Report

Homework 1. Function classification

Prepared by:
Sorokoletova Olga
1937430

29 Nov 2020

Task definition

Given a dataset with the represented assemply functions labeled by 4 classes (encryption, math, sort and string) perform and compare 2 different models for the function classification task. Given a blind test file deliver a text file with the predictions.

Data

Input data

noduplicatedataset.json - Dataset 2 (without duplicates) - for training and evaluation.
Total number of items: 6073.
blindtest.json - from the Dataset 1 folder - for blind test.
Total number of items: 757.

Output data

1937430.txt - output of the blind test (provided with this report and ZIP file used in the project).

Approach

The following approach to obtain the solution was considered: two different models were obtained by two different ways of data pre-processing (more precisely, feature extraction procedure): with and without usage of the control flow graph information. Then preprocessed datasets were split for the training and evaluation, the training ones were used to train the same classifier - SVM with the linear kernel and regularization parameter equal to 1. The best implemented model was used to generate output on the blind test dataset. The results of the evaluation procedure will be discussed below.

Pre-processing

Note: applying the kernelized SVM suggests normalization as a part of pre-processing procedure, but here all of the units appear to be the same and linear kernel is employed, so normalization will not bring any profit.

Firstly, for both models the initial dataset should be converted from json into pandas dataframe format.

Model 1

In the Model 1 features of the control flow graph are not considered and features extraction procedure completely based on the ‘lista_asm’ strings (lists) of the input items. The following feature columns of dataset were introduced (assigned in the ‘lista_asm’ string instructions were counted without repeated counts of those instructions which are included into cycles, i.e. calculated as a simple number of occurrences of a substring in a string):

Groups of instructions

`x_data['data_movements']` - the number of instructions that move data: instructions with one or two suffixes (mov - independently from if it occurs to move source to destination or to move byte to word, push - independently from if it occurs to push source onto stack or to move byte to word, pop), instructions with no suffixes (cwtl, cltq, cqto) and the only instruction that performs memory addressing calculations but doesn't actually address memory (lea). Conditional move instructions are also included into this group (everything which is started from ‘cmov’: cmove/cmovz, cmovne/cmovnz, cmovs, etc.)

`x_data['arithmetic']` - the number of the arithmetic instructions: unary arithmetic operations (inc, dec, neg, not), binary operations (leag, add, sub, mul, imul) and special arithmetic operations (imulq, mulq, idivq,

divq). Here bitwise binary operations and shifts are excluded in order to aggregate them in other features. This feature is considered in order to recognize the class of maths functions.

`x_data['shifts']` - the number of shift operations (sal, shl, sar, shr). This feature is considered in order to distinguish encryption functions since they use shifts frequently.

`x_data['comparisons']` - the number of comparison and test instructions (cmp, test) which are used a lot in string manipulations and also employed in sort functions (but these functions are not usually complex, so the number of comparisons differs from the number of comparisons in string manipulations).

`x_data['jumps']` - the number of jump instructions (everything that starts from ‘j’: jmp - to label or to specified location, je/jz, jne/jnz, js, jns, jg/jnle, jge/jnl, jl/jnge, jle, jng, etc.). Jump instructions are the common way to introduce nested For and If in the assembly language a lot of which are used by encryption functions and one or two by others, particularly, by sort functions.

`x_data['calls']` - the number of procedure call instructions (call - push return address and jump either to label or to specified location, leave, ret). They may characterize general complexity of the function and high complexity will then correspond to encryption function.

`x_data['bitwise']` - the number of bitwise instructions which are generally a part of an arithmetic operations group, but are separated here into a special group as soon as encryptions use a lot of them and a lot of xor especially (xor, or, and).

Groups of registers

`x_data['func_reg']` - the number of occurrences of registers %rdi, %rsi, %rdx, %rcx, %r8, and %r9, because they are used to pass the first six integer or pointer parameters to called functions and so they a helpful addition to the number of call instructions feature.

`x_data['stack_reg']` - the number of occurrences of %rbp and %rsp registers to identify manipulations with the stack (register %rsp is used as the stack pointer, %rbp is used to keep track of the base of the current stack frame).

`x_data['xmm_reg']` - the number of occurrences of the registers xmm* (i.e. which are started from ‘xmm’) supposing that they are employed as special registers for mathematical operations and will help to recognize math functions.

Calculations of all of the listed above features for Model 1 are implemented into a single cycle in the code of the programme.

Total number of features of the Model 1 is equal to 10.

Model 2

As will be shown in the results of the evaluation procedure usage the ‘lista_asm’ lists of assembly instructions of each functions only is not reasonable insofar as they are not complete source of data and can not represent general complexity of the function well which should be done in order to obtain high quality of classification. Because of that additionally to the classes of instructions in the Model 2 features of the control flow graph are considered.

The control flow graph of the input as a networkx graph. The features of it are obtained by usage of the networkx library methods and finally the following set of columns contains them:

```
x_data_copy['number_of_nodes'], x_data_copy['number_of_edges'],
x_data_copy['diameter'], x_data_copy['complexity'],
x_data_copy['simple_cycles'].
```

First of all, there is a block of code which takes nx_graph - graph from adjacency data format resulted from json_graph.adjacency_graph (saved into dataset in order to extract 'simple_cycles' feature later on) and gets from it some basic knowledges foreseen by the library, namely `x_data_copy['number_of_nodes'] -` the total number of nodes in the graph (the first feature), `x_data_copy['number_of_edges'] -` the total number of edges (the second feature), `x_data_copy['parts'] -` the number of strongly connected components to calculate graph cyclomatic complexity, `x_data_copy['diameter']`(the third feature) - graph diameter, i.e. the number equal to the distance between the most distant vertices of the graph (networkx's 'diameter' function requires an undirected graph as input, hence graphs should be transformed into this format).

Then the cyclomatic complexity `x_data_copy['complexity']` - measure of the number of linearly independent paths (the fourth feature) is calculated by the formula for the strongly connected graphs, i.e. where the cyclomatic complexity of the program is equal to the cyclomatic number of the graph: $M = E - N + P$ (E - the number of edges of the graph, N - the number of nodes of the graph, P - the number of strongly connected). This may be also seen as calculating the number of linearly independent cycles that exist in the graph (those cycles that do not contain other cycles within themselves).

Finally, the most complicated calculations are employed in order to extract the fifth feature - the number of the simple cycles `x_data_copy['simple_cycles']`.
In general, networkx library provides users with the `nx.simple_cycles` method which suggests as a simple cycle (or elementary circuit) a closed path where no node appears twice, except that the first and last node are the same. Two elementary circuits are distinct if they are not cyclic permutations of each other. This function applies to the networkx graph and returns a generator to be converted into a tuple or list length of which we can obtain and save the number of simple cycles.

However, as noted in the documentation, the problem is the time complexity of the function which results in extremely long execution time: $O((N+E)(C+1))$. I was trying to handle this problem through the switching to the GPU provided by the google colab, but even that could not help to obtain satisfiable execution time. Meanwhile, experiments have shown that calculations ‘get stuck’ exactly on the encryption functions and nowhere else since exactly encryption functions contain a huge number of simple cycles.
Summarizing, a long execution time of the simple cycles calculation directly correlated with the number of simple cycles and moreover appears to be the distinguishing feature for the encryption recognition. Therefore, instead of calculating the number of simple cycles in such cases this value in the column may be filled with the maximum number of simple cycles for encryption functions amongst the samples where this number may be obtained with the reasonable time, namely:

```
default_enc = x_data_copy_clear[x_data_copy_clear['semantic'] ==
'encryption']['simple_cycles'].max(),
```

meaning default value for too complex encryption functions obtained by those ones which are less complex, where too complex encryption functions are excluded from the `x_data_copy_clear` dataframe and included into so to be said ‘black list’ with their ids (ids of the encryption functions with the high weight):

```
high_weight_enc_ids =
[8685,14999,1087,10862,6168,12044,10258,1086,15000,6169,8683,1779,10861
,1778].
```

Summing up all of the described features of the CFG we can capture the complexity of it and hence the complexity of the function to be classified.

Total number of features of the Model 2 is equal to 15.

Split

Both pre-processed datasets were split into train and evaluation parts in the ratio to 8:2 using train_test_split method sklearn library with the same random state.
Training set size: 4858.
Test set size: 1215.

Classifier

Standard kernelized SVM with the linear kernel and regularization parameter equal to 1 installed from the sklearn library was employed in both cases. The multiclass support is handled here according to a one-vs-one scheme.

The experiments with the tuning of the hyperparameters were performed as well, but showed no visible increase in results, possibly because further improvements may be obtained by the extracting more peculiar properties of the data classes rather than variability of the model.

Evaluation

The quality of the predictions is represented by the means of confusion matrix (with the function drawing it in the graphical form) and classification report which displays values of the precision, recall, f1-score and accuracy metrics and from which is clearly seen that in terms of accuracy both models have satisfiable results (the final values of the test set accuracies are about 0.93 and 0.97 for Model 1 and Model 2 respectively), but the first one has a poor class recognition of the ‘sort’ class of functions (f1_score = 0.69, meanwhile f1-score of other classes is greater than 0.90):

```
           precision    recall  f1-score   support

  encryption       0.99      0.95      0.97       256
        math       0.98      0.98      0.98       456
        sort       0.80      0.61      0.69       112
      string       0.89      0.96      0.92       391

    accuracy                           0.93      1215
   macro avg       0.91      0.88      0.89      1215
weighted avg       0.93      0.93      0.93      1215
```

Applying the Model 2 significant improvements are noticeable:

```
             precision    recall  f1-score   support

  encryption       0.99      0.96      0.97       256
        math       0.96      0.99      0.98       456
        sort       0.90      0.93      0.92       112
      string       0.99      0.96      0.97       391

    accuracy                           0.97      1215
   macro avg       0.96      0.96      0.96      1215
weighted avg       0.97      0.97      0.97      1215
```

Reasoning about the source of the ‘sort functions’ problem will be provided right below in the conclusion section of the report.

Conclusion

Model 1 performs worse in the comparison with the Model 2, especially, referring to sorting functions recognition. The reason is that omitting the information about complexity of the function to be classified which can not be obtained without usage of the CFG entails the situation where ‘sort’ class has no features which could represent its specific characteristics compared to other classes. For example, mathematical functions can be distinguished with such features as the number of xmm* registers or the number of arithmetic operations provided by the Model 1 pretty well since a lot of xmm* registers and arithmetic operations are explicit distinctive features for this class of functions exactly. The same may be said about string manipulation functions and, for example, the number of comparisons or for the encryption functions and the numbers of the shift and bitwise instructions. Meanwhile, there does not exist any unique feature belonging to the class of the sort functions on the ‘privileged rights’. With the precise look on this class of functions the same as everywhere movement, comparison, call, jump operations will be found, with the lesser complexity, however. That is why obtaining the results of the Model 1 leads to the inferences that complexity of the control flow graph should be also captured.

In both cases models were fit fast, so the main issue in the sense of time is the procedure employed to count simple cycles in the CFG.