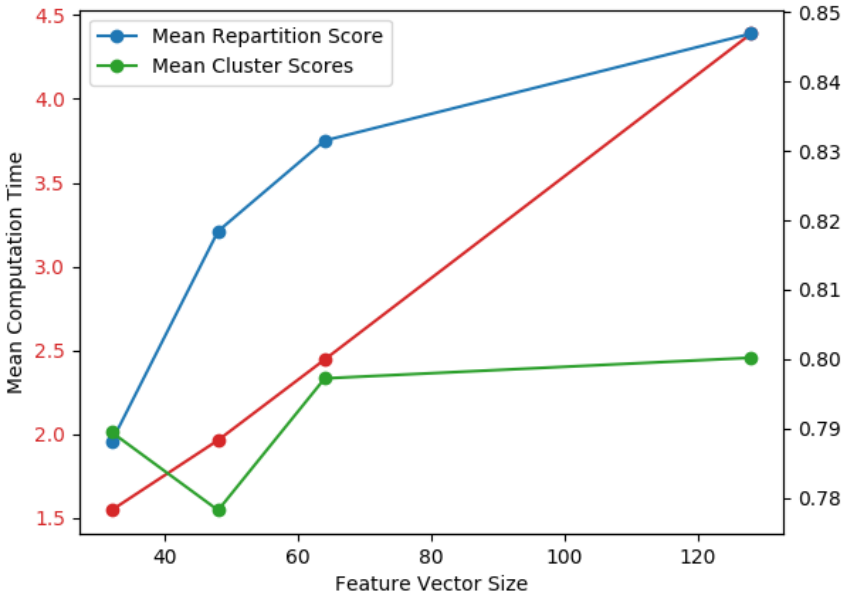


**Figure 11. Results for our 4 vector size experimentations**

Our goal is to find a good compromise between computation time and results. We can see that below a size of 64, our cluster scores start dropping quickly. $size = 64$ therefore looked like the best compromise, and it is the value used in our code. Cluster Score $= 0.8$ and Repartition Score $= 0.845$ are *excellent* results comparing to reference, or even our previous iteration of KMeans.

### 6.5. Comparisons with previous states of our KMeans

#### 6.5.1. Reference

- Mean Computation Time: 88.71 seconds
- Mean Cluster Score: 0.0632851
- Mean Repartition Score: 0.1495542

#### 6.5.2. KMeans1

- Mean Computation Time: 6.01 seconds
- Mean Cluster Score: 0.2152014
- Mean Repartition Score: 0.1902532

### 6.5.3. KMeans2

- Mean Computation Time: 2.46 seconds
- Mean Cluster Score: 0.7994581
- Mean Repartition Score: 0.8431287

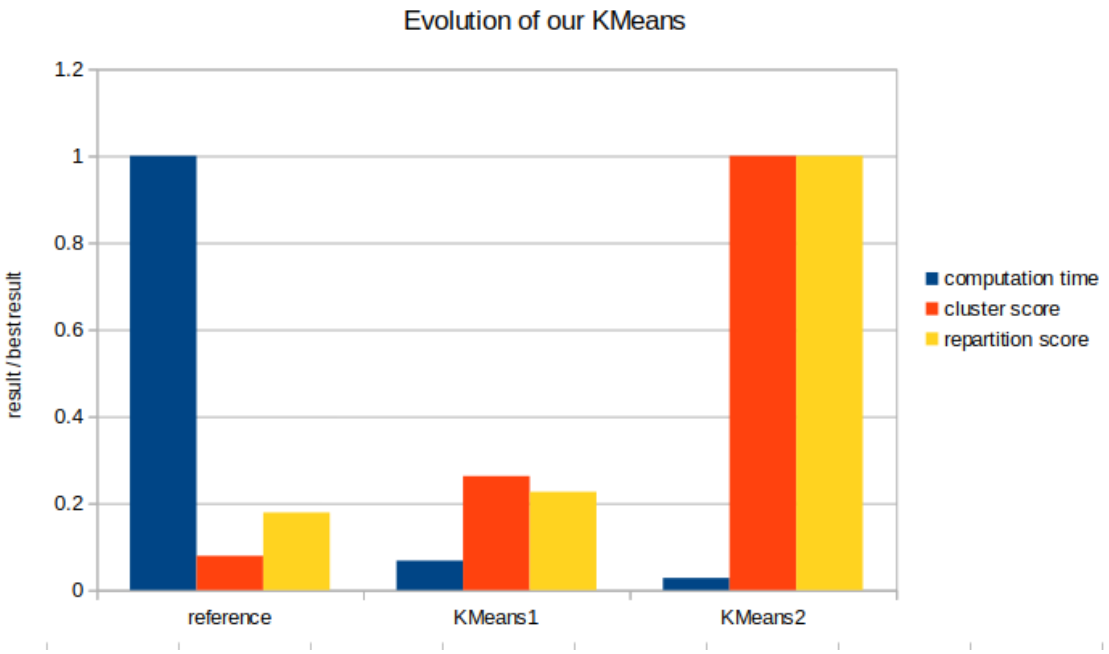


**Figure 12. Our results matched against eachother**

Which goes to show how our KMeans2 yields both good computation time optimization (although it is possible to be even quicker), and more importantly actual clustering of our data, whereas initial KMeans more or less placed vectors in random clusters.

## Sources


[1] H. S. Anderson and P. Roth. "EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models". In: *ArXiv e-prints* (Apr. 2018). arXiv: 1804.04637 [cs.CR].

[2] Sergei Vassilvitskii David Arthur. "The Advantages of Careful Seeding". In: *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics Philadelphia, PA, USA* (2007).

[3] Daniel Godoy. "Understanding binary cross-entropy / log loss: a visual explanation". In: *https://towardsdatascience.com* (2018).

[4] Koen Van Impe. "How to choose the right malware classification scheme to improve incident response". In: *securityintelligence.com* (2018).

[5] Dr. Vigna Giovanni. "How AI will help in the fight against malware". In: *Techbeacon.com* (2018).

[6] Yoshua Bengio Xavier Glorot Antoine Bordes. "Deep sparse rectifier neural networks". In: (2011).