Homework 1

ML Classifier for functions

Martina Evangelisti 1796480

November 15, 2020

Contents

1 Introduction

The goal of the learning algorithm described in this report is to classify binary functions belonging to four different classes: encryption, string manipulation, math and sorting.

1.1 Dataset

The dataset used to train and test this algorithm is made of 14397 functions and it is given as a JSON file having a json object, representing a function, for each line. Each function is described with an id, a label called *sematic* which tells the class to which the function belongs, a field called *lista_asm* containing the list of the function's assembly instructions and the *control flow graph*.
The other dataset available is called *blind test* and contains 757 function without the relative labels. These functions will be classified by the developed algorithm.

2 Preprocessing

2.1 Processing the original dataset

I processed the original dataset in order to extract the features i considered significant for the classification.
I analyzed the dataset's JSON file line by line and i saved the processed dataset in a new file called *processed_dataset.json*.

The features i considered are related to the characterization of the four classes: the functions manipulating strings use comparisons and memory swaps, math functions use floating point instructions and *xmm* registers, the encryption function are characterized by a complex *cfg* and use a large number of *xor* and the sorting functions often call external functions.
I analyzed both *lista_asm* and *cfg* as it follows.

First of all i extracted the field *lista_asm* from the dataset: from that list i decided to take into account only the assembly instructions. For every instrucion a function *decide_class* is called.
This function takes the main assembly instructions without the variable postfix which depends on the particular type of data that the instruction refers to.
Morover, i classified all the instructions applying a shift operator under the same label called *shift*, all the floating point operations under the label *float* and all the conditional jumps under the label *jumpc*.

I decided not to consider all the registers and the addresses contained in *lista_asm* except the *xmm* register and the words *byte*, *qword* and *dword* because i considered them worthwhile for the classification.

Regarding the control flow graph i used *networkx* [1] python library to analyze the graph features.
I filled the processed dataset with information about the number of edges and i added a field containing a value set to 1 if the graph has cycles and 0 if not.
Using *momepy* [2] library i computed the cyclomatic complexity and added it as another feature.

At the end the processed dataset is a JSON file containing, for each line, the main features of a function as a JSON object with the following fields:

- id
- edges
- cycles
- cyclomatic
- instructions

2.2 Vectorization

The file containing the processed dataset is read using *Pandas* library [3].
I instanciated a Count Vectorizer, made available by *sklearn* library [4], which converts text documents to a matrix of token counts creating a dictionary with the vocabulary found analyzing the data.
I set the parameter *min_df* to 0.005 in oder to ignore terms with a frequency in the documents strictly lower than this value.
I fitted the vectorizer with the field *instructions* of the processed dataset in order to obtain the occurrences of each instruction in each function.

2.3 Final Dataset

I added to the data obtained by the vectorizer the control flow graph features using the *scipy* [5] library. I created a matrix for all the features: one for the number of edges, one for the cycles and another for the cyclomatic complexity and i created the final set *X_all* by merging this sets with the *hstack* function.
This structure can bee seen in Figure 1.

(14397, 51)

	add	and	bswap	bt	byte	call	...	xmm	xor	edges	has_cycles	cyclomatic_c
0	8	0	0	0	1	12	...	0	0	30	0	8
1	0	0	0	0	0	2	...	21	0	20	0	6
2	24	102	0	0	58	0	...	0	24	33	1	16
3	0	0	0	0	0	2	...	0	0	1	0	0
4	4	0	0	0	0	14	...	0	18	35	1	14
5	4	22	12	0	0	60	...	0	152	62	0	28
6	42	76	0	0	33	0	...	44	32	40	1	20
7	0	44	0	0	17	0	...	0	44	4	1	2
8	8	6	0	0	0	0	...	0	2	26	1	12
9	10	0	0	0	0	14	...	0	4	16	0	3

Figure 1: example of X_all structure

The final dataset, in the form:

$$D = \{(x_n, t_n)\} \tag{1}$$

is made of *X_all*, computed as explained previously, and *Y_all* which is taken from the field *semantic* of the dataset.

3 Learning algorithms

Once defined the dataset, the first thing to do is to split it into two parts: one for the training, and the other one for testing. I let the 70 % of the dataset be used for the training phase and the remaining 30 % for testing.

3.1 Training

I used and compared three different machine learning models:

- BernoulliNB
- MultinomialNB
- Support-vector machines

SVM is used with a linear kernel and C parameter set to 1. The parameter C is used to have a trade-off between low training errors and low testing errors balancing the maximum margin and the times it's possible to violate margin's constraint. C has to be balanced in order to generalize well when predicting on new data, if it's low we are allowing more outliers and encouraging a larger margin, in order to not have a model overfitted on the dataset.

To do the training i fitted each model giving as input the training subsets labeled *(x_train, y_train)*.

Using *time* library i measured the time spent to train the models.

model	**training running time**
bernoulliNB	0.04593944549560547 sec
multinomialNB	0.05290484428405762 sec
SVM	31.868048191070557 sec

Both bernoulli and multinomial are Naive Bayes classifiers, in fact their training running time has the same order of magnitude.
The difference is that MultinomialNB works with occurrence counts,while BernoulliNB analyzes boolean features. In facts Multinomial provides the frequency of a word occurrence in a text, while Bernoulli considers if a word occurs in a text or not.
As shown in the previous table, SVM is significantly slower than the other two models.
This is due to the fact that the time to train this model scales quadratically with respect to the number of samples in input.

3.2 Prediction and Evaluation

The model previously trained is used to predict the data in the test subset. To evaluate the models's predictions i used the confusion matrix and the classification report made available by *sklearn* library [4].

BernoulliNB:

	precision	recall	f1-score	support
encryption	0.90	0.81	0.85	810
math	0.91	0.90	0.91	1395
sort	0.81	0.87	0.84	1187
string	0.85	0.85	0.85	928
accuracy			0.87	4320
macro avg	0.87	0.86	0.86	4320
weighted avg	0.87	0.87	0.87	4320

accuracy_score: 0.8668981481481481

Figure 2: Bernoulli classification Report

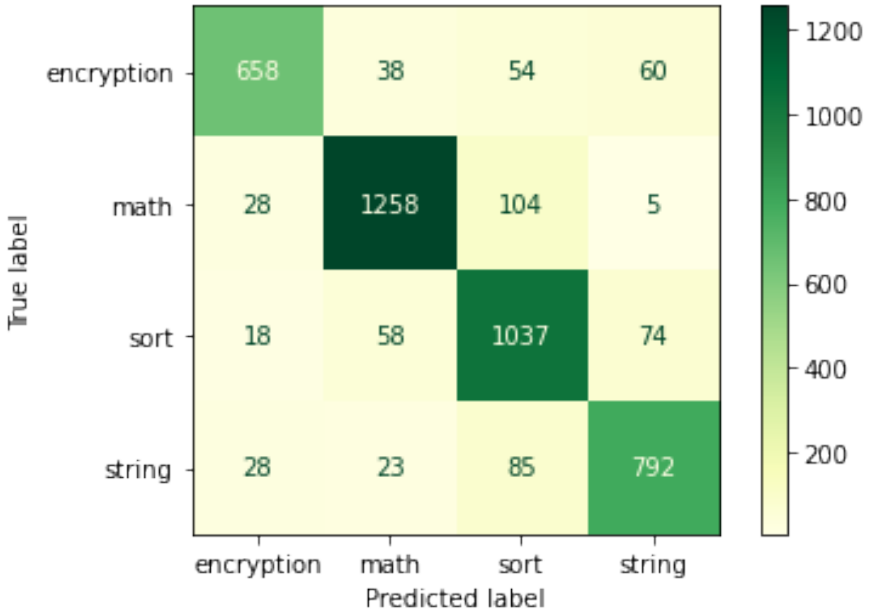

Figure 3: Bernoulli confusion matrix

.

MultinomialNB:

	precision	recall	f1-score	support
encryption	0.98	0.78	0.87	810
math	1.00	0.88	0.93	1395
sort	0.80	0.88	0.84	1187
string	0.76	0.93	0.84	928
accuracy			0.87	4320
macro avg	0.88	0.87	0.87	4320
weighted avg	0.89	0.87	0.87	4320

accuracy_score: 0.8715277777777778

Figure 4: Multinomial classification Report

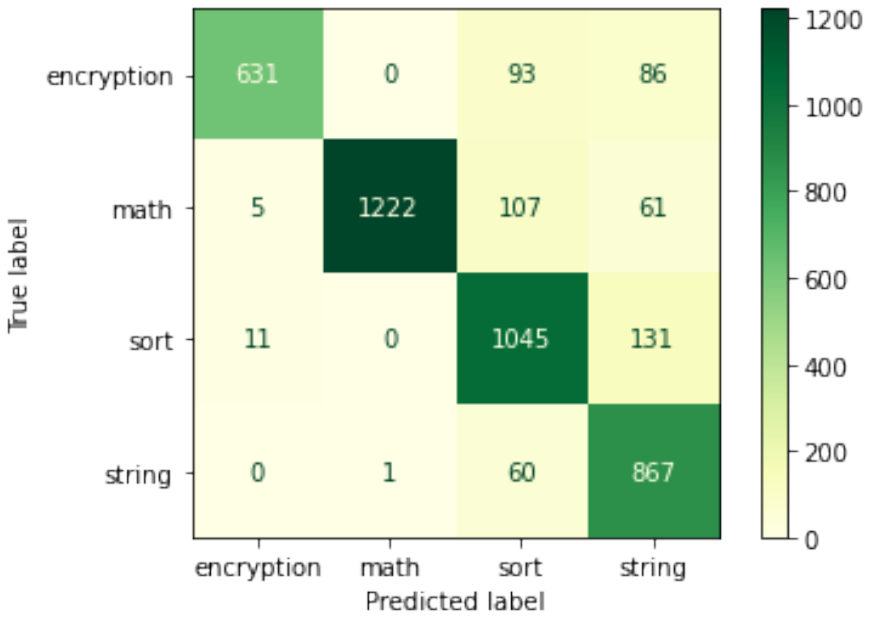

Figure 5: Multinomial confusion matrix

.

SVM:

	precision	recall	f1-score	support
encryption	1.00	1.00	1.00	810
math	1.00	0.99	1.00	1395
sort	0.95	0.99	0.97	1187
string	0.99	0.94	0.96	928
accuracy			0.98	4320
macro avg	0.98	0.98	0.98	4320
weighted avg	0.98	0.98	0.98	4320

accuracy_score: 0.98125

Figure 6: SVM classification Report

.

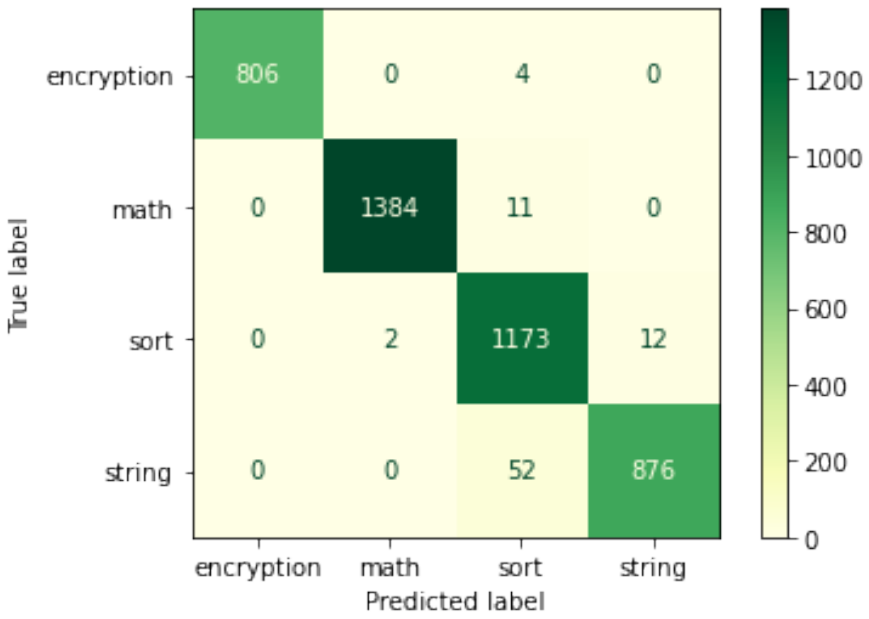

Figure 7: SVM confusion matrix

As we can see from the plots above, MultinomialNB and BernoulliNB have very similar performaces, the first one has a slightly higher accuracy value. SVM, instead, performs significantly better than the other two models.

Looking at the cells of the confusion matrix outside the diagonal, we can see the errors in classification and understand which classes are more confused. In all three models we can observe that encryption and math functions are easily recognizable, on the contrary there are more errors in the classification of sorting and string manipulation functions.
This is due to the fact that the features of encryption and math are very accurate: for example the use of the xor operator for the first one and floating point operations and xmm register for the second one.

4 Final Model

The final model i chose [1] is the one which performed better in the evaluation and it's linear SVM.
I evaluated the performances of this model using k-Fold Cross Validation, splitting the dataset first into 5 samples and then into 10 samples and i obtained almost the same performances in the different folds.

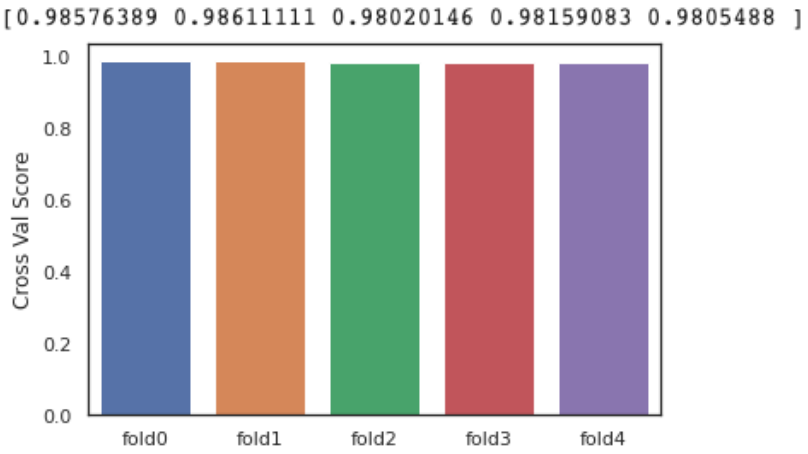

Figure 8: Cross Val Score of the model K=5

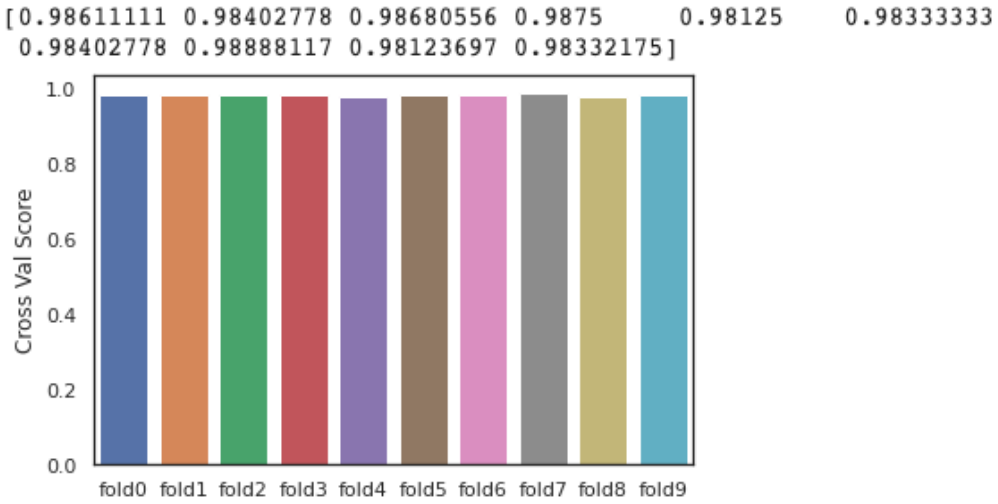

Figure 9: Cross Val Score of the model K=10

[1] file: model.ipynb

This methods helps understanding if the result obtained from a single evaluation based on a train set and a test set may be obtained from a biased test.
By training k different models we have a more general and reliable evaluation. Each point can be tested once and it's used in training k-1 times.
The disadvantage of this method is that it is really slow, in fact the training algorithm is applied k times, so it takes k times the average running time spent in training. Since the computational complexity of SVM is high this method takes a lot of time.

5 Blind Test

The blind test is given with the same format of the dataset and the preprocessing phase is applied in the same way.
To predict the given samples i used the model previously explained. The predictions made by the model are stored into a file [2] with one prediction per line.

6 Conclusions

This report compared MultinomialNB, BernoulliNB and SVM models in function classification and the results show that SVM achieves the best performances.
As we could see, machine learning it's a powerful tool to classify different classes of functions.
This ability is useful when reversing a software that is unknown in order to understand which function compose it. For example an important field of application could be malware analysis.

[2] file: 1796480.txt

References

[1] Networkx documentation
`https://networkx.org/documentation/stable/reference/readwrite/json_graph.html`

[2] Momepy documentation
`http://docs.momepy.org/en/stable/api.html`

[3] Pandas documentation
`https://pandas.pydata.org/docs/reference/io.html#json`

[4] SKLearn: Skikit learn documentation
`https://scikit-learn.org/stable/modules/classes.html`

[5] Scipy documentation
`https://docs.scipy.org/doc/scipy/reference/`

[6] Machine Learning and security research.
Giuseppe Antonio Di Luna