ESLR Project Recruitment Project: Endgame Malware Project for Research

NoneOfAllOfTheAbove

Arielle LEVI

Théo LEPAGE

Quoc Duong NGUYEN

Brian BANG

December 2019 - January 2020

Contents

1 Introduction

This report will compile what has been done for the ESLR recruitment project, **EMBER** (Endgame Malware BEnchmark for Research), by the group NoneOfAllOfTheAbove. The project started on December 19th, 2019 and was due on January 4rd, 2020.

Such project aims at testing our ability to look for information and use it properly, as well as approaching machine learning and parallel programming techniques. Parallel programming techniques are implemented using OpenMP which is a C/C++ parallel programming library and machine learning ones will be seen through Scikit-learn and Keras libraries in Python.

The full implementation, including some technical details, are available on the project repository and on Google Colab notebooks. Links to these resources are listed in the conclusion.

2 Team presentation

NoneOfAllOfTheAbove is a group composed of four members:

- Arielle LEVI <arielle.levi@epita.fr>
- Theo LEPAGE <theo.lepage@epita.fr>
- Quoc Duong NGUYEN <quoc-duong.nguyen@epita.fr>
- Brian BANG <brian.bang@epita.fr>

We are all EPITA students from the BING group and worked together on the well-known project *42sh*. We are applying as a team as the *42sh* project went well and we believe that we will be able to carry out other projects together.

This name, "NoneOfAllOfTheAbove", refers to the frustration that we all felt during MCQs in front of an ambiguous question and especially before checking the perilous well known last answer "All of the above are correct".

3 The k-means algorithm

3.1 What is k-means?

To make it simple, k-means is a *clustering* method allowing one to determine how similar are samples of a given study, by projecting them onto an n-dimensional space, n being the number of features that are considered relevant for the study. The goal is to group these samples into k-clusters.

Note: k can be either specified or not, from what we have learned some techniques allow one to determine how many clusters best fit the dataset.

Our objective was to optimize computations i.e removing redundancy and splitting the workload in smaller ones without being less accurate than the provided k-means which had a 56% accuracy.

3.2 Lloyd's implementation

Lloyd's implementation is some kind of a naive way to implement k-means. It consists of an *initialization* step which assigns vectors to a cluster randomly and then two main states: the *assignment* step and the *update* step.

The *assignment* step Assign each sample to the cluster that is the nearest by comparing the Euclidean distance between the sample and the centroid of each cluster. The chosen cluster is the one which has its centroid closest to the sample.

The *update* step Recompute the cluster centers with the new set of points that has been obtained at *assignment* step. The last two are repeated until the center of each cluster converges, or the number of iteration specified is reached.

3.3 Hamerly's algorithm

Some of the calculations were redundant, especially the distances computations between each vectors and each centers. Therefore, we decided to implement Hamerly's k-means algorithm which allows avoiding going through the inner loop i.e *assignment* and *update* steps in specific cases. We chose to implement Hamerly's algorithm over Elkan's algorithm even if it works better on higher dimension vectors, which is our case. However, Hamerly's algorithm is well documented and is easy to grasp facilitating both the comprehension and the debugging part.

Hamerly's implementation is based on Elkan's implementation, which uses the triangle inequality to avoid many distance computations when recomputing the assignment vectors between two iterations.

We used the pseudo-code given in the Hamerly's research paper to implement it. Nonetheless, we noticed that the given algorithm was not sufficient to create a working model. We had then to find other sources, including the GitHub repository of Hamerly which shows some concrete usage of the given pseudo-code. Thanks to this change in our k-means program, we reduced our execution time to approximately one second.

3.4 A new initialization method: k-means++

k-means++ is an initialization algorithm that takes place before the actual k-means algorithm. Our previous initialization method was to randomly generate clusters and putting every vector in the first cluster. k-means++ tries to choose centers that are at the same time random and are as far away of each other as possible.

First, we choose one random vector as our new center. Then we compute all the distances between each vector and this new center, and choose the one with the highest distance from this center, to be the second center. This last step is repeated until reaching the wanted number of centers. The goal of this algorithm is to take centers very far from each others to ensure more or less that each of those centers belong to a different cluster. By doing this, we significantly start the chosen k-mean algorithm with centers that are already close to their final destinations. The number of iterations required before converging is then drastically reduced.

We made some changes to avoid recomputing the first iteration in the Hamerly's k-means because the assignment vectors just is already filled completely. The results were significant as it made our program deterministic. It converges every single time with the same accuracy (around 60%) and the same number of iterations. It made us gain at least eight iterations and the execution time of the program was significantly reduced.

Currently, our k-means uses k-means++ but it is easy to go back to the previous initialization method. Is it noteworthy that we made our best time on the leaderboard without k-means++.

3.5 Parallelization with OpenMP

OpenMP is a library providing tools for multiprocessing purposes. As k (the number of clusters) is negligible before the number of samples s, in our work it was more interesting to split the workload of the iterations on s than on k.

Implementing multi-threading was done following the steps below (please refer to the appendix on page 17):

1. Allocate an array $change\_cluster\_t$, $centroids\_sum\_t$ and $centroids\_count\_t$ for each thread (lines 8-11).

2. Tell OpenMP to split the workload of the main for loop among threads (line 14).
3. If the cluster has changed, reduce $centroids\_sum\_t$ and $centroids\_count\_t$ by adding their value into their respective array in the state (lines 34-47).
4. Use the atomic directive of OpenMP to specify that these operations need to be thread-safe to avoid race conditions (lines 38 and 43).
5. Use the reduction directive of OpenMP to do the reduction of $change\_cluster$ as it is a simple boolean.
6. Free thread relative data (lines 50-53).

Since some threads may be considered “lucky” and would not need to go through the inner loop with Hamerly’s implementation it may finish early and just wait for the other threads to be over. To solve this we used OpenMP’s schedule(type, chunk-size) instruction by setting its type to “guided”. It allows each thread to ask for another workload of at least chunk-size if possible, once it completed the one that has been given to him beforehand.

3.6 SIMD and other optimizations

SIMD stands for Single Instruction on Multiple Data and is different from SIMT (Single Instruction on Multiple Threads). With a SIMD processor, data is understood to be in blocks and a number of values can be loaded all at once.

Our implementation still relies mostly on a function returning the Euclidean distance between two vectors. Instead of having a series of instructions saying "add $d * d$ to dist, and do it again", we want to have a single instruction that says "add $d * d$ to dist n times". Using OpenMP’s simd reduction directive we are able to force the compiler to implement this reduction.

Finally, as the square root is time consuming and is not mandatory in specific situations (when finding the two closest centroids of a vector), we modified our distance function to avoid applying *square root* on the result.

4 Classification with machine learning

4.1 What is classification?

Classification is a supervised machine learning technique best described as the process of predicting to which label or category new data inputs belong. Classification modeling is done by training a classifier with known data and confronting what it has learnt with unknown ones.

4.2 Preprocessing

To begin with, all the unlabeled vectors were removed as handling them is considered as a bonus.

4.2.1 Principal Component Analysis

Data were analyzed and compressed using the **Principal Component Analysis** - or PCA - which determines which features, or dimensions, are the most significant by getting rid of irrelevant features that may compromise the generalization of our model while training. In other words, it will lead to *overfitting* which is the fact that the model will learn details and noise that will impact negatively the performance of the model.

As one of the goal was also to reduce memory consumption for the program to run on a 8GB RAM laptop, we tried using sklearn's IncrementalPCA that offers a method called *partial_fit*() allowing to fit our dataset using PCA method using mini-batches. This worked well until we tried to apply the dimension reduction to our original dataset using the *transform*() method which took more than 25GB RAM according to Google Colab's indicators.

Since it was not working properly with IncrementalPCA and we were not able to identify what was the origin of such memory consumption we decided to go back to normal PCA which only took up to 14GB of memory compared to the IncrementalPCA implementation.

4.2.2 Choosing the number of components to keep

As mentioned previously, the goal is not to keep too many features as some of them can be irrelevant and will decrease the performance of our model. The goal is then to be able to know how many features we want to keep.

This is possible by using the cumulative summation of the explained variance which is the ratio of the variance of a given feature over the total variance.

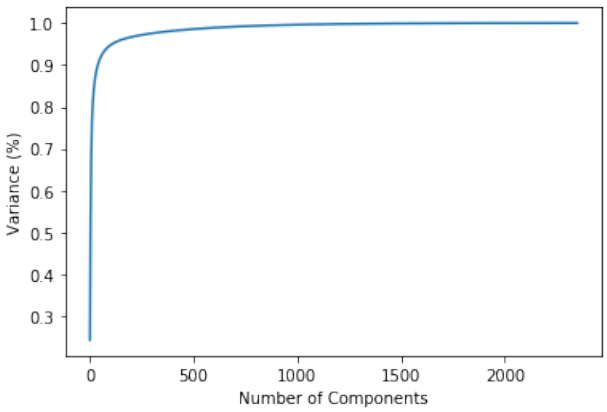

Figure 1: Cumulative summation of explained variance

On **Figure 1**, one can see that with about 500 components it is possible to preserve more than 95% of the total variance. Thus it has been assumed that keeping about a quarter of the features was enough not to lose to much information.

4.3 Building the classifier

We used one of the **Naive Bayes classifiers**, which are classification algorithms for binary and multi-class classification problems. This technique is easier to understand when described using binary or categorical input values. The classifier is based on the *Bayes Theorem* with an assumption of independence among predictors. It is easy to build and is useful in the case of large dataset. Bayes Theorem:

$$P(A \mid B) = \frac{P(B \mid A)\, P(A)}{P(B)}$$

More specifically, we have chosen the Bernoulli's Naive Bayes model, as it fits our dataset which consists of binary features, either malignant or benign programs. If X is a random variable, it can assume two values, and their probability is:

$$P(X) = \begin{cases} p & if\ X = 1 \\ 1 - p & if\ X = 0 \end{cases} \quad with\, 0 \leq p \leq 1$$

4.4 Evaluating the performance

We can evaluate the accuracy of our model by comparing the values of the labels of the test set and the predictions. We obtained an accuracy of 77%.

5 Classification using Deep Neural Network

5.1 Principle of a neural network

The principle of a neural network is something rather easy to grasp since we have already seen it last year with our OCR project. It tries to mimic the way our brain operates. However, the logic behind this is rather complicated since it uses notions such as loss functions and gradient descent. Keras library makes all the computations for us and our main goal was to find the best topology.

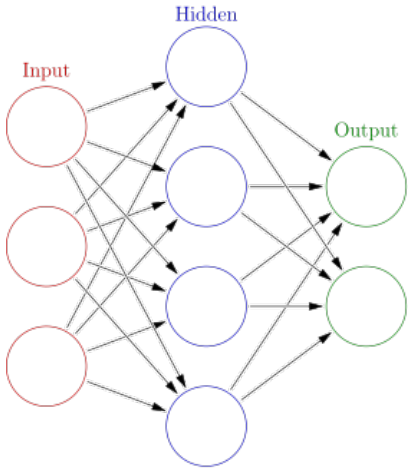

Figure 2: Structure of a neural network.

Now, let's dive right in the mathematics. Let's denotes $hiddenRes$, the 1-dimensional array that will contain all the results of the input to hidden layer process for each hidden neuron. Then we have the following formula:

$$hiddenRes[i] = \sigma(\sum_{j=0}^{n}(inputs[j] * weightInputToHidden[j][i]) + biasHiddenLayer[i])$$

1. with our activation function: Relu for example.
2. with n = numberInputNodes
3. with **weightInputToHidden**, the matrix of weight between the two first layers.
4. with **inputs**, the array of inputs values.
5. with **biasHiddenLayer**, the matrix of bias between the first two layers.

5.2 Building our model

We used Google Colab in order to do heavy computations with Python and the Keras library. Also, this allowed to get really fast results since the whole dataset could be loaded in RAM.

In order to do supervised learning, we had to get rid of all the unlabeled data in the dataset and in the label set. We split the resulting set in two parts, one being our brand new testing set and the other one our former testing set (the newer testing set being larger than the other). We wanted to use the Xtrain file but did not quite understand to which label those vectors were corresponding (if it was the first vectors of the Ytrain or other ones) so we kept what we had and just divided our dataset in two.

To implement the neural network, we simply read the manual of the given libraries. Then, we tried implementing it according to what we wanted to do, taking into account how other similar problems were resolved in the documentation.

5.3 Evaluating our model and testing our accuracy

After many researches, we found out that finding the perfect topology for the neural network required many tries and that there was not any simple method. We decided to make some tests and tried step by step, eventually reaching a good accuracy and a reasonable computation time. We made at least ten different topologies and recorded the number of neurons in each layer, the loss function used, all the given results, the time for evaluating the dataset, the time for each epoch, the accuracy for the training set and the one for the dataset. All the tests and results are available in detail in the Google Colab. Here is the final topology we have chosen.

5.3.1 Final topology

3 layers:

256	activation = relu	optimizer = adam
256	activation = relu	loss function = sparse_categorical_crossentropy
2	activation = softmax	metrics = accuracy

results:

for the training set:		for the testing set: time: 4s	
loss = 20.7225	accuracy = 0.5094	loss = 10.2904	accuracy = 0.6068

5.3.2 Interpretation of results

Having a topology of 3 layers with 2 hidden layers of 256 neurons each and one last with 3 seems to be the best choice as it combines both performance and accuracy.

The algorithm is indeed significantly faster with this topology in comparison with the other 2 tested: one with the same number of hidden layers and 512 neurons in each one, and another one with the same number of neurons in each layer but with a third hidden layer. Moreover, the results show that this topology gives a better accuracy for the training set and the testing set. These accuracy, although not exactly equal, are more similar than the ones given by the other configurations.

The chosen loss/cost function is the sparse_categorical_crossentropy because it was the only function tested that gave acceptable results with the best accuracy around 0.6.

6 Siamese Network

6.1 Principle of a siamese network

A **siamese neural network** is based on a single neural network that uses the same weights while working on three different input vectors. Associated with a **triplet loss** as the loss function, the aim is to make sure that two feature vectors with the same label have their embeddings close together or far away if they have different labels.

This structure is often described as a **triplet network** and metric learning refers to the techniques of learning a distance function over objects.

To train the neural network, we use triplets composed of: an anchor, a positive with the same label as the anchor and a negative with a different label.

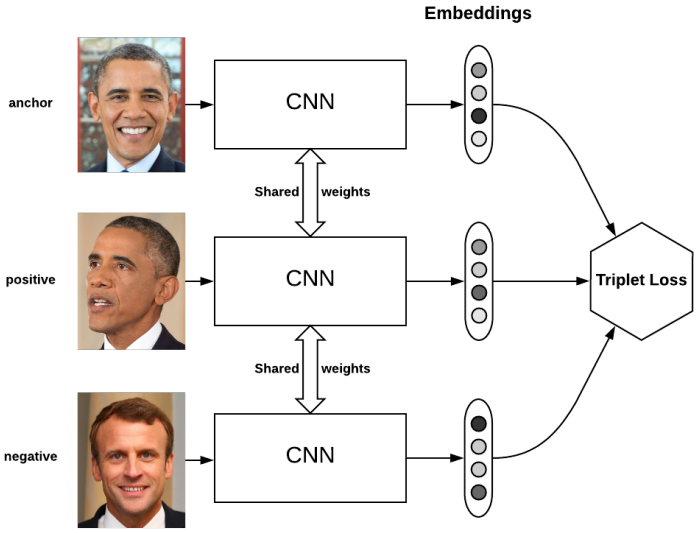

Figure 3: Structure of a triplet network.

6.2 Triplet loss

Considering a triplet (A, P, N) where d represents the Euclidean distance function, A is an anchor input, P is a positive input of the same class as A and N is a negative input of a different class from A, the loss function is: $L = max(d(A, P) - d(A, N) + margin, 0)$.

The aim is to minimize this loss by having $d(A, P) = 0$ and maximizing $d(A, N)$ to have $d(A, N) > d(A, P) + margin$. The cost function is therefore the sum of all losses.

6.3 Triplet mining

6.3.1 What?

It is noteworthy that one of the main implementation problems with triplet loss is how to sample the triplets. Indeed, based on the definition of the loss, there are three categories of triplets:

1. Easy triplets: triplets which have a loss of 0.
2. Hard triplets: triplets where the negative is closer to the anchor than the positive.
3. Semi-hard triplets: triplets for which the positive is closer to the anchor than the negative, but which still have loss.

6.3.2 When?

The mining can be "offline" or "online". In the first case, the algorithm create triplets at the beginning of each epoch, in the latter one it computes useful triplets on the fly. Online mining is therefore more efficient.

6.4 Our implementation

We rely on the triplet semi-hard loss function from *tensorflow_addons*. This function relies on semi-hard online mining. Tensorflow handles all the training computations, however some modifications need to be applied on our model:

1. Removing our output layer (we want embeddings at the end).
2. Removing the activation function of the last layer.
3. L2 normalizing the embeddings.

Unfortunately, the loss does not seem to decrease over the training period even after trying multiple model topologies. The minimum we obtained is $\simeq 0.85$. Not using "batches" from the dataset may explain this phenomenon.

6.5 Using k-means with the new embeddings

Our neural network is now able to create new embeddings from our test dataset. These results (100000 vectors of dimension 256) can be plotted as 2D points on projector.tensorflow.org. Although we are not able to distinguish two separate groups on the embeddings space, vectors belonging to the same class tend to be close to each other.

Figure 4: Representation of the generated vectors with UMAP.

Moreover, the results can be used to train our k-means algorithm. The training is relatively quick because of the small dimension and the small number of vectors. Furthermore, the accuracy is of 0.8601.

The accuracy is significantly greater than the one we get by using k-means on the whole dataset. This can be explained by the fact that we train and test on the same dataset but still the result is worthwhile as doing so on the whole dataset would not give such accuracy.

Therefore, triplet networks seem to be very efficient to compute new feature vectors (embeddings) which, with their small dimension, can be used in parallel of other classification techniques.

7 Conclusion

7.1 The project

This project was really interesting as it has given us the opportunity to broaden our knowledge. We learned about new and different techniques in machine learning and deep neural networks for classifying a dataset, as well as clustering, and thus the different steps of the workflow of a machine learning project. We also learned about different tools for implementing these techniques in Python, such as Scikit-learn and Keras.

The table below is summing up the results obtained in terms of accuracy, time and implementation difficulty. As we implemented k-means on a low level perspective, this method is faster but gives a lower accuracy than common machine learning libraries.

	Accuracy	Time	Implementation	Link
k-means	$\approx 56\%$	Fast (0.300 seconds)	Hard	Gitlab
Classification with ML	$\approx 77\%$	Medium (PCA)	Medium	Google Colab
Classification using DNN	$\approx 60.6\%$	Slow (training)	Medium	Google Colab
Siamese Network	$\approx 86.01\%$	Slow (training)	Medium	Google Colab

7.2 Our thoughts

Working in competition against other teams was also really motivating and it pushed us to do better. The k-means part was the one that pleased us the most since we like working with low-level languages. It was even more pleasing to work on such project as it is not the usual kind of project we see in EPITA. Also, reading computer science research papers was enriching as it allowed us to deepen our knowledge on machine learning. We all have a common interest in artificial intelligence and some of us wish to pursue their career in that field. Our group is working well together and it would be delightful to join the ESLR team and work on the Autocar project.

8 Bibliography

- Making k-means even faster, Greg Hamerly
  https://pdfs.semanticscholar.org/103e/3167b2308987a809d5eed679dff213861664.pdf
- Initialization des centres
  https://stats.stackexchange.com/questions/30723/initializing-k-means-centers-by-the-means-of-random-subsamples-of-the-dataset
- Optimization of Hamerly's K-Means Clustering Algorithm: CFXKMeans Library
  https://colfaxresearch.com/cfxkmeans

- StatQuest: Principal Component Analysis (PCA) Step by Step
  www.youtube.com/watch?v=FgakZw6K1QQ
- PCA dimension reduction
  https://towardsdatascience.com/an-approach-to-choosing-the-number-of-components-in-a-principal-component-analysis-pca-3b9f3d6e73fe
- Data compression via dimensionality reduction - Principal Component Analysis
  https://bogotobogo.com/python/scikit-learn/scikit_machine_learning_Data_Compresssion_via_Dimensionality_Reduction_1_Principal_component_analysis%20_PCA.php
- PCA in sklearn
  https://sdsawtelle.github.io/blog/output/week8-andrew-ng-machine-learning-with-python.html#PCA-in-sklearn
- Naive Bayes methods
  https://towardsdatascience.com/naive-bayes-classifier-81d512f50a7c
- Naive Bayes implementation with scikit
  https://scikit-learn.org/stable/modules/naive_bayes.html

- TensorFlow Addons Losses: TripletSemiHardLoss
  https://www.tensorflow.org/addons/tutorials/losses_triplet
- Triplet Loss and Online Triplet Mining in TensorFlow, Olivier Moindrot
  https://omoindrot.github.io/triplet-loss
- Keras documentation
  https://keras.io/

9 Appendix - kmeans_aux function source code

```
static unsigned kmeans_aux(float *vectors, struct kmeans_state *state)
{
    unsigned change_cluster = 1;

#pragma omp parallel reduction(&: change_cluster)
    {
        // Allocate thread memory
        struct kmeans_thread *t = calloc(1, sizeof(struct kmeans_thread));
        t->change_cluster_t = calloc(state->K, sizeof(unsigned));
        t->centroids_sum_t = calloc(state->K * state->vect_dim, sizeof(float));
        t->centroids_count_t = calloc(state->K, sizeof(unsigned));

        // Apply k-means algorithm for each vector
#pragma omp for schedule(guided, 10)
        for (unsigned i = 0; i < state->vect_count; i++)
        {
            float m = fmax(state->s[state->assignment[i]] / 2, state->lower_bounds[i]);

            // First bound test
            if (state->upper_bounds[i] > m)
            {
                // Tighten upper bound
                float *v1 = vectors + i * state->vect_dim;
                float *v2 = state->centroids + state->assignment[i] * state->vect_dim;
                state->upper_bounds[i] = distance(v1, v2, state->vect_dim, 0);

                // Second bound test
                if (state->upper_bounds[i] > m)
                    update_assignment(vectors, i, t, state, &change_cluster);
            }
        }

        // Reduction
        for (unsigned c = 0; c < state->K; c++)
        {
            if (t->change_cluster_t[c])
            {
#pragma omp atomic
                state->centroids_count[c] += t->centroids_count_t[c];
                for (unsigned d = 0; d < state->vect_dim; d++)
                {
                    float value = t->centroids_sum_t[c * state->vect_dim + d];
#pragma omp atomic
                    state->centroids_sum[c * state->vect_dim + d] += value;
                }
            }
        }

        // Free thread memory
        free(t->change_cluster_t);
        free(t->centroids_sum_t);
        free(t->centroids_count_t);
        free(t);
    }

    return change_cluster;
}
```