ML - HW1 Report

Eduardo Rinaldi 1797800

November 2020

Contents

1 Introduction

This is my report about the first homework assigned for the course of **Machine Learning**.

1.1 Assignment

The assignment of the homework was to solve a classification problem about assembly functions: given a function in assembly determine whether it is an **encryption, string, math** or **sorting** function.

The goal of this homework, in other words, is to train a model (any supervised model, except Neural Networks) and find a function:

$$f : X_{asm} \rightarrow \{"Encryption", "String", "Math", "Sorting"\}$$

where X_{asm} is the set of instructions, with the relative CFG (Control Flow Graph). Later X_{asm} will be mapped into the features space.

1.2 Tools used

The language chosen is `Python 3.6.8` because it has a lot of libraries and support for the machine learning world. The following Python libraries have been used to carry out this project:

1. **Numpy:** it adds support for large, multi-dimensional arrays and matrices, it also provides a lot of functions for manipulating datas and is well integrated with the other libraries.
2. **SKLearn:** it's the main library for this homework, that's because it provides several implementation of different models for Machine Learning. It also provides some evaluation metrics that can help us to understand which model is better.
3. **Matplotlib:** provides functions for plotting datas.
4. **Json:** built-in module that is used for parsing the dataset.

In order to extract information from the CFG I decided not to use any library because the ones already present gave problems on performance, so this "module" has been implemented personally.

2 Dataset

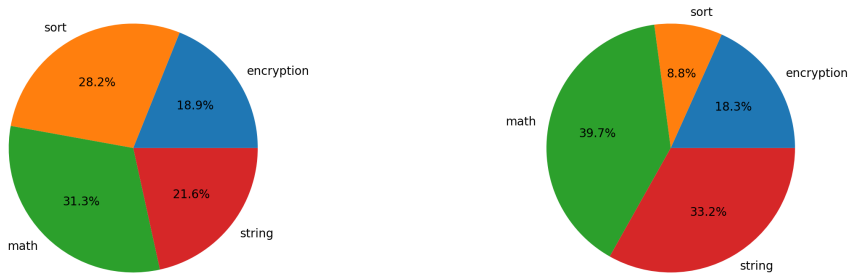
The dataset used, and which was provided to me by the professors, is present in two versions:

1. With duplicates: 14397 entries

2. Without duplicates: 6073 entries

Both versions were tested and in the end only one of the two was chosen for training the model.
Each entry in the dataset gives us:

- A list of all the single assembly instructions used in this specific function
- The control flow graph, which can be analyzed for knowing more information about for example the complexity of the function
- Function type, which can be seen as the target to our problem so $t \in \{"Encryption", "String", "Math", "Sorting"\}$



(a) Dataset with duplicates

(b) Dataset without duplicates

Figure 1

As you can see from figure (1), the dataset without duplicates is quite unbalanced w.r.t. the sorting functions, which is only the 8.8% of the dataset.
In the dataset with duplicates instead this situation is reversed, and this allows us to automatically solve the problem of unbalanced dataset because an upsampling procedure has already been applied. As said before, both datasets has been tested for training and evaluated according to some metrics.

3 Feature extraction

The dataset as it is cannot be passed to a model for training, it is therefore necessary to define a $\phi(x)$ function which takes as argument an entry of the dataset and maps it to the feature space.
More precisely, this mapping function given an x entry (which corrisponds to a json line) returns a vector like:

$$\phi(x) =< bw, istr, mv, xm, arithm, cmp, swp, loops, cplx, ncomp >$$

where:

1. *bw*: is the number of bitwise instructions in the function
2. *istr*: is the number of instructions which compose the function
3. *mv*: is the number of `mov` instructions in the function
4. *xm*: is the number of times an `xmm` register has been used in the function
5. *arithm*: is the number of arithmetic instructions in the function
6. *cmp*: is the number of `cmp` instructions in the function
7. *swp*: is the number of `swap` instructions in the function
8. *loops*: is the number of loops in the functions
9. *cplx*: is an estimated complexity of a component (then it will be defined what we means for component)
10. *ncomp*: is the number of components in the graph

The targets have also been mapped in the following way:

$$\phi_t(x) = \begin{cases} 0, & \text{if x target is Encryption} \\ 1, & \text{if x target is Sort} \\ 2, & \text{if x target is Math} \\ 3, & \text{if x target is String} \end{cases}$$

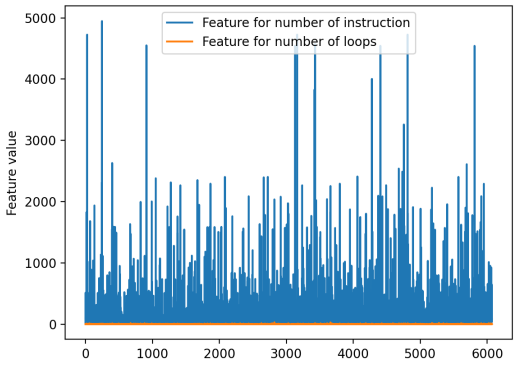
3.1 Get features from CFG

As I said before no library has been used for counting loops and for finding the complexity of the function. The idea behind this is based on the fact that if we explore the graph (with a BFS for example), we can notice that in some cases we get into a loop of nodes, but if there's a loop in the CFG there must be a loop also in the code and that's the intuition behind all.

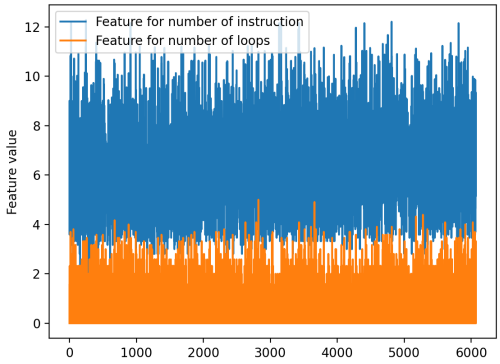
So I implemented the simplest version of the *Tarjan algorithm* for finding strongly connected components. Each component that has at least 2 nodes is considered as a loop and the loops counter is increased (the final value of this count is the *loops* value returned by ϕ).
Extra informations are given by the size of the components, which is the *cplx* value returned by ϕ (the size of the largest component), and the number of components in the graphs which is *ncomp*.

3.2 Scaling the features

To prevent features that vary over a range of very small values (such as the number of loops) from being not considered by the model as important as other features that vary over a range of high values (such as the number of asm instructions in the function), it is necessary to scale the values.



(a) Before scaling the features



(b) After scaling the features

Figure 2

The problem here is that we cannot apply a simply min max normalization because there is a min but there's no max value for each feature (because they're counters of occurrencies), so we cannot map value from a range to another (i.e. the range [0, 1]). What we can do is to apply a further transformation to features with a function that grows very slowly; a function that reflects this property is the logarithm.

As we can see in figure (2), after applying the logarithm for the scaling of the features they vary in a range of values very close to each other.

4 Model selection

In the field of machine learning it is difficult to know a priori which model works best for a certain case of use, so what you do is to train and compare several models and then choose the one that, according to some criteria, seems to perform better.
In our case the models chosen are:

- **Decision Tree** - `sklearn.svm.SVC()` implementation
- **Support Vector Machine** - `sklearn.tree.DecisionTreeClassifier()` implementation

After selecting the models, and mapping data to features space, the next step is to split the data in training and test sets (generally 33% for testing) and then feed the training data to the model. Then the model has been evaluated through some metrics like: *confusion matrix, precision, recall* and *f1 score*. Results are presented in the next section.

5 Results

Results are divided in 4 different cases:

1. Dataset with duplicates and decision tree as model
2. Dataset without duplicates and decision tree as model
3. Dataset with duplicates and svm as model
4. Dataset without duplicates and svm as model

For each case there'll be presented scores and brief comment about the result.

The metrics that are used for evaluating each model are:

- **Confusion matrix:** where each column of the matrix represents the instances in a predicted class while each row represents the instances in an actual class. So on the diagonal of the matrix we will have how many times the model succeeded in predicting a given class.
- **Precision:** this metric makes us understand how many true positives there are among all the positives identified by the model. So it tells us the precision of the model.

- **Recall:** this metric tells us how many true positives the model has identified among all those that are really positive (false negatives+true positives)
- **F1 Score:** this metric correlates precision and recall

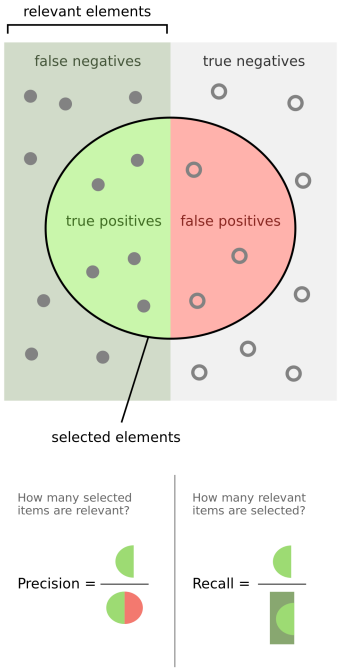
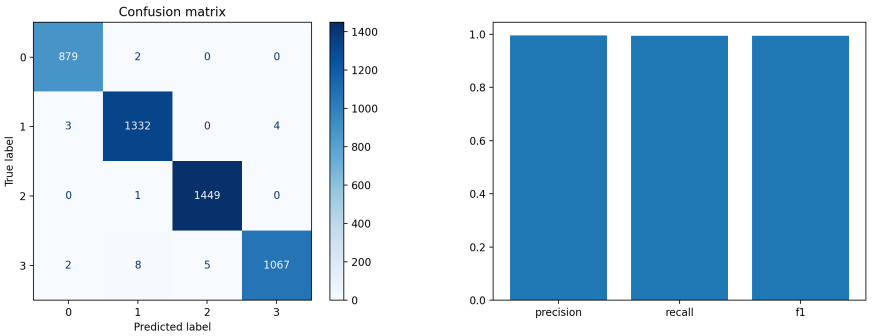


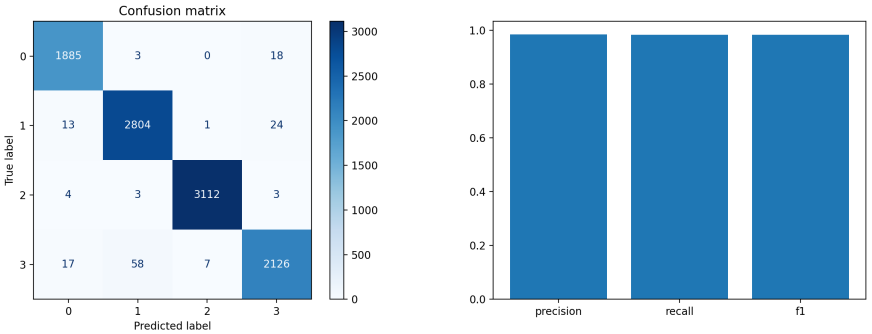
Figure 3: Image that show us intuitively what are precision and recall

5.1 Dataset with duplicates and decision tree as model



(a) Confusion matrix (b) precision, recall, f1 scores

Figure 4: Test set size 33%

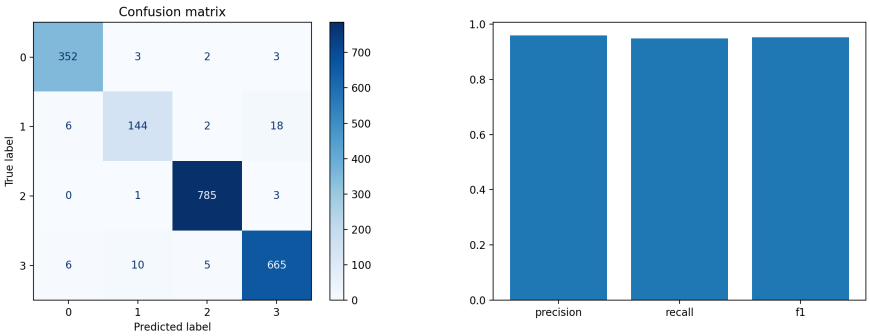


(a) Confusion matrix (b) precision, recall, f1 scores

Figure 5: Test set size 70%

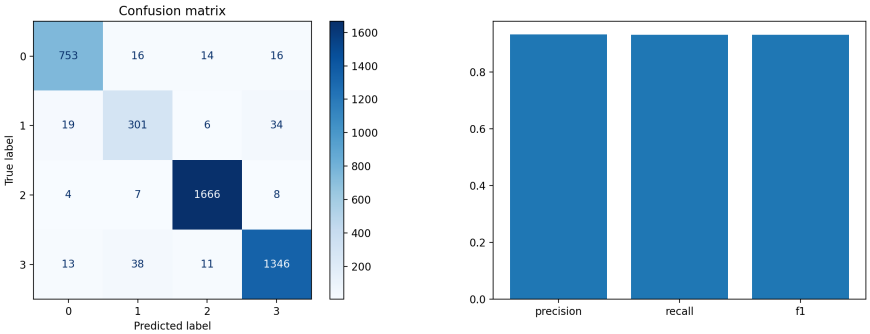
As we can see from figure (4) this model on this version of the dataset obtain almost perfect results. This led me to think that the model could have gone in overfitting, so I decided to upscale the test size to 70% of the whole dataset and as we can see from figure (5) results remain very good.

5.2 Dataset without duplicates and decision tree as model



(a) Confusion matrix (b) precision, recall, f1 scores

Figure 6: Test set size 33%

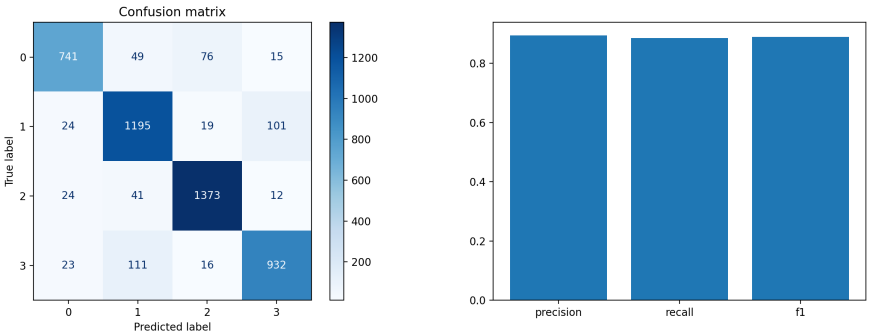


(a) Confusion matrix (b) precision, recall, f1 scores

Figure 7: Test set size 70%

Even if we remove duplicates decision tree model mantain very good results.

5.3 Dataset with duplicates and svm as model

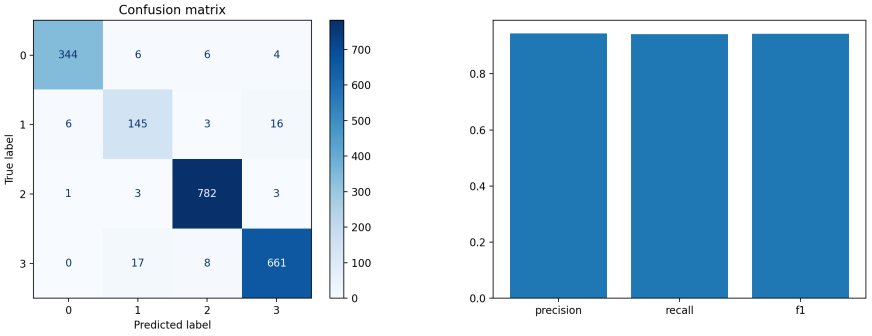


(a) Confusion matrix (b) precision, recall, f1 scores

Figure 8: Test set size 33%

The SVM model seems to perform worse than the Decision Tree model, as we can see from the confusion matrix in figure (8) this model often confuses the first class, that is Sort, with the third class, that is String (and vice versa), and in fact the f1 score that we obtain is about 0.89, which is lower then the f1 score obtained by Decision Tree (almost 1) in both duplicate and no duplicate datasets.

5.4 Dataset without duplicates and svm as model



(a) Confusion matrix (b) precision, recall, f1 scores

Figure 9: Test set size 33%

As we can see for this model, removing duplicates leads us to better results. The model does not confuse anymore String class with Sort class, but and we get all the three score very high (about 0.96 for each kind of score).

6 Conclusion

In conclusion we can say that decision tree for this problem seems to perform better also with a (relatively) small amount of data and duplicates does not affect so much the performance of the model. SVM, instead, seems to perform better without duplicates. The kernel that seems to work better for this model is the linear one with parameterized with $C = 1$. Other kernels lead the model to very low scores around 0.6, and increasing the C value slow down the fit process.
When I was comparing the results I did not focus too much on the importance of the various scores as they all returned more or less the same values in each case.

To conclude the best of the two models seems to be the decision tree, which even with a small test set is able to bring excellent results.