DEVELOPMENT OF AN ASSEMBLY X86-64 FUNCTIONS CLASSIFIER

MACHINE LEARNING HOMEWORK 1 REPORT

Gioele Migno - 1795826

11/28/2020

ABSTRACT

The project consists of the development of a classifier that is able to classify a function, written in Assembly x86-64 code, according to the type of operation that it performs, in particular, the typologies of interest are: Encryption, Math, String, Sort. The goal of the project is to provide a useful tool that facilitates the analysis of a program by its Assembly code.

PRELIMINARIES

During the development of the project, several methods of classical machine learning have been tested, in particular: KNN, SVM Kernel, and Random Forest. For each of them, some tests have been performed in order to obtain the best value for the hyperparameters.

For each method, the hyperparameters that have been changed are described below:

KNN

- `n_neighbors`: Number of neighbors used to predict new samples

SVM Kernel

- `kernel`: Type of kernel used
- `C`: Regularization parameter. The strength of the regularization is inversely proportional to C.

Random Forest

- `n_estimators`: The number of trees in the forest.

PREPROCESSING

FEATURES EXTRACTION

The dataset used is in JSON format and contains for each sample (Assembly function) the following information:

- `id`: Code that uniquely identifies a sample within the dataset
- `lista_asm`: Assembly code as a list in which each element is a line
- `cfg`: (Control Flow Graph) It is the control flow of the function represented as an adjacent list of a graph
- `semantic`: Label that indicates the type of function

In order to use this dataset to train a machine learning algorithm, it is necessary to perform vectorization of the samples, namely, extract the features and represent each sample as a vector of numbers.

The vectorization has been performed by considering both the Assembly code and the control flow graph.

Control Flow Graph Analysis

The extraction of the features from the cfg has been performed by using networkx Python library, that allows to extract the following information:

- `num_loops`: Number of loops inside the graph
- `num_edges`: Number of edges
- `num_nodes`: Number of nodes
- `cyclomatic_complex`: Cyclomatic Complexity calculated as following:

$$CC = E - N + 2P \qquad E = \#\ edges \quad N = \#\ nodes$$

$$P = Connected\ Components\ (=1)$$

`num_loops` and `cyclomatic_complex` capture the complexity of the code, so for example, a Math function has a number of loops and a cyclomatic complexity less than an Encryption function.

Assembly Code Analysis

In order to extract features from an Assembly code, it is necessary to analyze the instructions and the registers used. Regarding the instructions, it is interesting to count the number of occurrences of a specific instruction and also the sum of the occurrences of each type of instruction (e.g. data movement or jump). In the case of registers instead, the interesting things are the occurrences of each register and the number of different registers used, this last is based on the assumption that a complex program uses more different registers than a simpler one.

To make the Assembly Analysis easily editable and adaptable to a different set of instructions according to another architecture like ARM, the analysis program uses as input two different types of text file that indicates the keywords to search and collect:

- `.instr`: Contains the keywords associated with the instructions to looking for.
- `.reg`: Contains the keywords associated with registers.

The program parses a line of code by dividing it into two parts:

[**INSTRUCTION**] - [SOURCE OPERAND - DESTINATION OPERAND]

The keywords contained in `.instr` files are searched in the first part of the line and the keywords of `.reg` files in the second part. A keyword of this last type can be found more times in the same line, so each occurrence is considered.

For example, considering two files:

The features extractor, regarding only the keywords to search, represents an assembly code considering the following components:

- `jmp` //number of occurrences of jmp instruction
- `je`

- `jump` //the name of this feature corresponds to the filename
  //It contains the sum of all occurrences of the keywords defined in that file
- `eax` //occurrences of eax
- `ecx`
- `registers` //number of different keywords found (different registers). In this case the maximum value is two.

About the registers, sub-registers are considered independents, for example, %ax, %ah, and %eax are counted as different registers.

There are also considered the number of memory accesses (by counting the occurrences of '['), and the number of lines.

In summary, an Assembly code is vectorized in the following way:

- **`num_loops`** of the cfg
- **`num_edges`** of the cfg
- **`num_nodes`** of the cfg
- **`cyclomatic_complex`** of the cfg
- **`memory_access`**
- **`num_asm_lines`**
- <list of keywords defined in the keywords files>

The features related to the keywords appear in alphabetical order according to the filename and at first, there are the occurrences of each keyword of a file and then the feature with the same name of the file that in the case of .instr files, represents simply the sum of all occurrences. The chosen keywords and the categories are based on *x64 Cheat Sheet Fall 2019* document, however, several special registers and instructions used in the dataset are not described in that document, so the other ones have been manually classified as shown in *[PREPROCESSING_1]_Analysis_ASM_instructions.ipynb* Jupyter notebook.

In the dataset exported, the types of function (labels) have been coded as an integer with the following mapping:

- **`0`**: undefined (used to convert samples of the blind test)
- **`1`**: encryption
- **`2`**: string
- **`3`**: sort
- **`4`**: math

DATASET EXPORT

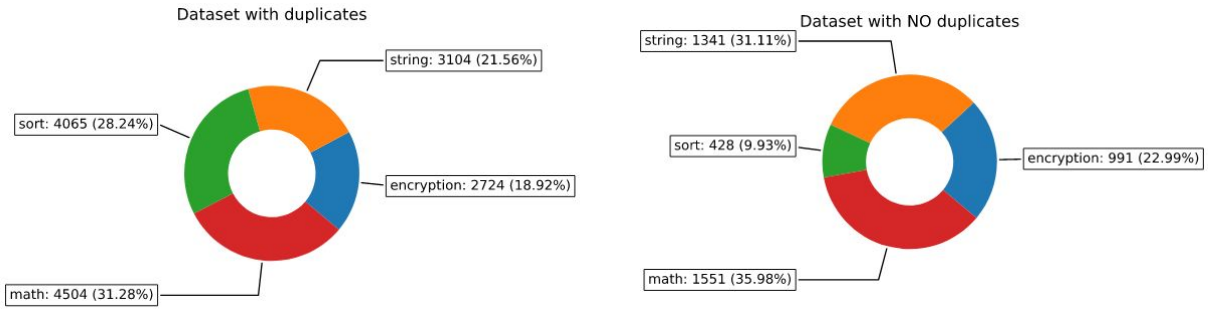
Once performed the vectorization of all samples, the new dataset has been exported as a JSON file with the following keys:

- **`x_meaning`**: It is an array of string that indicates the meaning of each features
- **`y_meaning`**: It is an array of string that indicates the meaning of each class indicated in the dataset by an integer
- **`dataset`**: Contains the samples, for each of them:
    - **`ID`**: Identifier of the sample from the original dataset
    - **`x`**: Sample as a vector of features
    - **`y`**: Integer that indicates the class

The original dataset has been generated by using different source files and compiling some of them several times but using different build flags. This method can generate duplicates or very similar assembly files that after the vectorization become identical. Therefore the dataset vectorized contains several duplicates.

In order to analyze the difference between the use of a dataset with or without duplicates, both have been used and their performances have been compared using the same validation set. The validation set has been generated by the dataset without duplicates by splitting it into 85% training set and 15% validation set, the split has been performed so that the distribution of each class is the same of the dataset without duplicates. Finally, the training set with duplicates has been obtained by removing the sample of the validation set from the dataset.

Figure 1 shows the distribution of classes in the dataset with duplicates (a) and without duplicates (b). In Figure 2 instead, are represented the splits of both datasets (training set with and without duplicates and validation set).



(a) (b)

Figure 1: Distribution of classes in datasets

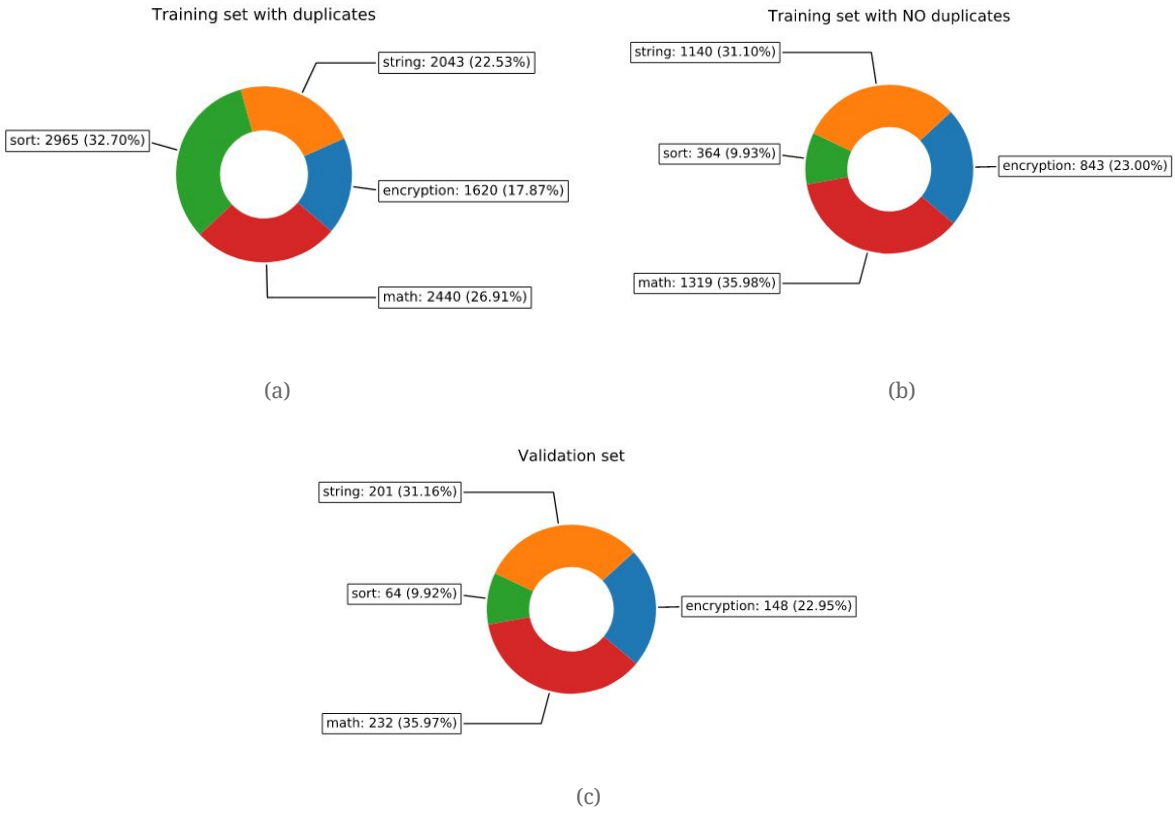


Figure 2: Distribution of classes in training sets and validation set

Reduced Datasets

After the vectorization each sample is represented as a vector of 301 components, it is possible to reduce the dimensions of a sample by selecting the most important features. The features selection can be performed by a Random Forest model trained with the dataset, indeed, during the training, the algorithm measures the importance of a feature according to how much that feature has decreased the impurity of a tree node.

Both of the datasets (with or without duplicates) have also been tested using the reduced version.

In the case of the dataset without duplicates, the features with at least 0.01% of importance have been selected. The reduced samples have 137 components and the features selected represent 99.86% of the importance of all features.

In Table 1 are shown the ten most important features with their description and the importance. It is interesting to note that these features represent about 36.36% of the importance of all features.

FEATURE NAME	DESCRIPTION	IMPORTANCE
`xmm0`	Number of uses of the register	≈4.72%
`xmm`	Number of different xmm registers used	≈4.50%
`xmm1`	Number of uses of the register	≈4.08%
`binary_op`	Number of uses of binary operation (e.g. lea, add, xor)	≈3.83%
`rdi`	Number of uses of the register	≈3.65%
`ucomisd`	Number of uses of the instruction	≈3.39%
`xor`	Number of uses of the instruction	≈3.23%
`js`	Number of uses of the instruction	≈3.16%
`eax`	Number of uses of the register	≈2.99%
`jp`	Number of uses of the instruction	≈2.81%
		≈36.36%

Table 1: The ten most important features

EXPERIMENTS

The project has been developed by considering several classical machine learning methods and for each of them, some hyperparameters have been tweaked to obtain the best performance on the validation set. The metrics used to compare the performance are Accuracy and F1 Score with the weighted average between classes. This last has been chosen in order to consider both recall and precision in equal measure since in our task both of them are relevant, in other cases it is necessary to keep in mind the precision-recall tradeoff.

As mentioned above, the models have been tested in both training sets, with or without duplicates, moreover, for each of them also the version with reduced number of features has been considered.

In this part are described the methods used by highlighting the different performance achieved, in the next part instead, are shown in detail all results obtained.

KNN

The only hyperparameter tested is the number of neighbors considered during the prediction of a new sample, values from 1 to 50 have been tested.

Training set without duplicates

Considering the training set without duplicates, in both datasets, standard and with features reduction, the best number of neighbors is 1. The best results for the validation set have been obtained with the standard dataset: Accuracy = 0.9039, F1_Score = 0.9046.

Figure 3 shows the performance according to the number of neighbors used.

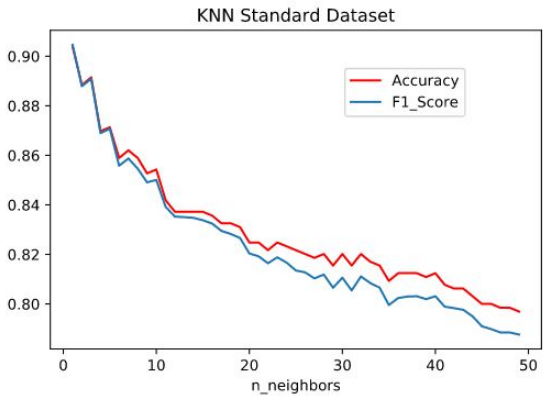


Figure 3: Performance on validation set according to number of neighbors

Training set with duplicates

Using the dataset with duplicates, the best value of the hyperparameter is 7 neighbors, the higher performance has been obtained with standard dataset: Accuracy: 0.9070, F1 Score: 0.9096

Figure 4 shows the results with the standard dataset by changing the number of neighbors.

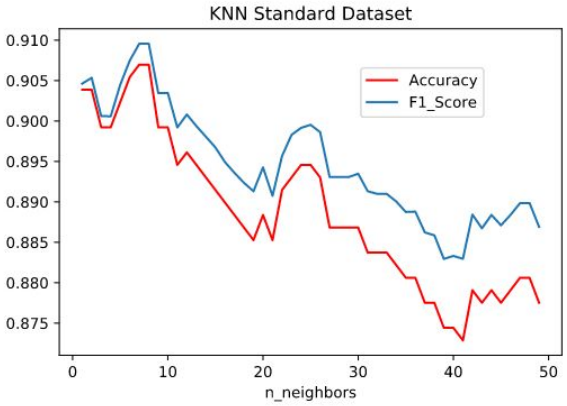


Figure 4: Performance on validation set according to number of neighbors

Using a KNN model, the higher performance has been obtained using the standard training set with duplicates. Accuracy: 0.9070, F1 Score: 0.9096

SVM

The hyperparameters tweaked of the SVM model are the type of kernel used and the regularization parameter C.

At first, the performance has been compared according to the kernel, the best results have been obtained by using linear kernel in both types of dataset with features reduction. Therefore, using the linear kernel, the regularization parameter C has been tweaked.

Training set without duplicates and features reduced

Using a linear kernel, the best performance has been obtained using the dataset with features reduced. The best value for C is 0.5 with: Accuracy=0.9473, F1 Score=0.9486.

In Figure 5 are shown the performance in the training set and validation set, these plots are used to check possible overfitting.

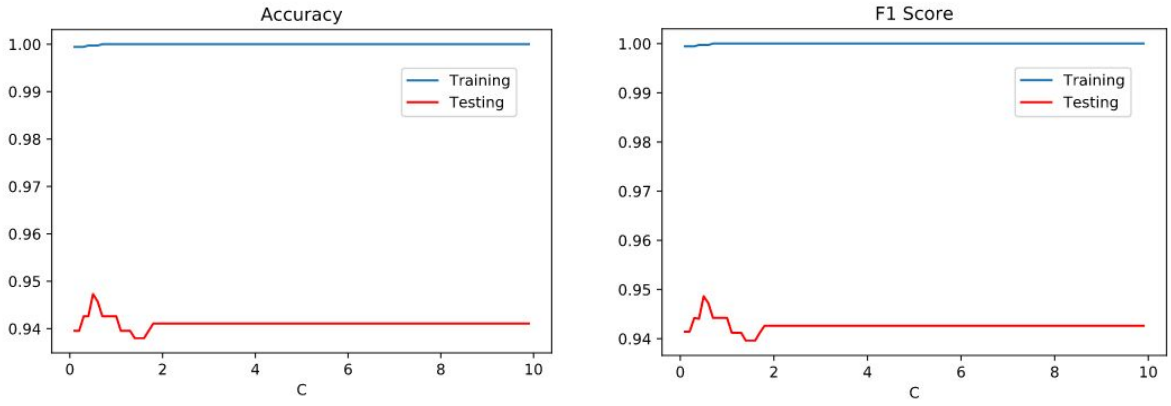


Figure 5: Performance on training and validation set according to C hyperparameter

Figure 6 shows the performance only in the validation set.

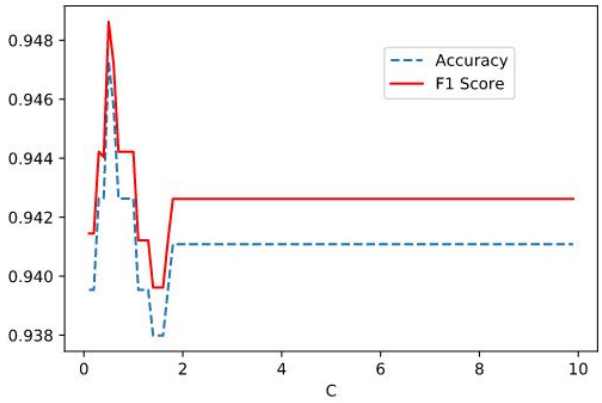


Figure 6: Performance on validation set according to C hyperparameter

Training set with duplicates and features reduced

Also using the training set with duplicates, the best results have been obtained with linear kernel and the reduced dataset, the higher performance is achieved settings C=0.2 (Accuracy=0.9473, F1_Score 0.9486).

Figure 7 shows performance in the validation set according to C.

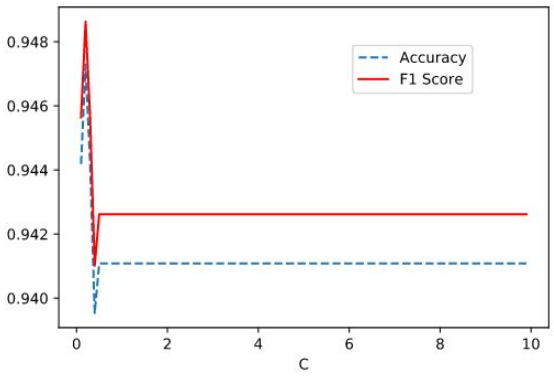


Figure 7: Performance on validation set according to C hyperparameter

Using an SVM model, the performance is the same with the dataset with or without duplicates and features reduced. (Accuracy=0.9472, F1_Score 0.9486).

RANDOM FOREST

The hyperparameter changed in the random forest model is the number of trees (n_estimators). The values that have been tested are 1 to 1000 with a step of 10, the different models have been initiated with the same random seed.

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                       max_depth=None, max_features='auto', max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=491,
                       n_jobs=-1, oob_score=False, random_state=20201123,
                       verbose=0, warm_start=False)
```

Training set without duplicates

The best number of estimators for the model trained with the dataset without duplicates is 491 and the performance obtained are: Accuracy=95.35, F1_Score=95.58).

In Figure 8 are shown the performance in the training set and validation set.

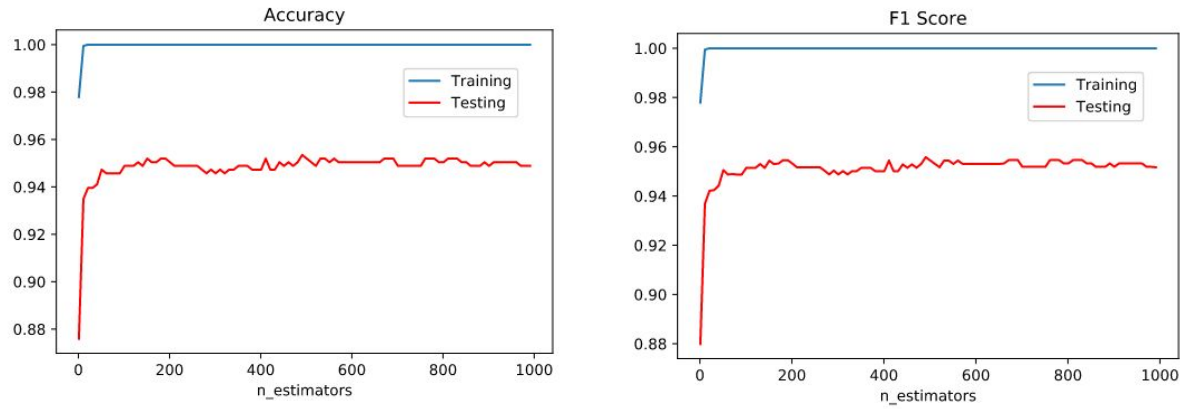


Figure 8: Performance on validation set according to the number of trees used

Training set with duplicates

The better performance has been achieved using the training set with duplicates, moreover, fewer estimators than the previous case are needed, indeed the higher results are obtained with only 371 trees. (Accuracy=0.9597, F1_Score=0.9616).

In Figure 9 are shown the performance in the training set and the validation set.

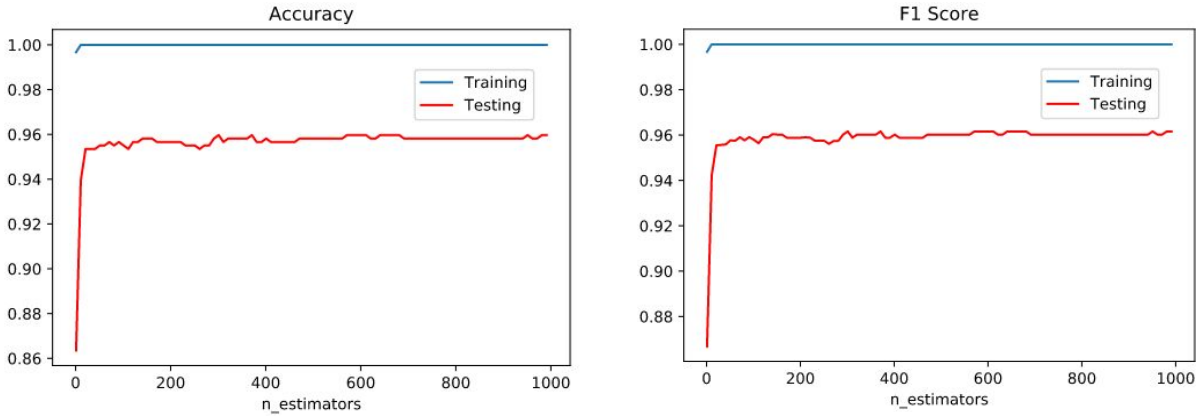


Figure 9: Performance on training set and validation set according to the number of trees used

The best performance using a Random Forest is obtained with the dataset that contains duplicates (Accuracy=0.9597, F1_Score=0.9616), the reason why could be the fact that the duplicates in the dataset add more diversity to each sub-training set used by the trees and the bootstrap in the dataset without duplicates is not enough since there are relatively few samples.

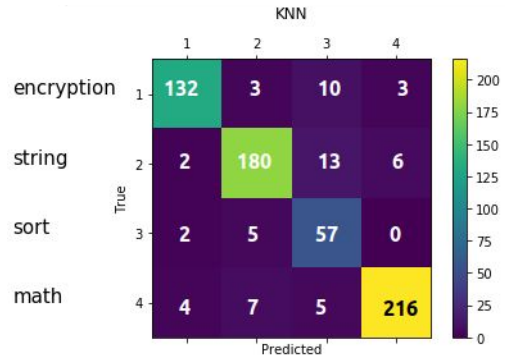
RESULTS

In the following are shown the results of each experiment described above.

KNN

	n_neighbors	STANDARD DATASET		FEATURES REDUCED	
		Accuracy	*F1 Score*	*Accuracy*	*F1 Score*
TRAIN NO DUPL.	1	90.39%	90.46%	90.23%	90.30%
TRAIN WITH DUPL.	7	90.70%	90.96%	90.54%	90.80%

Confusion Matrix of the best model



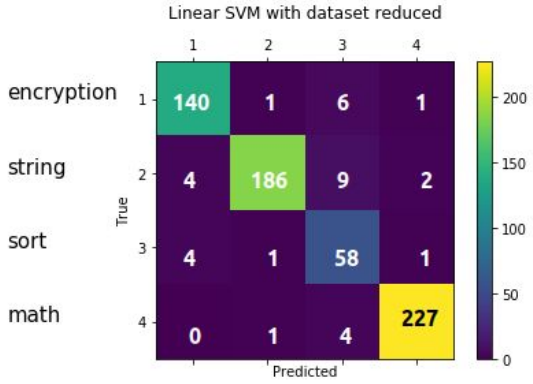
SVM

STANDARD DATASET	KERNEL [C=1]							
	poly		linear		rbf		sigmoid	
	Acc.[%]	*F1[%]*	*Acc.[%]*	*F1[%]*	*Acc.[%]*	*F1[%]*	*Acc.[%]*	*F1[%]*
TRAIN NO DUP.	91.32	91.50	93.95	94.12	83.57	83.57	21.24	17.88
TRAIN WITH DUP.	91.63	92.08	93.80	93.96	84.19	84.49	12.71	11.42

FEATURES REDUCED	KERNEL [C=1]							
	poly		linear		rbf		sigmoid	
	Acc.[%]	*F1[%]*	*Acc.[%]*	*F1[%]*	*Acc.[%]*	*F1[%]*	*Acc.[%]*	*F1[%]*
TRAIN NO DUPL.	91.78	91.98	94.26	94.42	78.45	77.95	27.29	16.41
TRAIN WITH DUPL.	91.47	91.85	94.11	94.26	79.84	79.98	9.46	2.00

BEST RESULTS	KERNEL	C	FEATURES REDUCTION	
			Accuracy	*F1 Score*
TRAIN NO DUPL.	linear	0.5	94.73%	94.86%
TRAIN WITH DUPL.	linear	0.2	94.73%	94.86%

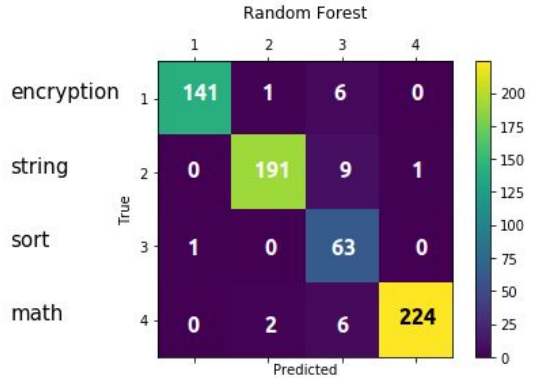
Confusion Matrix of the best model



RANDOM FOREST

BEST RESULTS	n_estimators	STANDARD DATASET	
		Accuracy	*F1 Score*
TRAIN NO DUPL.	491	95.35%	95.58%
TRAIN WITH DUPL.	371	95.97%	96.16%

Confusion Matrix of the best model



CONCLUSIONS

In order to develop an Assembly x86-64 function classifier, several classical machine learning algorithms have been tested, the higher performance has been achieved by using a Random Forest Classifier trained with the dataset with duplicates. Through that model the results obtained are: Accuracy=95.97%, F1_Score_weighted=96.16%. Looking at the corresponding confusion matrix, it can be observed that most misclassifications are caused by the wrong classification of a function as Sort type. These mistakes are due to the fact that the dataset used (training set with duplicates) has more samples with Sort type than the other ones (32.70%). This imbalance leads the model to predict more frequently a sample as a Sort function.

Hence, to improve the performance is required to use a balanced dataset and increase the number of samples. However, in order to achieve much higher performance is necessary to change the method used during the vectorization of an Assembly code, indeed through the analysis of the keywords, it is not possible to highlight the local priorities between lines of code, to get these properties it is necessary to use an approach used in natural language processing like LSTM. Another possible method is to convert a line of code into a row vector and represent a function as a vertical image, then use a feature extractor like a Convolutional Neural Network. The images with different lengths can be adapted using padding, however, this method requires to set a maximum number of lines that can be analyzed and if most functions in the dataset are short, then the training could be very slow.

REFERENCES

[1] Aurélien Géron **Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition**