Machine Learning - Homework 1

Classification Function

Gianluca Pulicati - 1708686

Introduzione

The first Machine Learning course's homework assigned to the students of Engineering Computer Science at "Sapienza Univerità di Roma" is a first useful approach to the tools and methodologies used to analyze and work with datasets in order to make any sort of prediction and evaluation.
The homework is based on the seminar of *"Machine learning & Security Research"* hold during lectures. In the seminar was discussed how is possible for malware developers to eventually make even a small change to evade security defenses and, on the other hand, how hard is to reverse real-world binaries to understand if it's a malware or a safe software. Thus the machine learning approach can be used to analyze even huge binaries files to extract keywords which are fundamental to classify a software as a malware or not, reducing time and effort needed for developers to, for example, build solutions and fix problems.

Objectives

The main object of the homework is to classify binary functions which belong to four different classes:

- Encryption
- String Manipulation
- Math
- Sorting

In order to accomplish this objective there are several characteristics which can help to distinguish one function type to another. One example, used for this homework, is to account the kind of instructions contained in each function.

Datasets

The datasets given for this homework consist in *"dataset.json"* and *"blind.json"*. The first data-set is used to pre-process data and train the model in order to be able to predict and classify the label of each binary function accordingly. The latter data-set is instead used to fully test the model built with the first data-set with "blind" data.
In general, this type of problems require a "well-formed" data-set with the most complete and accurate information possible, thus to have a good model able to work even with the "blindset" which gives low information or, even worse, a lot of noise.
Files given are "JSON" or, to be more precise, "JSON Lines" files; the difference lies on the fact that in JSON Lines format we have a JSON object on each line of the file which is useful for pre-process operations.

Each JSON object is a dictionary *("key1:value1,key2:value2,key3:value3,...)*
The keys are:

- ID: unique id of each function
- semantic: the label of each function in *math,sort,encryption,string*
- lista_asm: the linear list of assembly instructions of each function
- cfg: the control flow graph encoded as a networkx graph

ID is used to keep order between every dataset's sample. *Semantic and lista_asm* are the core of the data-set and where the supervised learning take place as described below. *Cfg* describes the flow of every function as a networkx format graph. Unfortunately this feature was not used (and implemented) due to a problem during the test and executions of some library function.

To help extracting core features from the data-set these tips are given:

1. **Encryption:** Complex functions, use a lot of xor, shifts and bitwise operations.
2. **Sorting:** Simple logic, one or two cycles, some compare functions.
3. **Math:** A lot of arithmetic operations.
4. **String:** A lot of comparisons and swap operations.

Tools and methodologies

To accomplish this homework different tools have been used (and imported):

Import necessary libraries

```
[2] import pandas as pd
    import numpy as np
    import matplotlib.pyplot as plt
    import time
    from sklearn.model_selection import train_test_split
    from sklearn import svm
    from sklearn import tree
    from sklearn.naive_bayes import BernoulliNB
    from sklearn.metrics import classification_report, confusion_matrix
    from sklearn.feature_extraction.text import *
```

The Python's code above shows all the libraries used for the homework. As shown, *"sklearn"* library is imported to solve the problem, which is a open-source library for machine learning in python, even for beginners.
panda: panda is the famous framework used to work, import, extract data from data-sets; it can be used for every kind of data analysis and manipulation.
Then from sklearn:
numpy: To handle arrays
pyplot: Necessary to print confusion_matrix
time: Used to measure training computational time of different models.
train_test_split: function to split the dataset into random train and test subsets.
svm: Support-vector machines model.
tree: Regression Tree model.
naive_bayes: Naive Bayes classifier.
metrics: tools to plot classification report and confusion matrix (no plot).
feature_extraction: library for vectorizers.
plot_confusion_matrix (not shown): Imported function to plot confusion matrix of trained models.

Note: The whole homework has been developed on "Google Colab", firstly to have a smooth "hardware" setup in order to test, run and debug the problem (which could be very slow on a dated computer) and also to add some layout while coding the solution.

Data-set file management

```
functions_dframe = pd.read_json("/content/drive/My Drive/dataset.json", lines =True).drop_duplicates(subset=['lista_asm'])
labels = ['encryption', 'math', 'sort', 'string']

### Print some information about dataset ###

print('Printing some infos ... \n')
class_names = functions_dframe.columns
print(class_names)

## In a clearer way... ##
#class_names_clear = np.array([str(c) for c in class_names])
#print(class_names_clear)

print('\nShape of the dataset:')
print(functions_dframe.shape)
```

As mentioned, the homework has been developed on Colab: with the first line the script imports the data-set file from the Google Drive folder after prompting a warning for permission; here the file is imported with *read_json* function with parameter *lines=True*, this implements the import from a JSON Line file. The function saves the data-set on a *pandaframe* from the relative library.
Further on some data-set's info are printed, such features (columns) and shape (Number of lines = data-set's samples, Number of columns = data-set's features). Other information (not shown in the code above) can be checked removing comments on lines (for example: print the first 5 samples, or samples [300:349]).

Note on drop_duplicates:
drop_duplicates function has been added some days later the start of the homework: course's professors communicated that the data-set given is full of duplicates, thus when building and making predictions with the model was possible to have as result really high accuracy (like all '1.00' for test_set in use); to solve this issue was possible either to try dropping duplicates (and firstly i tried to do so by cycling samples and applying a manual check, but was a heavy and not optimal solution), or use another data-set given which was free of duplicates
In the end, both data-sets were used to measure performance differences although this was minimal (both data-sets have a shape of 6073 samples).

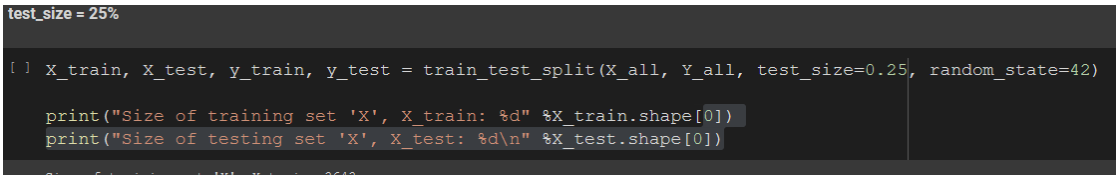
Vectorization

```
vectorizer = CountVectorizer(token_pattern="['][a-z]+\s")

X_all = vectorizer.fit_transform(functions_dframe['lista_asm'])
Y_all = functions_dframe['semantic']

## Print the function's names captured by the pattern ##
#print(vectorizer.get_feature_names())
```

In this piece of code the true pre-processing of data takes place.
In the first line *CountVectorizer* is initialized; it provides a simple way to both 'tokenize' a collection of text documents (or, as in this case, a text describing binary functions in assembly) and then build a vocabulary of selected words in order to learn from it how many times a specific *'string'* appears in a specific document and then classify it accordingly.
Then the vectorizer trasforms and fits the values of **'lista_asm'** of the dataframe imported from data-set file. At the end the result of this operation along with the values of **'semantic'** field (labels) are saved in two different variables used later.

Note on token_pattern:
Without any setting, CountVectorizer counts every string as a single token for the vocabulary. Before using and setting the token_pattern value, was tried a manually solution of isolating the necessary words from the lista_asm camp, indeed this solution was even working but after discovering the possibility of using regex value to 'capture' correct and useful worlds, this solution was removed.
Analyzing the data-set's lista_asm field, was decided to extract assembly commands called by every sample function (for example: 'call', 'move', 'add', 'pop', 'imul', ...); these commands follows a simple regex's pattern:
Before every command there is a "'" (**[']** in regex), then a sequence of chars follows (**[a-z]+** in regex, with the plus meaning at least one) and then a blank space follows as a "string terminator" (**/s** in regex).
This setting (and intuition) was very helpful to realize a clearer and faster solution.

Data-set Split



In this phase finally the learning part begins:
The pre-processed data is divided into four portions to first train the model and then to evaluate it in order to measure his accuracy and performances:

- **x_train:** The training set
- **y_train:** The set of labels to all the data in x_train
- **x_test:** Validation set
- **y_test:** The set of labels to all the data in y_test

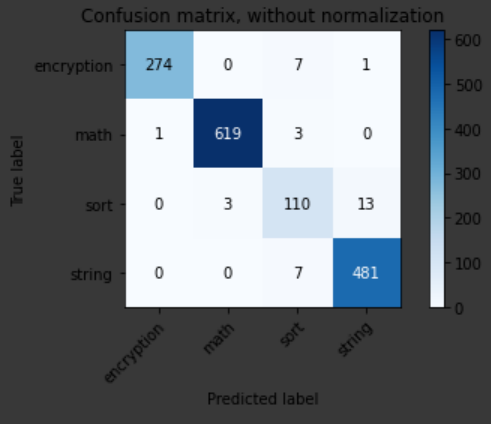
Also **X_all** is the vectorizer after the transformation and **Y_all** is the set of labels, both passed as parameter of the split function.

Note:
test_size and **random_state** are also very important parameters: the first describe the percentage of the data-set that will be used as validation set (and of course the rest will be the percentage of training set) while the latter is used as random seed (changing this value will effect which sample will be part of the training set or validation set.)

Decision Trees model

```
time_start = time.clock()
#model = svm.SVC(kernel='linear', C=1).fit(X_train, y_train)
model = tree.DecisionTreeClassifier().fit(X_train, y_train)
time_elapsed = (time.clock() - time_start)
print("Time for model 'fit' =  ", time_elapsed)
print("\n\n")

prediction = model.predict(X_test)

print(classification_report(y_test,prediction))
plot_confusion_matrix(y_test, prediction, classes=labels_array, normalize=False)
```

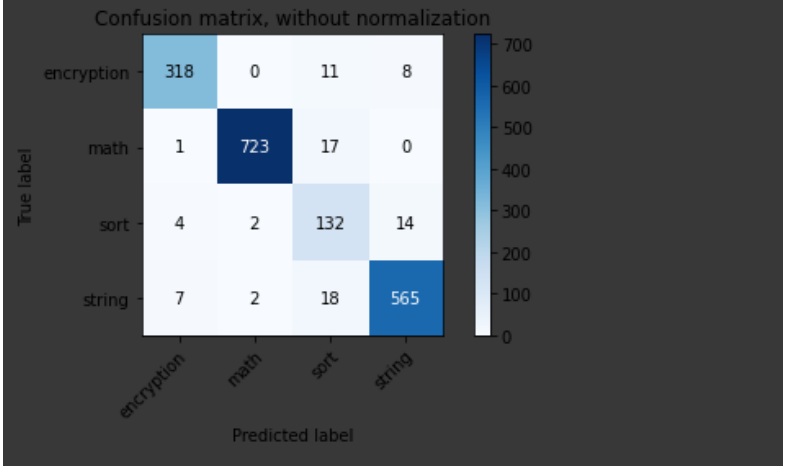
```
Computational training time =   0.0651239999999973


              precision    recall  f1-score   support

  encryption       1.00      0.97      0.98       282
        math       1.00      0.99      0.99       623
        sort       0.87      0.87      0.87       126
      string       0.97      0.99      0.98       488

    accuracy                           0.98      1519
   macro avg       0.96      0.96      0.96      1519
weighted avg       0.98      0.98      0.98      1519

<matplotlib.axes._subplots.AxesSubplot at 0x7fb0ea0b6c18>
```



Bernoulli model

```
time_start = time.clock()
model = BernoulliNB().fit(X_train, y_train)
time_elapsed = (time.clock() - time_start)
print("Computational training time =  ", time_elapsed)
print("\n")

prediction = model.predict(X_test)

print(classification_report(y_test,prediction))
plot_confusion_matrix(y_test, prediction, classes=labels_array, normalize=False)
```

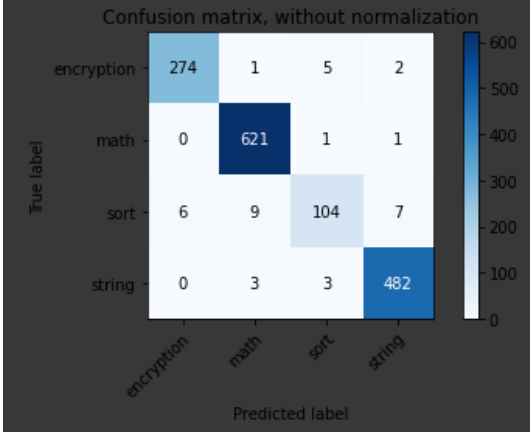
```
Computational training time =   0.028696000000000055


              precision    recall  f1-score   support

  encryption       0.96      0.94      0.95       337
        math       0.99      0.98      0.99       741
        sort       0.74      0.87      0.80       152
      string       0.96      0.95      0.96       592

    accuracy                           0.95      1822
   macro avg       0.92      0.94      0.92      1822
weighted avg       0.96      0.95      0.96      1822

<matplotlib.axes._subplots.AxesSubplot at 0x7fb0eac58c88>
```



Support Vector Machine - Linear

```
time_start = time.clock()
model = svm.SVC(kernel='linear', C=1).fit(X_train, y_train)
time_elapsed = (time.clock() - time_start)
print("Computational training time =  ", time_elapsed)
print("\n")

prediction = model.predict(X_test)

print(classification_report(y_test,prediction))
plot_confusion_matrix(y_test, prediction, classes=labels_array, normalize=False)



#model = BernoulliNB().fit(X_train, y_train)
#model = tree.DecisionTreeClassifier().fit(X_train, y_train)
#model = svm.SVC(kernel='linear', C=1).fit(X_train, y_train)
```

```
Computational training time =   0.5161650000000009


              precision    recall  f1-score   support

  encryption       0.98      0.97      0.98       282
        math       0.98      1.00      0.99       623
        sort       0.92      0.83      0.87       126
      string       0.98      0.99      0.98       488

    accuracy                           0.97      1519
   macro avg       0.96      0.95      0.95      1519
weighted avg       0.97      0.97      0.97      1519

<matplotlib.axes._subplots.AxesSubplot at 0x7fb0d8a944a8>
```



Conclusions

Note:
Every solution adopted for this homework was tested multiple times, both by changing test_size subset size and random state. Solutions displayed in above pages were computed with a **test_size = 0.30** and **random_state = 42** (if not mentioned otherwise).

As shown in the previous pages, three different approaches were chosen to model this homework and predict the solution. Firs of all, I tried to model the problem with Decision Trees: honestly i didn't trust much the "good" numbers resulted from the computation; the way the CountVectorizer has been implemented, in my opinion, is not a good way for Decision Trees. This because the solution simply counts the occurrences of every single assembly function called by the sample given; and analyzing the CountVectorizer's items is possible to notice that there are a lot's of functions which are very rare or called just few times: I think that this maybe created a problem of overfitted model, or in another words, the model builds over-complex tree and does not generalize the data well. A solution would be to prune the tree to at least remove features (assembly's function) which have low importance.
This could create other problems thought. As suggested from tips, the data-set has "Math" labeled samples that make huge use of math operations, but suppose that even another function type make use of some math operations mixed with other things: Will the tree simply assign the sample to Math just because finds a reasonable number of math functions? Maybe cause other features were pruned? The key is to find balance between a overfitted tree and a not enough "deep" one. Also maybe adding some other elements into the classification process would be useful (for example use of registers and special registers, like "xmm*").

On similar line of Decision Trees, just slightly better, there is Naive Bayes solution, and, to be more precise, BernoulliNB which brings some probability on evaluating the model.
I simply thought that instead of declaring a sample of this or that type basing the prediction only on the occurrences numbers of some functions was not enough to generalize the solution. So i tried to compute the model using only binary occurrence of functions (is this function in the sample?) using BernoulliNB (not multinomial). The data-set is presented with enough level of "separation" ("Math" uses math function, "Sorting" makes huge use of compare and mov*, ...), and so trying this approach seemed a good idea.
In my opinion the solution is still afflicted by some overfitting and as explained before this could be resolved by accounting also registers into CountVectorizers.

At the end, though, this approach helped me to realize that "Sort" and "String" samples make use of similar assembly's function, this is an important factor to remind when building a more solid train-set in pre-process phase (some sort/string samples where badly assigned, thus the accuracy decreased slightly).

Trying to bypass the "overfitted or not" dilemma which is the main protagonist of this homework, brought me to the idea of implementing Support Vector Machines, thus to have a model with some sort of regularization on parameters. The key here is to choose the right kernel and regularization parameter "C". I trusted the linearity of the data-set given (even on the way it was pre-processed) so i used Linear kernel with "standard" regularization (C=1).
Results seems very good with this approach, and indeed i felt much more confident that with Decision Trees and Naive Bayes.

As expected, the computational training time "winner" is the BernoulliNB Model, simply because it accounts much less parameters and tests to evaluate the model. The loser is of course Decision Tree model, and this slightly confirms the fact that the model is building a "over-complex" tree (overfitted, of course), it's even slower than SVM and his regularization process, I didn't expect it at all!