Machine Learning - HW1

Gianfranco Romani 1814407

November 29, 2020

Abstract

In this report I will compare several Machine Learning methods used to classify functions from their assembly code. I will particularly focus on comparing the results obtained by training these models on two versions of the same dataset, one that has duplicates and one that doesn't, then I will try to use some methods to balance the number of samples over the classes.

1 Introduction

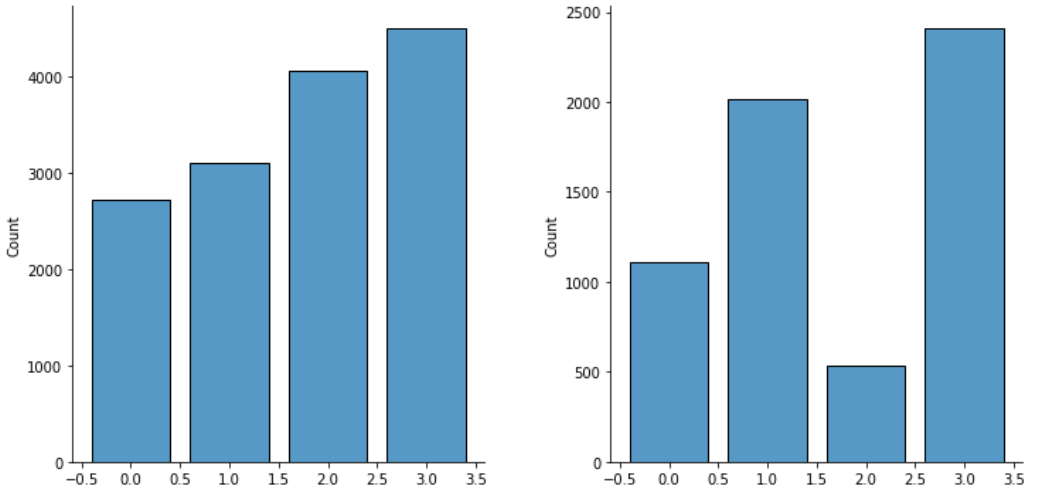
To keep up with malwares, it is needed to improve constantly our defences, but detecting a malevolent piece of code is not always easy and it often takes a lot of time. A way to help in the analysis of software is to use machine learning, that can, for example, classify specific functions in unknown software. In this report I will describe the results obtained in such task by different ML models. I will firstly analyze the dataset (2) and the techniques I used for the preprocessing (3), then there are the sections about the training(4) and results (5) in which I will comment the results obtained.

2 Dataset

The dataset that has to be used for this homework consists in a JSONL file, so it has a json object for each line. Each object is a dictionary that describes a function. The keys are:

- ID: identifier of the function, unique for each one of them;
- semantic: type of the function;
- lista_asm: list of assembly instruction of each function;
- cfg: control flow graph represented as a networkx graph.

There are four possible labels for the functions: encryption, string, sort, math. The number of lines, and so of the functions, is 14397, but there are several duplicates that, once eliminated from the original dataset, reduce the samples to 6073. As I am going to show later, it is important to remove the duplicated because their presence can influence the performances of the models.



(a) *Distribution of the samples over classes in the dataset with duplicates*

(b) *Distribution of the samples over classes in the dataset with no duplicates*

Figure 1: Number of samples for every class

As shown in the image 1 the samples are not equally distributed between the classes, in particular way in the set without any duplicates. This has to be considered when splitting the data into train and test set and also when evaluating the performance of a model.

3 Preprocessing

To use ML models on this dataset, it is needed to perform some preprocessing. As first thing I encoded the possible classes as integers, in particular: encryption as 0, string as 1, sort as 2, math as 3; so the labels' array will be composed by these integers. The features are extracted from the list of assembly instruction in lista_asm. An intelligent way to decide what in the assembly code could be useful, is to analyze what are typical characteristics of these kinds of functions. Encryptions functions are generally long and complex, they have a lot of nested if-conditions (jne and cmov for example) and for loops and instructions like shifts, xor and other bitwise operations (and, or) are heavily used. Sort functions, on the contrary, are very simple: just one or two for loops and some cmp, test and mov operations. String manipulation functions are mainly composed by comparisons (cmp, test) and swap of memory locations. Lastly, math functions use, of course, arithmetic operations (div, imul, inc etc) and special registers as xmm. Besides these features I also considered the presence of jumps in the code (jmp) and function calls (call, leave).

To obtain the features and transform the dataset in a way that is easier to modify and on which the models could train, I wrote the function $take\_features(data)$ that takes as input the JSONL file and returns a two dimensional numpy array of 6073 rows (dataset with no duplicates) and 12 columns, one for each feature plus the identifier and the label of the function. The features are extracted by the function $find\_in\_asm(asm)$ that just counts the occurences for each feature.

The last operation done in this phase of the work is the data splitting. After deleting the column with the IDs of the functions and creating the array for the labels, $train\_test\_split(data\_new, labels, test\_size = 0.3, random\_state = 42)$ makes the train_set, the test_set and the relative labels. At the beginning I passed to *train_test_split* the parameters test_size=0.3, random_state=42 to take 70% for the train and the rest for the test_set, dividing the samples randomly, but I didn't have

considered the fact that the dataset is unbalanced. Due to the disparity of classes in the samples, the algorithms could tend to categorize into the class with more instances giving the false idea of high accuracy and performing badly on the class that has less elements. To improve performances I have used the parameter $stratify = labels$. Despite this modification the final results show that the more represented classes are predicted better than the one that has less samples, particularly I will underline this with the confusion matrix.

3.1 Resampling

After some tests it was clear that the limits of my models were in part due to the unbalancing of the dataset, so I decided to try some techniques (from the library: *imbalanced-learn* [2]) to solve this problem. The possibilities that I try to go through are called over-sampling and under-sampling. The first seemed to me the more appropriated because it consists in balancing the data increasing the number of samples of the less represented class, duplicating some samples taken randomly, while the second method reduces the number of samples of the abundant class. These proceedings can improve performances, in particularly for ML models that seek good splits of the data, like SVM and decision trees, but they could cause overfitting for the minority class. After some attempts with these two approaches (the results are discussed at the end of the report) I decided to try another technique: SMOTE. SMOTE stands for Synthetic Minority Over-sampling Technique and it consists in taking a sample of the dataset and its k nearest neighbours and then it plots a vector between all these elements. Multiplying this vector by a random value from 0 to 1 we obtain a new sample for the minority class, called synthetic data point. There are several versions of SMOTE, the one I used is Borderline SMOTE that focuses on samples that are situated near the border of an optimal decision function. After using this method all the classes have 2412 samples.

4 Training

This section will be divided into two main parts, the first one in which I am going to report the trainings on the data with duplicates, while in the second it is considered the dataset without duplicates. At the end of this two sections there is a smaller one about the attempts with the library *imbalanced-learn*. The comments on the statistics presented here are in the results chapter.

As first method I used decision trees because, generally, this model is useful when the number of samples is higher compared to the number of features, as in this case. The main problem with decision trees is overfitting, that could be easily obtained, and the lesser accuracy on classes that have less samples. To avoid these problems I also trained a random forest that outperformed the former method. Then I also implemented a support vector machine to confront it with random forest. For all three algorithms I used cross validation to check the validity of the models. For all the tests the number of folds is 10 and the scoring is relative to accuracy. The library from which I took these methods is scikit-learn [1].

Every test was done using Google Colaboratory.

4.1 Dataset with duplicated samples

Total number of samples: 14397, train set: 10797, test set: 3600.

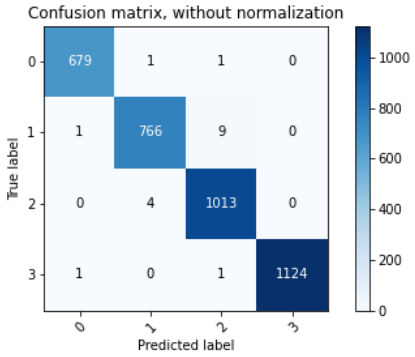
4.1.1 Decision tree

I carried out different tests in which I have modified the parameters *criterion* and *class weight*. I haven't noticed big differences in accuracy (always good) for the tests, but the version of which I submit the results is the one with *criterion = entropy* and *class_weight = balanced* because the errors on the classes were more distributed between them instead of centered on the string and sort classes. Entropy is used instead of gini when we care more about having a slightly more balanced tree, losing a bit in velocity of computation, but I registered very small differences.

The best results I got (using cross validation) are: an average result of 99.76% in accuracy with a standard deviation of 0.1. Training time and predictions turn out to be very fast.

	precision	recall	f1-score
0	0.997	0.997	0.997
1	0.994	0.987	0.990
2	0.989	0.996	0.993
3	1.000	0.998	0.999
accuracy			0.995
macro avg	0.995	0.995	0.995
weighted avg	0.995	0.995	0.995

(a) *Classification report*



(b) *Confusion matrix*

Figure 2: Training results for decision tree with duplicates

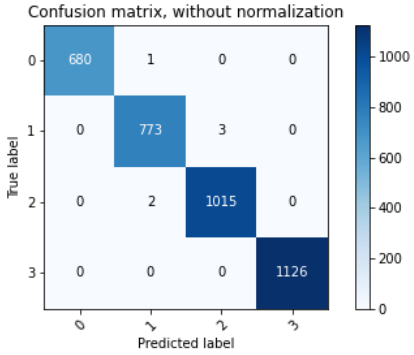
4.1.2 Random Forest

Random forest is an ensemble method which means that it trains a group of decision trees classifiers, each on a different random subset of the training set, and then it predicts the class that gets the most votes. The number of trees to be used is defined by the parameter *n_estimators* and after some tests I decided that a good amount of estimators to obtain a balance between accuracy and training speed is 100. As in decision trees I tried to change *criterion* to entropy and the performance were better, even if the improvements were little. The last parameter I use in this model is *n_jobs*, that specifies the number of jobs to run in parallel, where -1 stands for all processors available.

The results are: 99.92% in accuracy with a standard deviation of 0.05. This model results much slower than decision trees, both in training and predictions.

	precision	recall	f1-score
0	1.000	0.999	0.999
1	0.996	0.996	0.996
2	0.997	0.998	0.998
3	1.000	1.000	1.000
accuracy			0.998
macro avg	0.998	0.998	0.998
weighted avg	0.998	0.998	0.998

(a) *Classification report*



(b) *Confusion matrix*

Figure 3: Training results for random forest with duplicates

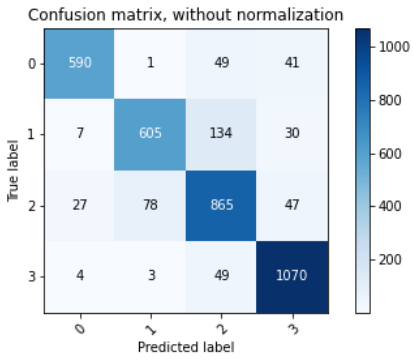
4.1.3 Support Vector Machines

I choose to train SVM because of their capacity to perform both linear and nonlinear classification. Their are considered to not be the best choice if the dataset is large or in case of overlapped classes, so I expected worse results than the previous models. The paramaters I used are: *kernel*, *gamma* and *class_weight*. The first specifies the kind of kernel to be used between 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'. 'Poly' is the one that obtained the worst results, rbf the best. Gamma is a coefficient for 'rbf', 'poly' and 'sigmoid' kernels, I set it to 'auto' that uses $1/n\_features$. For the last parameter, *class_weight*, I have used the balanced mode, that adjust weights in a inversely proportional way in regard to class frequencies in the input data, and the default value that fixes all weights to one. Standardization of the dataset is done through the function StandardScaler() that calculates the score of a variable x as $z = (x - u)/s$ where u is the mean and s the standard deviation.

Training speed is a bit slower compared to random forest. Accuracy varies from $\approx 83\%$ for 'poly' kernel to 93.21% for 'rbf' kernel.

	precision	recall	f1-score
0	0.939	0.866	0.901
1	0.881	0.780	0.827
2	0.789	0.851	0.818
3	0.901	0.950	0.925
accuracy			0.869
macro avg	0.877	0.862	0.868
weighted avg	0.872	0.869	0.869

(a) *Classification report*



(b) *Confusion matrix*

Figure 4: Training results for support vector machines with duplicates

4.2 Dataset without duplicated samples

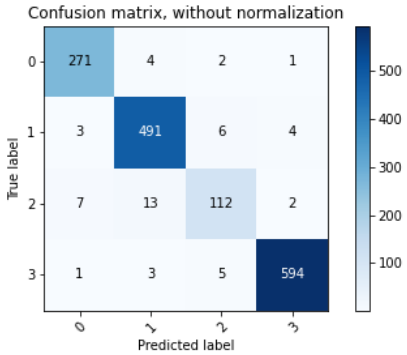
Total number of samples: 6073, train set: 4554, test set: 1519. The observations made on the parameters in previous paragraphs are valid also in this part of the report, so I just indicate the results and eventually some differences with the previous implementations.

4.2.1 Decision trees

The accuracy reached by decision trees with no duplicates: 98.12%.

	precision	recall	f1-score
0	0.961	0.975	0.968
1	0.961	0.974	0.967
2	0.896	0.836	0.865
3	0.988	0.985	0.987
accuracy			0.966
macro avg	0.952	0.942	0.947
weighted avg	0.966	0.966	0.966

(a) *Classification report*



(b) *Confusion matrix*

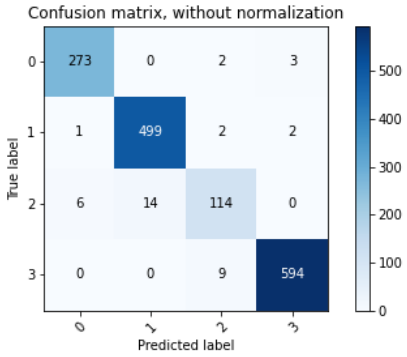
Figure 5: Training results for decision trees with no duplicates

4.2.2 Random Forest

Knowing that this method could be influenced by the disparities in the number of samples, I tried to use an additional parameter, *class_weight='balanced'*, to try to ease the loss of accuracy compared to the attempts with the duplicates. The results, however, were not good and I registered a lost of one percentage point in accuracy. Accuracy after training: 98.66%.

	precision	recall	f1-score
0	0.975	0.982	0.978
1	0.973	0.990	0.981
2	0.898	0.851	0.874
3	0.992	0.985	0.988
accuracy			0.974
macro avg	0.959	0.952	0.955
weighted avg	0.974	0.974	0.974

(a) *Classification report*



(b) *Confusion matrix*

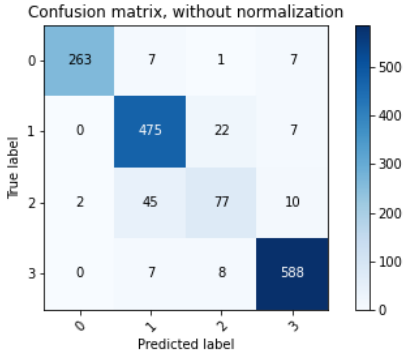
Figure 6: Training results for random forest with no duplicates

4.2.3 Support Vector Machines

Accuracy varies from ≈89% for kernel of type 'poly' to 96.04% for type 'rbf'.

	precision	recall	f1-score
0	0.992	0.946	0.969
1	0.890	0.942	0.915
2	0.713	0.575	0.636
3	0.961	0.975	0.968
accuracy			0.924
macro avg	0.889	0.860	0.872
weighted avg	0.921	0.924	0.921

(a) *Classification report*



(b) *Confusion matrix*

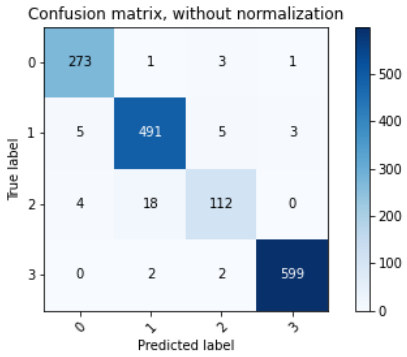
Figure 7: Training results for support vector machines with no duplicates

4.3 Resample of the dataset

Several attempts were made for both oversampling and undersampling methods, changing the parameters *sampling_strategy* and *random_state* in each attempt, but the results were not good. In fact the accuracy remained more or less the same, but the f1-score decreased sensibly and, especially after RandomOverSampler, I believe that there was a little of overfitting for the minority class. The SMOTE method instead is more satisfying because it obtains 98.52% in accuracy and the following outcome for f1-score and confusion matrix after training a random forest:

	precision	recall	f1-score
0	0.968	0.982	0.975
1	0.959	0.974	0.967
2	0.918	0.836	0.875
3	0.993	0.993	0.993
accuracy			0.971
macro avg	0.960	0.946	0.952
weighted avg	0.971	0.971	0.971

(a) *Classification report*



(b) *Confusion matrix*

Figure 8: Training results for random forest after using SMOTE

5 Comments

I think that, seen that the problem is about the classification of different kinds of functions, the metric that is more useful is the f1-score because we are interested in the correct detection of

possibly dangerous software but we also don't want to falsely categorize an harmless functions as a treat. Accuracy will be used to check if a model overfits.

As in the training section I will discuss about the results in three different paragraphs.

5.1 Duplicates

The results obtained from the trainings on the dataset with duplicated samples are generally very good. I couldn't obtain an accuracy lower than 83% and also the f1-score is always quite good. The worst model of the three is SVM that makes some errors inverting the predictions between sort and string classes, in particular the string class is the one with more False Negative and thus lower recall. Decision trees and random forest are almost perfect in predictions, with an accuracy that is near to 100%. The improvements from the first model to the ensemble one are so small that I could suggest to use decision trees because of the fact that they are faster in training and predictions.

All these results seem great but there is one thing to be considered, that there were many duplicates in the dataset, at least 8000 samples. This means that in the test set there are many samples already seen by the models during the training phase and that heavily affects the results. So I don't think that these models could be considered good because they could perform worse outside this tests.

5.2 No duplicates

As expected the results are worse if compared to the previous tests, wrong classifications are more than before and accuracy decreases except for SVM. This last method perform way better (+3%) but commits a lot of errors when it has to predict sort functions, classifying almost a third of samples of this class as string functions. The fact that decision trees and random lose in accuracy, while support vector machines not, proves that the first two models are more sensitive to the distributions of samples.

Generally string and sort classes, the second in particular way, caused more problems to the methods, probably because they are very similar to each other and the models would have needed more samples of the sort class to be train on. This is evident on the f1-score obtained on this class, that is at least ten percent worse compared to the f1-score of the other classes.

I believe that the models trained on this dataset are better than the ones analyzed before because they obtain very good results, except few cases as stated before, but the results are not biased towards the samples with duplicates.

5.3 Resampling

Even if I hoped to see better improvements, SMOTE reaches the goal to increase, albeit slightly, the predictions over the minority class, in particular the f1-score gains some points over the same methods trained on the data without duplicates and resampling. I would like to say that, even if I don't show here explicitly the results, the method that improves the most after SMOTE is SVM that passes from a f1-score of 0.575 to almost 0.7. On the other hand undersampling and oversampling did not improve results, on the contrary I am afraid that oversampling approach caused overfitting for the minority class.

6 Conclusions

In conclusion of this work, I could say that, considering the unbalancing of the samples for the classes, all the models trained behave very well, with some mistakes on the minority class that are not so severe, taking into account that the most important class to detect correctly is the one about encryption functions that are more likely involved in some malevolent software. Interesting could be to inquire more on the resampling methods because they could solve (or at least relieve) the problem of unbalancing present in this dataset.

As method to use for the blind test requested for this homework I will select the second version of random forest, the one trained on the unbalanced dataset with no duplicates, because it is the model that obtains the best f1-score on the encryption class.

References

[1] https://scikit-learn.org/stable/

[2] https://imbalanced-learn.readthedocs.io/en/stable/over_sampling.html#smote-adasyn

[3] https://arxiv.org/pdf/1106.1813.pdf