Machine Learning for Malware Analysis

Engineering on Computer Science

Daniele Cantisani 1707633

November 2019

Contents

1 Introduction

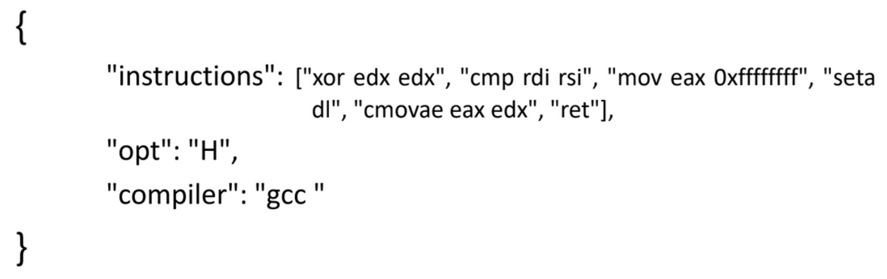
This report is written about the homework of the Machine Learning course. The final objective is that given the Assembly code of functions, we must be able to recognize the optimization and the compiler who produced them.

2 Using Machine Learning for Malware Analysis

The growth of technological development has brought numerous discoveries and possibilities in the computer technology. However, the use of increasingly sophisticated and useful devices in everyday life has introduced a very important topic: security. Despite being a very large topic, we want to focus on a fundamental problem such as the recognition of malicious software: malware analysis. Malware is a set of coded instructions that allows a person (attacker) to succeed in malicious intent in a specific task.
The analysis of this malicious software can be done in two different ways: In the static analysis way the malicious file is analyzed and using a disassembler the analyst is able to look at the binary code to understand what's going on. Otherwise there is the the dynamic analysis where the malware is executed in a controlled environment and its action on the system are registered and analyzed.
There are several techniques that an analyst can use to recognize malware such as identifying portions of similar code, using certain special instructions, etc. However malware can defend itself against these techniques by obfuscation. In this environment the role of Machine Learning for Malware analysis is fundamental. Thanks to a large data set we can be able to recognize similar features in different malware and therefore we may be able to recognize any bad software even if it implement obfuscation. In binary analysis there are different problems where Machine Learning can be useful like binary similarity, compiler provenance or function naming. For the objective of the homework we are interested to focus on the problem of compiler provenance. The compiler provenance is composed by two classical classification problems; indeed, we must be able to recognize the optimization and compilation used from binary instructions. For the classification of optimization we have a binary classifier while for the compiler classification we have a multi-class classification problem.

3 Presentation of Data Set

To train our system to solve the problem of the compiler provenance correctly, we have a large data set consisting of thirty thousand elements. Each element is composed of a series of instructions in assembly separated by a space, the optimization (High or Low) and the compiler (gcc, icc, clang) used.



```
{
        "instructions": ["xor edx edx", "cmp rdi rsi", "mov eax 0xffffffff", "seta
                        dl", "cmovae eax edx", "ret"],
        "opt": "H",
        "compiler": "gcc "
}
```

Figure 1: Example of an element in data set

In addition to the 30,000 data set elements, there is another file composed of 3000 elements; this is the test set called blind set. This file is composed only by assembly functions to which I will have to apply the best model to try to predict the optimization and the compiler that produced them.

4 Models

4.1 Naive Bayes Classifier

To build two of the models, I apply a Naive Bayes learning approach in respect to the data set. This approach takes name from the Bayes Theorem:

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

- $P(h)$ = prior probability of hypothesis h
- $P(D)$ = prior probability of training data D
- $P(h|D)$ = probability of h given D
- $P(D|h)$ = probability of D given h

From the probability of this theorem, it can be possible to build a classifier by analyzing the training-set.
The follow algorithm is the one that reads the training-set and calculates the

probability of each instance, given a target value v.

Algorithm 1 Naive Bayes Algorithm

```
1: procedure NAIVE_BAYES_LEARN($A$, $V$, $D$)
2:     for all target value $v_j \in V$ do
3:         $P(v_j|D) \leftarrow estimateP(v_j|D)$
4:         for all attribute $A_k$ do
5:             for all attribute value $a_i \in A$ do
6:                 $P(a_i|v_j, D) \leftarrow estimateP(a_i|v_j, D)$
7:             end for
8:         end for
9:     end for
10: end procedure
```

After calculating all these probabilities, they can be used in the real classifier, which now can make prediction about every instance x by compare and taking the highest probability among the ones belonging to the instance x.
This is the basic algorithm I follow to implement the predictions. Of course, it has to be adapted to the specific case. In particular, what i have done is to implement a Naive Bayes classifier with Multinomial and Bernoulli distribution. The Multinomial distribution generalize the Binomial distribution: in distribution, random variables could provide only two possible outcomes. In Multinomial, instead, there are at most n possible outcomes, with n a finite number. This distribution is better because it has improved the algorithm's performances, in particular some evaluation metric as efficiency and precision. Instead, the Bernoulli distribution is a special case of the Binomial distribution where a single experiment is conducted so that the number of observation is 1. So, the Bernoulli distribution therefore describes events having exactly two outcomes.

4.2 Decision Tree Classifier

A Decision Tree is a simple representation for classifying examples. It is a Supervised Machine Learning where the data is continuously split according to a certain parameter. A set of training examples is broken down into smaller and smaller subsets while at the same time an associated decision tree get incrementally developed. At the end of the learning process, a decision tree covering the training set is returned. Decision trees are built using a heuristic called recursive partitioning. This approach is also commonly known as divide and conquer because it splits the data into subsets, which are then split repeatedly into even smaller subsets, and so on and so forth until the process stops when the algorithm determines the data within the subsets are sufficiently homogenous, or another stopping criterion has been met.

5 Experience

To start working it was necessary to install Pip. Pip is a repository that contains tens of thousands of packages written in Python. The first thing to do is import all the necessary modules. Some examples are Numpy that it is a very powerful library that can accelerate multidimensional array operations, Pandas, a library based on NumPy which provides additional tools for high-level manipulation of data, which simplify the processing of type data even more Tabular. Others are Matplotlib that allows a graphical display of quantitative data and Scikit-learn. I will use mainly to the scikit-learn library, which, currently, is one of the most popular open source machine learning libraries. After the importation of these fundamental libraries, the first phase began. In this phase I had to read the instructions and the corresponding results of the data set and insert them in special structures that would have allowed me to create training models. As you can see from the code, thanks to a simple reading command provided by the "pandas" library, I could read from the jsonl file.

```
#READING JSONL FILE
datasetone = pd.read_json('train_dataset.jsonl', lines=True)
datasetone['instructions'] = datasetone['instructions'].apply(','.
    join)

datasettwo = pd.read_json('test_dataset_blind.jsonl', lines= True)
datasettwo['instructions'] = datasettwo['instructions'].apply(','.
    join)

#CREATE TARGETS
targetBinary=datasetone['opt'].values
targetMulticlass=datasetone['compiler'].values
```

At this point it was time to create a particular structure: The function CountVectorizer() (it belongs to the Sklearn library) is applied which converts a collection of text documents to a matrix of token counts. Individual tokens are taken individually.

```
#CREATE VECTORS
vectorizer = skf.text.CountVectorizer()

vector = vectorizer.fit_transform(datasetone['instructions'])

print(vector)



#CHOICES ON VECTORIZER AND SPLIT THE DATA IN TRAIN/TEST DATASET
typebm = input("Binary or MultiClass ? (1 for Binary and 2 for
    Multiclass): ")
X_all = vector
if typebm == "1":
    y_all = targetBinary
else:
    y_all = targetMulticlass
```

Now I can assign the structures we collected to the training variables. In the example I put a test-size of 0.2 which means I will use 80% of the data set as a training base and the rest it will be used as testing to calculate the accuracy of the model. At this point I create the three training models: Bernoulli, Multinomial and DecisionTree. The instructions with the corresponding result are passed as argument of these learning functions. Finally I execute the predict() passing as argument the previously chosen set to use as test set. Not shown in the code below are the input instructions where I choose whether to address the binary optimization problem or the multiclass problem for the compiler.

```
X_train, X_test, y_train, y_test = train_test_split(X_all, y_all,
    test_size=0.2, random_state=15)


#CREATE MODELS

modelBernoulli = BernoulliNB().fit(X_train, y_train)
modelMultinomial = MultinomialNB().fit(X_train, y_train)
modelDecisionTree = DecisionTreeClassifier(random_state=0).fit(
    X_train, y_train)


#TESTING THE MODELS
y_predBernoulli = modelBernoulli.predict(X_test)
y_predMultinomial = modelMultinomial.predict(X_test)
y_predDecisionTree = modelDecisionTree.predict(X_test)
```

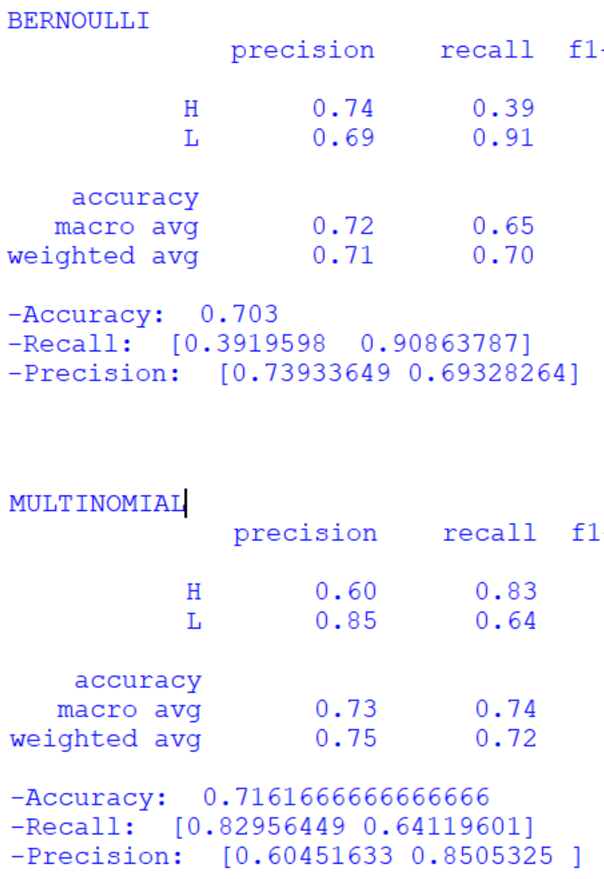
Finally we can observe the result obtained through the classification-report() function and print the confusion matrix and the ROC curves which allows us to analyze the result obtained.

```
print("BERNOULLI")
print(classification_report(y_test, y_predBernoulli))
print("-Accuracy: ", accuracy_score(y_test, y_predBernoulli))
print("-Recall: ", recall_score(y_test, y_predBernoulli, average=
    None))
print("-Precision: ", precision_score(y_test, y_predBernoulli,
    average= None))
```

6 Results

6.1 Binary Problem

After creating models with different algorithms and predicting a previously defined test size, we can analyze the results. The classification-report() function (parameters has both the solutions and the predictions), prints to us the values of Accuracy, Precision, Recall and F1-Score for each element to be predicted.
Accuracy is the most intuitive performance measure and it is a ratio of correctly predicted observation to the total observations. Precision is the ratio of correctly predicted positive observations to the total predicted positive observations. Recall is the ratio of correctly predicted positive observations to the all observations in actual class. Finally F1 Score is the weighted average of Precision and Recall. Therefore, this score takes both false positives and false negatives into consideration.



```
BERNOULLI
              precision    recall  f1-score   support

           H       0.74      0.39      0.51      2388
           L       0.69      0.91      0.79      3612

    accuracy                           0.70      6000
   macro avg       0.72      0.65      0.65      6000
weighted avg       0.71      0.70      0.68      6000

-Accuracy:  0.703
-Recall:  [0.3919598  0.90863787]
-Precision:  [0.73933649 0.69328264]



MULTINOMIAL
              precision    recall  f1-score   support

           H       0.60      0.83      0.70      2388
           L       0.85      0.64      0.73      3612

    accuracy                           0.72      6000
   macro avg       0.73      0.74      0.72      6000
weighted avg       0.75      0.72      0.72      6000

-Accuracy:  0.7161666666666666
-Recall:  [0.82956449 0.64119601]
-Precision:  [0.60451633 0.8505325 ]
```

Figure 2: Bernoulli and Multinomial results

As we can see the two methods that implement the Gauss theory have sufficient efficiency but do not reach very high values. Furthermore there is a big difference between the values of Recall and Precision (especially in Bernoulli) which shows that it is not a good implementation for this type of problem. Instead facing the problem through the Decision Tree Classifier it is possible to arrive at an excellent Accuracy of 0.82. So the best approach for this problem is the Decision Tree which is able to better classify the proposed data.

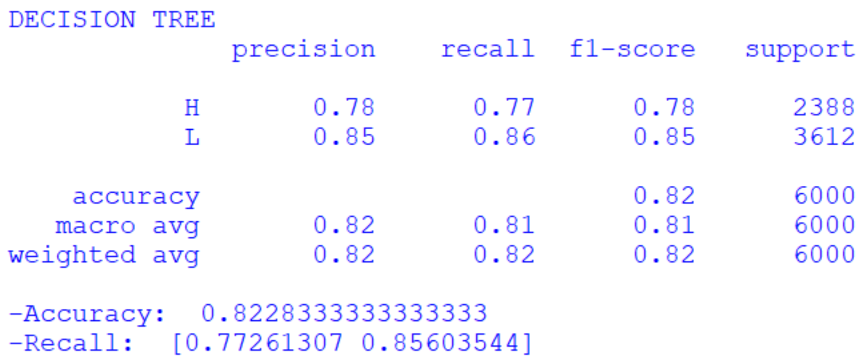
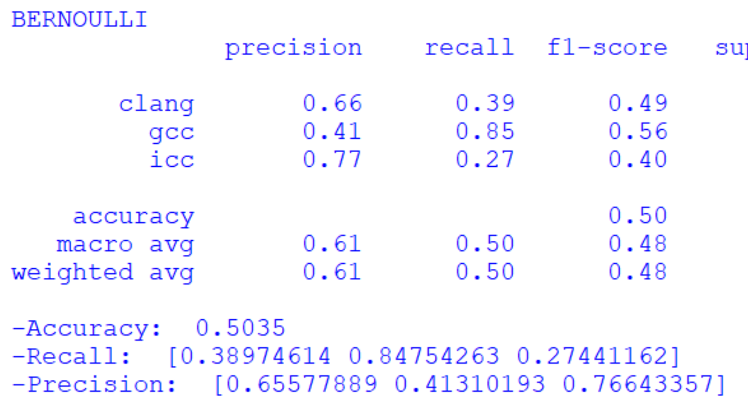

```
DECISION TREE
              precision    recall  f1-score   support

           H       0.78      0.77      0.78      2388
           L       0.85      0.86      0.85      3612

    accuracy                           0.82      6000
   macro avg       0.82      0.81      0.81      6000
weighted avg       0.82      0.82      0.82      6000

-Accuracy:  0.8228333333333333
-Recall:  [0.77261307 0.85603544]
```

Figure 3: Decision Tree results

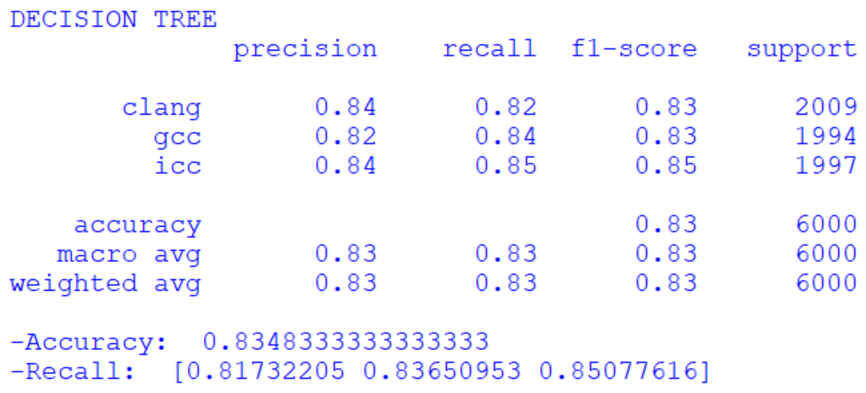
6.2 Multiclass Problem

Turning to the multiclass problem we notice how the probabilistic implementation completely collapses. Instead the Decision Tree algorithm has an excellent Accuracy of 0.83 and has excellent Precision, Recall and F1 Score values all very similar to each other.



```
BERNOULLI
              precision    recall  f1-score   support

       clang       0.66      0.39      0.49      2009
         gcc       0.41      0.85      0.56      1994
         icc       0.77      0.27      0.40      1997

    accuracy                           0.50      6000
   macro avg       0.61      0.50      0.48      6000
weighted avg       0.61      0.50      0.48      6000

-Accuracy:  0.5035
-Recall:  [0.38974614 0.84754263 0.27441162]
-Precision:  [0.65577889 0.41310193 0.76643357]
```

Figure 4: Classification Multiclass Bernoulli Model



```
DECISION TREE
              precision    recall  f1-score   support

       clang       0.84      0.82      0.83      2009
         gcc       0.82      0.84      0.83      1994
         icc       0.84      0.85      0.85      1997

    accuracy                           0.83      6000
   macro avg       0.83      0.83      0.83      6000
weighted avg       0.83      0.83      0.83      6000

-Accuracy:  0.8348333333333333
-Recall:  [0.81732205 0.83650953 0.85077616]
```

Figure 5: Classification Multiclass DecisionTree Model

Two other important measurements to see if a model works correctly are the confusion matrix and the ROC curves.
A confusion matrix is a table that is used to describe the performance of a classification model (or "classifier") on a set of test data for which the true values are known. The confusion matrix shows the ways in which our classification model is confused when it makes predictions. It gives us insight not only into the errors being made by a classifier but more importantly the types of errors that are being made. The confusion matrix is an excellent method to confirm the values of Precision, Recall and F1- Score. We can read the 3x3 confusion matrix (see below) in this way. The columns are the predictions and the rows must therefore be the actual values. The main diagonal (1642, 1668, 1699) gives the correct predictions. That is, the cases where the actual values and the model predictions are the same. So, for example, of the 1966 gcc compiler predicted by the model (sum of column gcc), 1642 were actually gcc, while 172 were clang incorrectly predicted to be gcc and 152 were icc incorrectly predicted to be gcc.

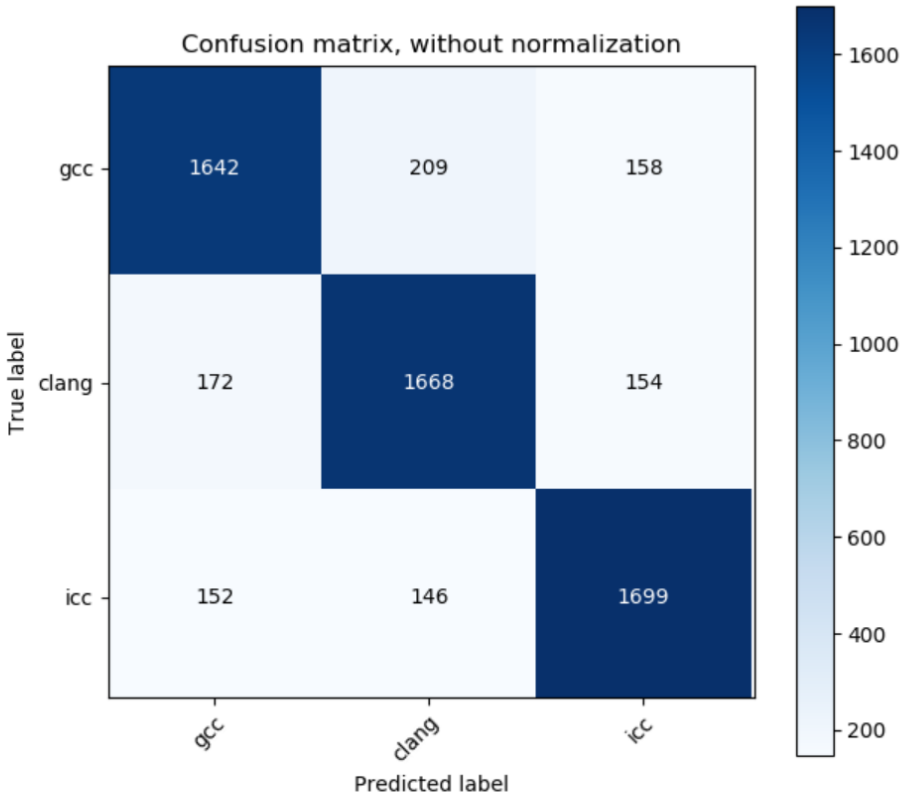


Figure 6: Confusion Matrix of Decision Tree Multiclass

ROC (Receiver Operator Characteristic) curves are useful tools for the selection of classification models based on their performance with respect to false positive and true positive rates. The diagonal of a ROC graph can be interpreted as the random indication. The classification models that fall under the diagonal are considered worse than the random choice. A perfect classifier is placed at the upper left corner of the graph, with a true positive rate of 1 and a false positive rate of 0.
We can see in the graph of the example how the precision is respected and confirmed by the ROC curves.

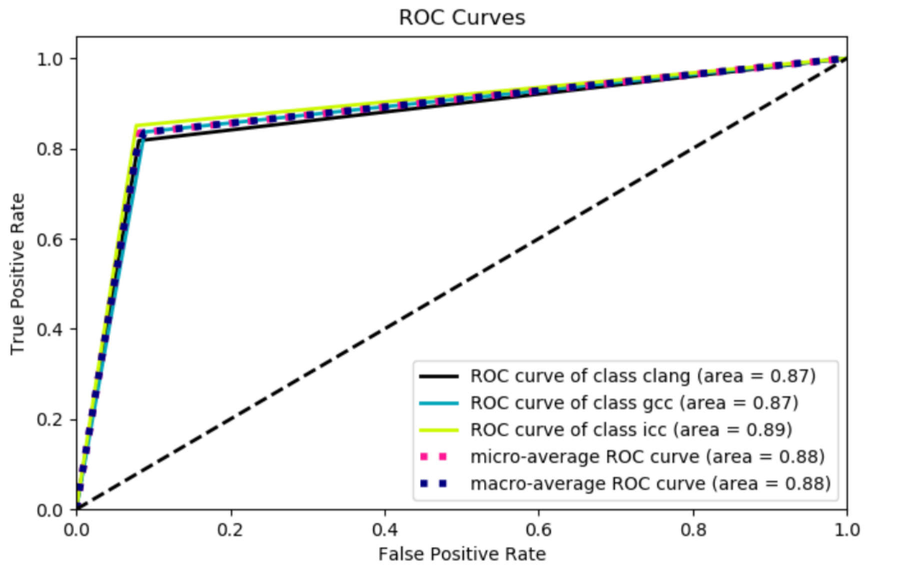


Figure 7: ROC curves of Decision Tree Multiclass

7 Last Task

After numerous tests carried out modifying the test size and the random state it was noticed how the result does not change much so it was decided to leave a test size of 0.2 and a random state of 0. Finally the last task is to build the csv file with the blind set predictions. It is required to use the best model, so in this case the Decision Tree Classifier will be used both for the binary and the multiclass problems.

```python
def exerciseBinary(dataset_train, test_blind):
    vectoresempio = skf.text.CountVectorizer()
    vectorTrainBlind = vectoresempio.fit_transform(dataset_train['
        instructions'])
    vectorBlind = vectoresempio.transform(test_blind)
    clf = DecisionTreeClassifier(random_state = 0).fit(
        vectorTrainBlind,dataset_train['opt'])
    y_predDecisionTree = clf.predict(vectorBlind)
    print(y_predDecisionTree)
    return y_predDecisionTree
```

After training using the Decision Tree algorithm and predicting the 3000 element data set, I insert the results in the form ¡compiler¿, ¡opt¿ in a csv file.

References

- Machine Learning for CyberSecurity...not only malware detection, Dr. Luca Massarelli
- https://scikit-learn.org/
- https://towardsdatascience.com
- https://blog.exsilio.com