Machine Learning – Homework 1

Andrea Gasparini
1813486

November 2020

Contents

1 Introduction

The task of this Homework was to build a *multiclass classifier* to determine whether an assembly function is an **encryption**, **math**, **string manipulation** or **sorting** function. Formally, the task was to aproximate a mapping function:

$$f : X_{bin} \rightarrow \{encryption, math, string, sort\} \tag{1}$$

where X_{bin} is an assembly set of instructions that can be classified in one of the target classes of the f function's right-hand side. To be more precise, since we have to define a features list to train a Machine Learning model, X_{bin} is a list of measurable properties extracted from the dataset after a feature extraction procedure (section 3).

2 The Dataset

The dataset used to train the model was provided in two versions:

- With 14397 assembly functions, containing duplicates
- With 6073 assembly functions, without duplicates

As we can see in Figure 2, removing the duplicates creates an unbalanced dataset and this can lead to problems in predicting the minority class. An oversampling procedure has probably been applied to obtain the dataset with duplicates and solve the unbalancing problem. Training a model with both versions of the dataset allowed to evaluate the differences with some metrics and then choose the best one.

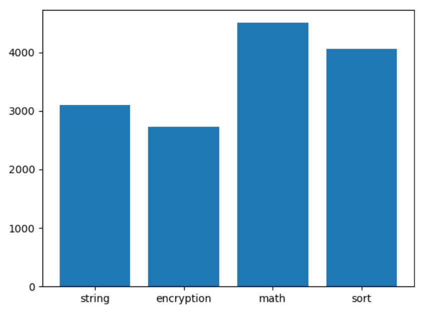

Figure 1: Dataset with duplicates

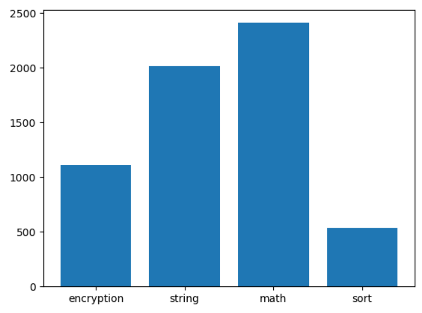

Figure 2: Dataset without duplicates

2.1 Undersampling procedure

Another dataset could be obtained applying an undersampling procedure on the one without duplicates. In other words, removing some entries of the majority classes to obtain a balanced distribution of all classes. The problem with this approach is in the size of the original dataset and of the minority class (*sort*), the process could lead to loss of valuable information and to a too small dataset. The idea was then discarded.

2.2 Data format

Each entry of the dataset, provided as a `JSON`, represent an assembly function and contains:

- The label of the function, that is one of the target classes defined in Equation 1 (*math*, *sort*, *encryption* or *string*)
- The linear list of his assembly instructions
- The control flow graph (CFG), econded as a NetworkX graph

3 Preprocessing and feature extraction

Since the assembly instructions were encoded as lists of strings, a first approach I tried was to solve the task as a standard *text classification problem.* In this way the features list for the model contained the occurences' number of every "word" in the assembly functions. The main problem in this case was caused by the lack of a *stop words* set. In fact, in a *text classification problem* we should always specify which words don't have to be considered because of their common use in the language (such as *the*, *is*, *which*, etc) and focus on the important words instead. For example counting the occurences of words representing registers wasn't necessary and could lead to wrong predictions. Then I changed the approach and thought of a better way to extract some specific characteristics from the dataset to train the model.

3.1 Characteristics of the target classes

For every target class we had some hints about its common characteristics:

- *encryption*: contains a lot of nested `for` and `if` (i.e. a complex CFG), uses a lot of `xor`, shifts and bitwise operations, and is extremely long

- *sort*: contains one or two nested `for` and some helper functions, uses compare and moves operations
- *math*: uses a lot of arithmetic operations and special registers `xmm*` for floating point instructions
- *string*: uses a lot of comparisons and swap of memory locations

The point was to take advantage of these characteristics to map the dataset's entries to features lists that could be used to train a model. Every entry in the dataset has been processed and mapped in a list with the following 9 features:

- *graph_nodes*: is the number of nodes in the CFG, where each node represents a *basic block*, i.e. a straight-line piece of code without any jumps or jump targets
- *graph_diam*: is the maximum diameter value of the CFG's strongly connected components, where the diameter of a component represents the maximum distance between its pair of vertices
- *mov*: is the number of `mov` instrunctions in the function
- *bitwise*: is the number of bitwise instrunctions in the function
- *xmm*: is the number of `xmm` registers in the function
- *arithm*: is the number of arithmetic instrunctions in the function
- *cmp*: is the number of `cmp` instrunctions in the function
- *shift*: is the number of shift instrunctions in the function
- *calls*: is the number of calls instrunctions in the function

4 Evaluation

In order to evaluate a solution based on a features list it's necessary to train some models and compare the performances according to some metrics.

The comparisons in this report are based on the results obtained from two models: **Decision Tree** and **Support Vector Machine**. I used different metrics to evaluate the correctness of these models, like plotting the *confusion matrices* and comparing *precision*, *recall* and *f1-score*. Another metric taken in consideration was the way of splitting the dataset. In fact the standard way is to reserve $\frac{2}{3}$ for the training and $\frac{1}{3}$ for the test. Decreasing the first one and therefore increasing the second one could get worse performances, but whether we still get acceptable results the approach is probably working well.

Precision With *precision*, for a certain class x, we indicate the percentage of correctly predicted instances among all the instances predicted as x:

$$\frac{TP}{TP + FP}$$

where TP represents the correctly classified instances count and FP the wrongly classified as x count.

Recall With *recall*, for a certain class x, we indicate the percentage of correctly predicted instances among all the really x:

$$\frac{TP}{TP + FN}$$

where FN represents the instances count of x wrongly classified in another class.

F1 score The *F1 score* is the harmonic mean between *precision* and *recall*

$$\frac{2 \cdot precision \cdot recall}{precision + recall}$$

Confusion Matrix A confusion matrix is an easy way to visualize the performance of a learning algorithm. Each row of the matrix represents the number of instances (or the percentage if it's normalized) in the true class while each column represents the instances in a predicted class. All correct predictions are located in the diagonal of the table, which elements represent the *recall* values, so it is easy to visually inspect the table for prediction errors, as they will be represented by values outside the diagonal.

4.1 Comparisons

The following comparisons shows the *confusions matrices* resulting of the two models' training on the dataset with duplicates (with-dup), and on the dataset without duplicates (no-dup). Comparisons have been made with a test size of 33% and of 66% to check if the accuracy consequently decreases or the results remain good.

4.1.1 Decision Tree

With this model the first impression with the final features list was to have an overfitting since the performance (Figure 3) seemed to be too high. An increase in the test size did not decrease the performance and this led me to reconsider the assumption of overfitting. The dataset without duplicates had a predictable result since the *sort* class was the minority.

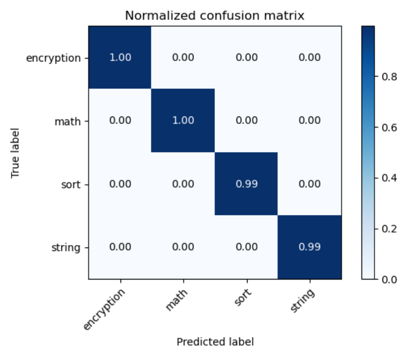

Figure 3: DT with-dup – test 33%

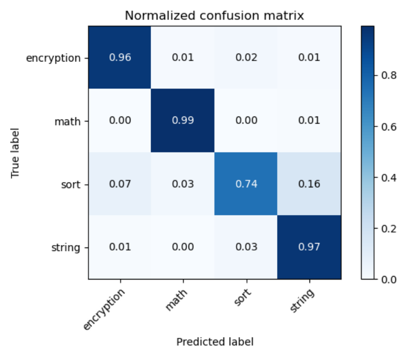

Figure 4: DT no-dup – test 33%

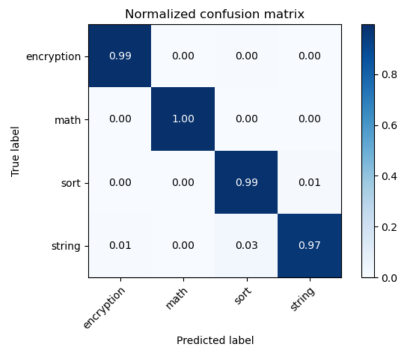

Figure 5: DT with-dup – test 66%

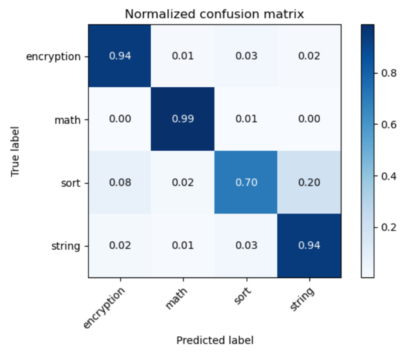

Figure 6: DT no-dup – test 66%

4.1.2 Support Vector Machine

In this case the model needed to be configured with a kernel and a parameter C, that have been set after some tests respectively as `linear` and 1 to have the best trade-off in performance and training time. Compared to the Decision Tree, SVM had a little worse performance, but still really good.

The model performed better without the duplicates, with an increase in the Recall for every target class except for *sort*, because it was the minority class in the unbalanced dataset.

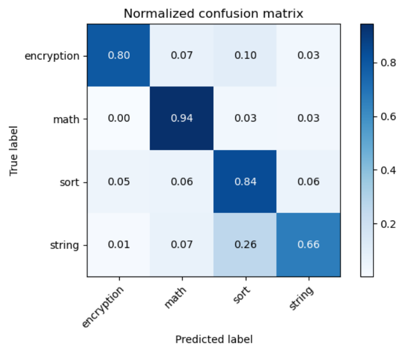

Figure 7: SVM with-dup – test 33%

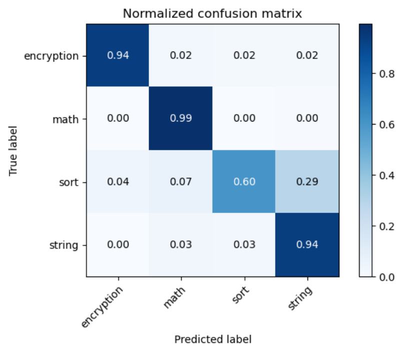

Figure 8: SVM no-dup – test 33%

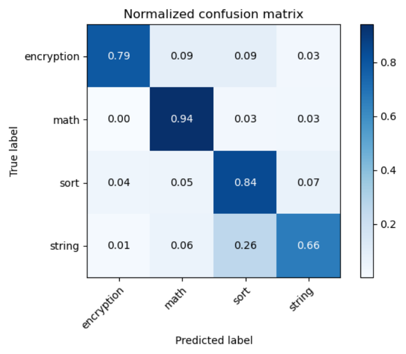

Figure 9: SVM with-dup – test 66%

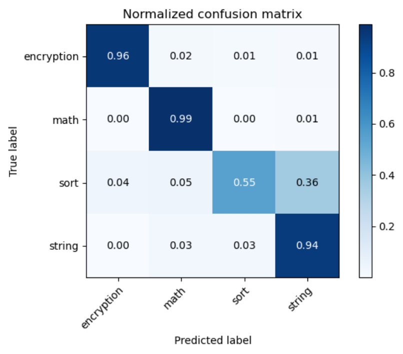

Figure 10: SVM no-dup – test 66%

5 Conclusions

The final features list defined in subsection 3.1 was obtained after some tests with less features and more than two models. The results showed seemed to be the best ones and a big increase in the performance was granted by the use of the CFG features (*graph_nodes* and *graph_diam*). For example by replacing *graph_nodes* with an apparently similar feature that is the number of assembly instructions we get 10% worse performance.

We can tell from the previous comparisons that Decision Tree obtains the best results with both datasets. The first impression was to have an overfitting since a Recall near to 0.99 seemed to be too high, but the results were still good with the second dataset (no-dup) and also with an increased test size, i.e. a decreased training size (that I thought should led to worse results). Another reason to prefer Decision Tree was the training time, that was always better than tested configurations of SVM without having worse performances.

In both cases the bigger problem was in the misclassification of *sort* and *string* functions, in fact both DT and SVM could classify a *sort* function as a *string* one because of the low number of instances. Incrementing the number of *sort* instances in dataset without adding duplicates would certainly improve the performances for these cases, but produce this new data would require a lot of effort.