Homework 1

Silvia del Piano
1759992
delpiano.1759992@studenti.uniroma1.it

1 Assignment Description

The homework was given during a seminar about the possible applications of machine learning in cybersecurity. In particular, the main topic was detection of malwares: what is known as malware analysis. Unfortunately, it's a very difficult work: starting from binary files, the security analyst should be able to convert them to an Assembly file and understand if the software it's dangerous or not. This is done with the help of some tools, like virtual machines to execute the binary file in a protected environment, a "sandbox", to understand if it behaves maliciously.
However, in spite of the expertise of analysts and these automatic tools, it still remains a very challenging task, even considering the progress periodically made by adversaries. Some malwares, for example, can detect if they're being executed in a sandbox, or on a machine that is not their main target, and exhibit normal behaviour. This allows them to stay hidden until they find the right moment to strike.
A good way to understand if a software is malicious or not is to analyze its functions. After reconstructing the Assembly code, the analyst will try to understand what is the task that the functions that are present will perform: in their actions there could be found fingerprints of a malicious intent. For example, the usage of encryption functions and calls to Russian DNS servers are not a good sign.
In the last years, an effort has been made to use machine learning techniques to help in this field. Results have shown that while this type of solution is very useful to detect well known threats with really high accuracy, it's also weak to small changes. Sadly, this is a major drawback, as a single malware usually generates a number of small variances to be more difficult to detect, spawning a "family" from a single software.
Furthermore, many researches that have been done on these techniques suffer of bias, as the datasets are not really representative of contemporary threats.
In this homework, we were asked to provide a helping hand to this problem, by classifying functions. We were given a dataset containing functions that are to be classified in four categories: math, encryption, string and sort.

2 Dataset description

The dataset we were given to train our model of choice has 14397 instances, each for a function:
- 2724 are encryption functions
- 3104 are string manipulation functions
- 4064 are sorting functions
- 4504 are math functions

The file is in JSON format, and each object has the following form:

```
{"id": "828",
  "semantic": "string",
  "lista_asm": "['jmp qword ptr [rip + 0x220882]', '
     jmp qword ptr [rip + 0x220832]', ..., 'ret ']",
  "cfg":
    {"directed": true,
      "graph": [],
      "nodes": [
        {"id": 4276480,
          "asm": "554889e54883ec2048897df048837df0000f
             841d000000",
          "label": "0x414100:\tpush\trbp\n0x414101:\
             tmov\trbp, rsp\n0x41410\tsub\trsp, 0x20\n
             0x414108:\tmov\tqword ptr [rbp - 0x10],
             rdi\n0x41410c:\tcmp\tqword ptr [rbp - 0x1
             0], 0\n0x414111:\tje\t0x414134"},

        ... ,
        {"id": 4276714,
          "asm": "8b45e0488b4de889018b45e4488b4de88941
             04837de4000f8419000000",
          "label": "0x4141ea:\tmov\teax, dword ptr [
             rbp - 0x20]\n0x4141ed :\tmov\trcx, qword
             ptr [rbp - 0x18]\n0x4141f1:\tmov\tdword
             ptr [rcx], eax\n0x4141f3:\tmov\teax,
             dwordptr [rbp - 0x1c]\n0x4141f6:\tmov\
             trcx, qword ptr [rbp - 0x18]\n0x4141fa:\
             tmov\tdword ptr [rcx + 4], eax\n0x4141fd:
             \tcmp\tdwordptr [rbp - 0x1c], 0\n0x414201
             :\tje\t0x414220"}
      ],
      "adjacency": [[{"id": 4276532}, ...,{"id": 42767
         43}]],
      "multigraph": false
    }
}
```

Each object is build like a dictionary, and represents a function, where:
- ID: unique id of each function
- semantics: the label of each function, one of the main four classes
- lista_asm: the linear list of assembly instruction of each function
- cfg: the control flow graph encoded as a networkx graph

Fortunately, we don't need to directly analyze binary files, but we're given the function as a set of Assembly instructions and as a graph. This is useful for malware analysis because, as stated before, if the analyst can understand which type of functions are in a software, he can have a clearer understanding of the danger the program could pose (encryption functions are usually a bad sign). We know that the type of a function can be identified, among other things, by

the instructions that are used in it:
- encryption: a lot of xor, shift and bitwise operations; contain a lot of instructions
- sorting: compare and moves operations
- math: a lot of arithmetic operations
- string manipulation: a lot of comparisons and swap of memories

3 Approach to the problem and data preprocessing

Given all that is stated above, we can try to search for features using the list of Assembly instructions. Taking inspiration from the exercises and the lectures explained in class, we can adopt the following solution.
We preprocess the data set using the bag-of-words technique, which consists of creating a vector, where each cell corresponds to a word. For every word, we count its occurrences in the function we want to classify in the list of Assembly instructions. To obtain this vector, we first need to extract the lista_asm field. Each lista_asm has this form: "['jmp qword ptr [rip + 0x220882]', 'jmp qword ptr [rip + 0x220832]', 'jmp qword ptr [rip + 0x220822]', ...]".
First of all, we need to delete the first and last square brackets (demonstration with Python shell):

```
>>> word = "['jmp qword ptr [rip + 0x220882]', 'jmp
   qword ptr [rip + 0x220832]', 'jmp qwordptr [rip + 0
   x220822]"
>>> word[1:-1] \\
"'jmp qword ptr [rip + 0x220882]', 'jmp qword ptr [rip
    + 0x220832]', 'jmp qword ptr [rip + 0x220822"]
```

Now, we need to split the string into the different instructions:

```
>>>word2 = word[1:-1].split("', ")\\
["'jmp qword ptr [rip + 0x220882]", "'jmp qword ptr [
   rip + 0x220832]", "'jmp qword ptr [rip + 0x220822"]
```

Finally, for every instruction we want to take only the Assembly method:

```
>>> for e in word2: e.split(" ", 1)[0]\\
...\\
"'jmp"\\
"'jmp"\\
"'jmp"\\
```

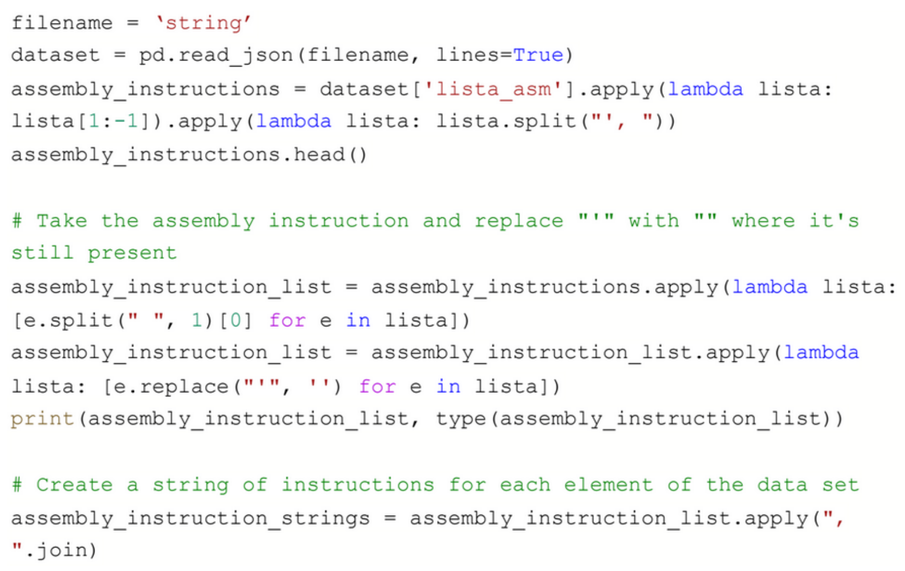
where split(" ", 1) means that we split the string where we find a blank space, in this form: [first word, rest of words in element]. If it was split(" ", 2), the split would produce this output: [first word, second word, rest of the words in the element].

```
>>> word2[0].split(" ", 1)\\
["’jmp", ’qword ptr [rip + 0x220882]’]\\
>>> word2[0].split(" ", 2)\\
["’jmp", ’qword’, ’ptr [rip + 0x220882]’]\\
```

Finally, we need to delete all the apices before the single method:

```
>>> for e in word2: e.split(" ")[0].replace("’", ’’)\\
...\\
’jmp’\\
’jmp’\\
’jmp’\\
```

In the end, a string for each instance will be created by joining the single instructions together. We need to be mindful of the fact that when we read the JSON data set and try to parse it, the objects returned belong to pandas.DataFrame. To make sure we process properly each instance, we use the apply function, which allows us to use a lambda function to process each of them. Therefore, the following code is the result:



```
filename = ‘string’
dataset = pd.read_json(filename, lines=True)
assembly_instructions = dataset['lista_asm'].apply(lambda lista:
lista[1:-1]).apply(lambda lista: lista.split("', "))
assembly_instructions.head()

# Take the assembly instruction and replace "'" with "" where it's
still present
assembly_instruction_list = assembly_instructions.apply(lambda lista:
[e.split(" ", 1)[0] for e in lista])
assembly_instruction_list = assembly_instruction_list.apply(lambda
lista: [e.replace("'", '') for e in lista])
print(assembly_instruction_list, type(assembly_instruction_list))

# Create a string of instructions for each element of the data set
assembly_instruction_strings = assembly_instruction_list.apply(",
".join)
```

To build the bag-of-words, we use the vectorizer functionalities provided by sklearn library:

```
# VECTORIZER (bag-of-words method)
vectorizer = CountVectorizer()
bag_of_words_vector =
vectorizer.fit_transform(assembly_instruction_strings)
```

We also need to split the data set into a set for training the model and one to test how well it performs. Following the empiric methods we studied, 2/3 of the data is dedicated to training, while 1/3 is for testing.

```
X_train, X_test, y_train, y_test =
train_test_split(bag_of_words_vector, Y_labels, test_size=0.3,
random_state=0)

print("Size of training set: %d" %X_train.shape[0])
```

```
print("Size of test set: %d" %X_test.shape[0])
```

We also extract the class names from the data to visualize correctly the results in a confusion matrix:

```
# For evaluation
class_names = np.array([str(c) for c in dataset['semantic']])
class_names = np.unique(class_names)
print("Class names: ", class_names)
```

4 Classifiers considered

For this assignment, the following classifiers were considered:

```
# MULTINOMIAL
#classifier = MultinomialNB()
#print("Multinomial Naive Bayes classifier created")

# SVM
#kernel_name  = 'poly' # 'linear', 'poly', 'rbf', 'sigmoid'
#classifier = svm.SVC(C=1.0, kernel=kernel_name, degree=3,
gamma='scale')
#print("SVM " + kernel_name + " classifier created")

# DECISION TREE
classifier = tree.DecisionTreeClassifier(random_state=0)
print("Decision tree classifier created")

# RANDOM FOREST
#classifier = ensemble.RandomForestClassifier(n_estimators=100)
#print("Random forest classifier created")
```

Let's have a brief look at each of them.

4.1 Multinomial Naive Bayes

It's a probabilistic classifier, based on the use of Bayes' theorem, with the assumption that all the features are independent among themselves, and equally distributed random variables. Therefore, we can compute their joint probability just by multiplying them.

4.2 Support Vector Machine with kernel method

This model represents the data instances as points in space, building hyperplanes, positioned such that if the points belong to different classes, they are separated by the greatest possible distance. They can be used for linear classification, but also for non-linear one with the help of kernel methods. These methods transform the input data, translating them from a space to another one, where the separation between the classes could be better. We considered a linear kernel, a polynomial one, a sigmoid one, and a radial basis function.

4.3 Decision Tree

These are trees used for classification. The leaves are the labels of the different classes, and the branches represent conjunctions of features. Its peculiarity is that it can effectively represent the decision process the machine makes to classify an instance into a class. This is an issue in machine learning, as other methods don't represent the decision process, and for a human are a black box that receives an input and produces an output.

4.4 Random Forest

It's a forest built by different decision trees. They are often used to correct overfitting, which could easily be present in decision trees. Therefore, they usually outperform decision trees with respect to new data, even if they have a lower accuracy.

After a classifier is created, the model is fitted and trained:

```
model = classifier.fit(X_train, y_train)
print("Model fitted")
y_pred = model.predict(X_test)
print("Model prections done")
```

5 Results

Let's now have a look at the parameters that are used to understand if the model is classifying instances correctly. First of all, we need to understand that when we classify data there are four categories in which the instance of our classification can belong: true positives and negatives, and false positive and negatives.

Precision: among all the classes taken into consideration, how many of them are classified correctly (true positives)

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives}$$

Recall: among all the positives, how many of them are classified correctly (true positives)

$$Recall = \frac{TruePositives}{TruePositives + FalseNegatives}$$

Accuracy: among all the predictions, which are correct

$$Accuracy = \frac{TruePositives + TrueNegatives}{TruePositives + FalsePositives + TrueNegatives + FalseNegatives}$$

To better visualize the results given by the different classifiers, the confusion matrix and a classification report has been printed, however, due to space reasons in the report, they are represented by the following tables.

Decision Tree

	Precision	Recall	F1-score
Encryption	1.00	0.99	0.99
Math	1.00	1.00	1.00
Sort	1.00	1.00	1.00
String	0.99	1.00	1.00

Accuracy: 1.00

Random Forest

	Precision	Recall	F1-score
Encryption	1.00	0.99	1.00
Math	1.00	1.00	1.00
Sort	1.00	1.00	1.00
String	1.00	1.00	1.00

Accuracy: 1.00

Multinomial Naive Bayes

	Precision	Recall	F1-score
Encryption	1.00	0.81	0.89
Math	0.97	0.90	0.94
Sort	0.81	0.88	0.84
String	0.90	0.83	0.86

Accuracy: 0.88

Support Vector Machine (polynomial kernel of degree 3, C = 1.0

	Precision	Recall	F1-score
Encryption	0.99	0.64	0.78
Math	0.38	1.00	0.55
Sort	0.67	0.12	0.21
String	1.00	0.02	0.04

Accuracy: 0.47

5.1 Performance variance

It's worth doing some comments about the results. The classifier that performs better is the Multinomial Naive Bayes, which seems to have good enough results without a high risk of overfitting.
Support vector machines perform very poorly with a polynomial kernel method. Changing the regularization factor C seems to produce some improvements: with C = 1.5, accuracy is 0.48; with C = 2.0, accuracy is 0.50; with C = 3.0 accuracy is 0.51. Increasing the value of C seems to produce good results: finally, with C = 4.0, accuracy is 0.53. Now, resetting C to 1.0, let's see if the results improve by changing the degree. With degree 2, accuracy improves to 0.60. With degree 1 accuracy becomes 0.86.
Now let's consider the case of a linear kernel. Accuracy increases until 0.99. Instead, using a sigmoid kernel accuracy drops to 0.70 with C = 2.0 and degree 3. Assigning C = 1 produces an accuracy of 0.69, while increasing C to 3.0, accuracy is 0.71.
Finally, considering the radius basis function kernel, we have accuracy of 0.90, with C = 1.0 and degree 3. Increasing C to 2.0 increases the accuracy to 0.92. C = 3.0 produces accuracy of 0.93. Changing the degree to 2 instead, while leaving C = 1, produces an accuracy of 0.90. Increasing the degree to 4 doesn't help much improvement, since accuracy stays at 0.90.

A summary of this considerations can be found in the following table:

Kernel	C	Degree	Accuracy
Linear	1	-	0.99
Polynomial	1.0	3	0.47
Polynomial	1.5	3	0.48
Polynomial	2.0	3	0.50
Polynomial	3.0	3	0.51
Polynomial	4.0	3	0.53
Polynomial	1.0	2	0.60
Polynomial	1.0	1	0.86
Sigmoid	1.0	3	0.69
Sigmoid	2.0	3	0.70
Sigmoid	3.0	3	0.71
Radius	1.0	3	0.90
Radius	2.0	3	0.92
Radius	3.0	3	0.93
Radius	1.0	2	0.90
Radius	1.0	4	0.90

For what concerns the decision tree classifier, increasing the random_state parameter doesn't produce a substantial change in the results.
The same is observed while changing the number of trees in the random forest classifier.

Overall, the best classifier seems to be the Multinomial Naive Bayes, together with Support Vector Machines, with a linear kernel (although it may suffer from overfitting). A polynomial kernel of degree 1 also gives a good result.
The decision tree and random forest have an impressive accuracy, but they might be suffering from overfitting.

6 Final implementation notes

Finally, some notes on the implementation. All the code has been written with Google Colab, and the dataset with duplicates has been used to train and test the model, using the blindset with duplicates to make predictions.
The results written in the .txt file have been computed with the Multinomial Naive Bayes model.