Machine Learning
Homework 1

Giuseppe Sensolini Arrà, 161198

November 2020

Introduction

The aim of this homework is to provide a solution for the function classification problem described in the seminar *Machine Learning and Security Research*.
This report is divided in three main sections. In the first one the feature extraction problem is introduced; methodologies and techniques used to spot meaningful features are investigated in this context.
Second section addresses the choice of the learning model and its implementation. Third section study in detail the performance of the learning model.

The language used to implement the proposed solution is `Python 3.8.5` with the `scikit-learn 0.23.2` library.

Feature Extraction

In this phase a preliminary analysis of the datasets has been conducted. In particular, two dataset has been used:

- `noduplicatedataset.json` is the dataset used to train and test the learning system. It is composed by 6073 entries and it does not contains duplicates.
- `blindtest.json` is used to perform a blind test, it contains 757 entries with duplicates.

Four classes of functions are present inside the datasets: string manipulation, math, encryption and sort. Each entry of the first set is composed by 4 fields:

- `id`: unique id identifying the function
- `semantic`: the class of the function (ground truth)
- `lista_asm`: list of assembly instructions
- `cfg`: control flow graph

In what follows only the `semantic` and the `lista_asm` will be used, while the `id` and the `cfg` fields will be discarded; clearly, the control flow graph could potentially provide others useful features, but as it will be shown later on, the assembly intructions are alredy sufficient in order to achieve satisfactory performances. Both sets are read in python, and meaningful features are extracted and stored respectively into `fncts` and `blind_fncts` matrices.
The chosen features are the following:

- `instr`: number of assembly instructions appearing in a function
- `mov`: number of The `mov` operations
- `cmp`: number of texttcmp operations

- `arithmetic`: number of basic arithmetic operations (`sum, sub, mul, div`)
- `bitwise`: number of basic bitwise operations (`not, and, or, xor`)
- `call`: number of `call` operations
- `jump`: number of jump operations (`jmp, je, jz, jne, jnz, js, ...`)
- `shift`: number of shift operations (`shr, shl, sal, sar`)
- `float`: take into account the number of single/double precision operations (`movss, movsd, addss, addsd, ...`) and the occurrences of the xmm registers.

A preliminarly analysis is done in order to spot best features to include by using `features_averaging()`. This function returns the average features distribution over the four classes. A human observer can easily notice from *Table 1* that the `math` class has (on average) a large number of float operations if compared to the other classes, while `encryption` class contains a huge average number of instructions and bitwise operations. The same reasoning cannot be extended for the `string` and `sort` classes, since they share a similar features distribution.

class	instr	mov	cmp	arithm.	bitwise	call	jump	shift	float
string	106.07	29.37	13.79	6.22	5.17	3.05	17.36	1.18	0.58
math	57.98	10.48	4.24	3.46	1.42	3.79	12.84	0.18	75.45
encr.	595.94	34.7	2.72	10.53	34.06	2.8	3.4	5.72	0.74
sort	107.76	34.89	9.76	8.86	4.21	3.41	11.81	1.0	2.27

Table 1: Average feature distribution

This table has been used only for a better understending of the dataset. Feature extraction is made by the following snippet of code:

```python
data_set = "../dataset/noduplicatedataset.json"
blind_data = "../dataset/blindtest.json"

data, train_set, test_set, blind_set = [],[],[],[]

# open dataset
with open(data_set) as f1:
    for line in f1:
        data.append(json.loads(line))

# features extraction
fncts, labels, header, classes = fe.features_extraction(data)
```

```
13
14  # divide the dataset in training (80%) and testing (20%) sets
15  test_size = int(len(labels)*0.20)
16  training_size = len(labels)-test_size
17  train_fncts = fncts[0:len(fncts)-test_size]
18  train_labels = labels[0:len(fncts)-test_size]
19  test_fncts = fncts[len(fncts)-test_size:len(fncts)]
20  test_labels = labels[len(fncts)-test_size:len(fncts)]
21
22  # open blindset
23  with open(blind_data) as f2:
24      for line in f2:
25          blind_set.append(json.loads(line))
26
27  # blind features extraction
28  blind_fncts = fe.features_extraction(blind_set, blind=True)
```

`feature_extraction()` in lines 12 and 28 extracts the features respectively for the main dataset and the blindset. This function returns:

- `fncts`: a 6073x9 matrix, each row contain the feature values for each entry of the dataset
- `labels`: a 6073x1 array, each row contains the class (semantic) of the entry.
- `header`: an array containing the feature names
- `classes`: an array containing the class names

`feature_extraction()` basically performs a counting of the occurrences of each feature for each entry of the dataset.

```
def feature_extraction(data, blind=False):
    """Given a dataset exract the features of interest

    Args:
        data: dataset
        blind (bool, optional): if true the dataset is considered blind. Defaults to Fal

    Returns:
        fncts: nxm matrix (n = dataset_length, m = number_of_features)
        labels: ground truth, contains semantic for each entry of the dataset
        header: contains feature names
        classes: contains class names
    """

    print("extracting asm lists... ")
    for elem in data:
        elem["lista_asm"] = elem["lista_asm"].split("'")
        while ", " in elem["lista_asm"]: elem["lista_asm"].remove(", ")
        elem["lista_asm"] = elem["lista_asm"][1:len(elem["lista_asm"])-1]
    print("done.")

    # famility of assembly operations
    arithmetic_ops = ["add", "sub", "mul", "div"]
    bitwise_ops = ["not", "and", "or", "xor"]
    jump_ops = ["jmp","je","jz","jne","jnz","js","jg","jnle","jge","jnl","jl,jnge","jle"
    shift_ops = ["shr", "shl", "sal", "sar"]
    float_ops = ["ss","sd"]

    # count features
    for fnct in data:
        fnct["features"] = init_feature_dict()
        fnct["features"]["instr"] = len(fnct["lista_asm"])
        for instr in fnct["lista_asm"]:
            op = instr.split(" ")[0]
            if op=="mov": fnct["features"]["mov"]+=1
            elif op=="cmp" or op=="test":
                fnct["features"]["cmp"]+=1
            elif op in arithmetic_ops:
                fnct["features"]["arithmetic"]+=1
            elif op in bitwise_ops:
                fnct["features"]["bitwise"]+=1
                if op=="xor": fnct["features"]["bitwise"]+=1
            elif op=="call":
                fnct["features"]["call"]+=1
            elif op in jump_ops:
                fnct["features"]["jump"]+=1
            elif op in shift_ops:
                fnct["features"]["shift"]+=1
            elif op[len(op)-2:len(op)] in float_ops:
                fnct["features"]["float"]+=1
            fnct["features"]["float"] += 0.5*(instr.count("xmm"))

    # prepare data for the decision tree
    header = list(data[0]["features"].keys())
    classes = ["string","math","encryption","sort"]
    fncts, labels = [],[]
    if blind:
        for fnct in data:
            fncts.append( list(fnct["features"].values()) )
        return fncts
    else:
```

```python
        # A useful print to better understand classes' features
        features_averaging(data)
        for fnct in data:
            fncts.append( list(fnct["features"].values()) )
            if fnct["semantic"]==classes[0]: labels.append(0)
            elif fnct["semantic"]==classes[1]: labels.append(1)
            elif fnct["semantic"]==classes[2]: labels.append(2)
            elif fnct["semantic"]==classes[3]: labels.append(3)
        return fncts, labels, header, classes
```

Machine Learning Model

At this stage, given the outputs of `feature_extraction()`, it is possible to apply a learning method. The main dataset is divided in 80% for the training, corresponding to 4859 samples, and 20% for the testing, corresponding to 1214 samples.
The learning model used to solve the proposed classification problem is a **decision tree**; its implementation is done using the *scikit-learn* library. In particular:

```python
clf = tree.DecisionTreeClassifier(  criterion = "entropy",\
                                    splitter = "best",\
                                    max_features = None)
clf = clf.fit(train_fncts, train_labels)
```

Different **hyperparameters** can be tuned.
`criterion` defines the function to measure the quality of a split, `entropy` is the one used. Supported criteria are `gini` for the Gini impurity and `entropy` for the information gain. Information gain uses the entropy measure as the impurity measure and splits a node such that it gives the most amount of information gain. Whereas Gini Impurity measures the divergences between the probability distributions of the target attribute's values and splits a node such that it gives the least amount of impurity.

$$Gini(E) = 1 - \sum_{j=1}^{c} p_j^2 \tag{1}$$

$$H(E) = -\sum_{j=1}^{c} p_j \log p_j \tag{2}$$

`max_features` field is setted as default, i.e., the number of features to consider when looking for the best split. In the evaluation phase it will be shown how it

affects performances.
Other hyperparameters are `splitter`, i.e., the strategy used to choose the split at each node; setting this as `best` it evaluates all splits using a Fisher-Yates-based algorithm.
Hyperparameters `max_depth`, `min_samples_split`, `min_samples_leaf`, `min_weight_fraction_leaf` are left as default, meaning that no max/min for these parameters is setted. Also `random_state`, `min_impurity_decrease`, `class_weight` are left as default.

The obtained feature importance are summarized in *table 2*.

float	0.4026
bitwise	0.2951
cmp	0.0689
call	0.0522
mov	0.0506
jump	0.0478
arithmetic	0.0425
instr	0.0262
shift	0.0138

Table 2: Feature importance

Not suprisingly, the features that better separate the data are `float` and `bitwise`; this is also coherent with the observation previously made about *table 1*.

Evaluation

In this sections results are evaluated. It is possible to test the previously trained model using the test set. The ground truth is known, and therefore it is possible to estimate the accuracy of the model by comparing it with the results of the classification.
Classification results in terms of error and accuracy are summarized in *table 3*.

Error	0.0074
Accuracy	0.9925

Table 3: Error and accuracy

An important evaluation metric is the **F1 Score**; it results to be 0.9864.
The decision tree has been also tested with the **K-Fold cross validation** method. In this case it is achieved dividing the dataset in 8 subsets. with this testing method the accuracy drop down to around 97%.
It is worth to notice that the choice of the `criterion` plays a role with regards to accuracy; indeed, the usage of `gini` instead of `entropy` returns a slightly lower accuracy - around 0.50% less on average - and a (negligible) lower execution time (because equation (1) is less a computationally intensive operation than equation (2)).

Confusion matrix (*figure 1* and *figure 2*) is useful tool to spot how misclassified samples are distributed a across classes.

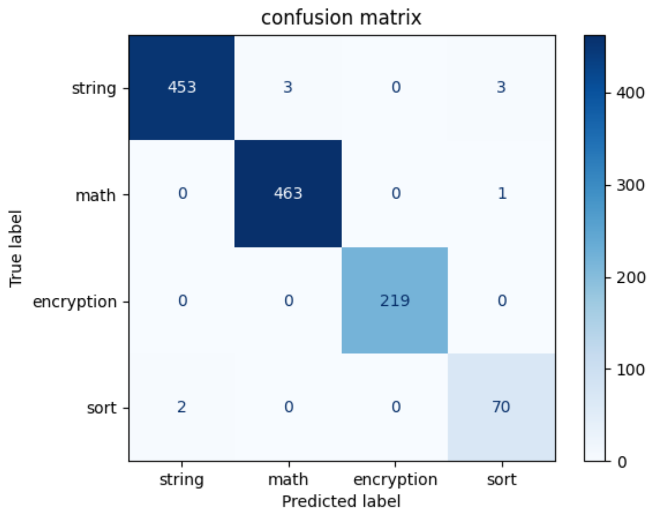

Figure 1: confusion matrix

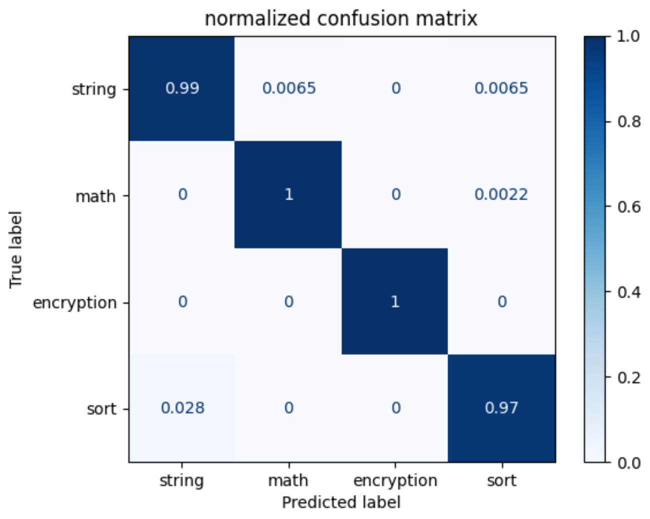

Figure 2: normalized confusion matrix

Comparison using different values of `max_features`. An hyperparameter of particular interest is `max_features`. Different tests have been performed using different values between 3 and 9. *figure 3* and *table 4* put in light how accuracy and running time are affected by this hyperparameter.

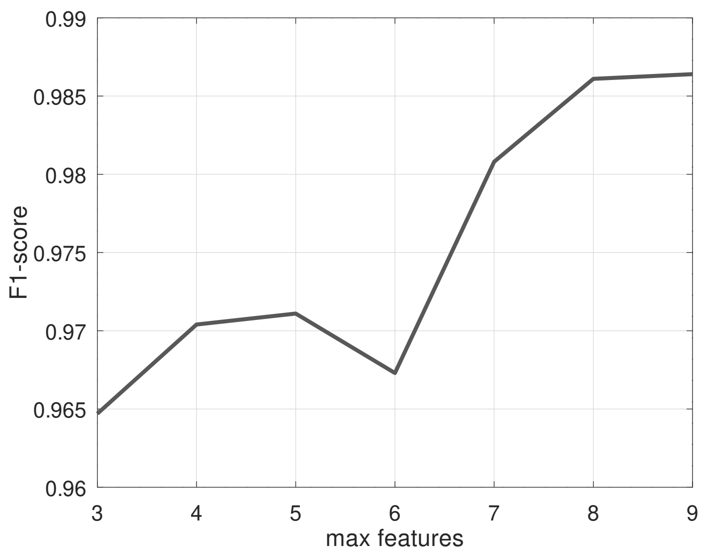

Figure 3: F1-scores for different values of `max_features`

	9 features	4 features
feature extraction	7.5228 s	7.4621 s
model training	0.0138 s	0.009 s
model testing	0.5074 s	0.4694 s
total time	8.044 s	7.9405 s

Table 4: Running time

Not surprisingly, the F1-score improves almost linearly with `max_features`. Execution time is not eccessively affected by this hyperparameter; this is coherent with the fact that most of the time is spent in the feature extraction phase; indeed, decreasing `max_features` speed up only the model training phase that becomes more than ten times faster in the 4-features-case with respect to the 9-features-case.

Conclusion

A comparison by means of different hyperparameters has been conducted, and the best configuration of hyperparameters resulted to be the one obtained setting `criterion=entropy` and `max_features=9`. It is reasonable to expect a greater benefits in increasing the `max_features` in the presence of more significant features (for example, by integrating also features extracted from the control flow graph).
At the end, the proposed model is able to classify the test set with an accuracy of around 99%, leading to quite satisfactory performance.