Malware Analysis

Engineering in Computer Science - Sapienza

HW1 - Machine Learning

Leonardo Salvucci

November 10, 2019

1 Introduction

Machine learning can help us to build tools that can support the analyst during the analysis process. In this report will be analysed two common problem of compiler and optimization provenance. In particular, given the binary code of a functions, we want predict the compiler and the optimization who produced it. It will be used a supervises learning approach given a labelled training set, performig different models that allow us to predict the expected values with different approaches.

The first step is the extraction of the dataset and its vectorization. After that, the data are processed by the chosen model, which, after training, should be able to generalize, managing to predict unknown values.
An important part of the homework concerns the analysis and the choice of the best model according to some parameters that will be explained in the next sections. Finally, the chosen model (for each classification problem) will make predictions on a new dataset, to evaluate the ability to generalize.

2 Dataset

The dataset is provided as a jsonl file. It contains 3000 functions, each of them compiled with three different compilers: *gcc*, *icc* and *clang*. The compiler distribution is proposed, unlike the optimization, having 1000 functions for each completer.

In particular way, each row is a json object with the following keys:
instructions: it contains the whole list of assembly command for the specific function;
opt: it is the optimization used to compile the code. It can take only two values (L(low) and H(high));
compiler: it is the compiler used to compile the code. It can assume only

three values (gcc, icc and clang).

3 Preprocessing

Before passing the data to the model, we must perform some basic operations: the first step is the extraction of dataset from a jsonl file, after that there is the transformation of the data into numerical features, and finally there is the division of the entire dataset.

3.1 Datset extraction

As mentioned before, the dataset is a jsonl file, composed of 3000 json objects. It has the following structure:

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

To extract the dataset I create a function named *get_dataset()* that create three list (*instructions*, *opt* and *comp*) and split the json object in three part by the key and add them to the specific list. Considering only the mnemonic of each instruction, each of these was splitted considering only the first word of the instruction.
So, the result of the previous example is:

```
['xor cmp mov seta cmovae ret']
['H']
['gcc']
```

3.2 Vectorizer

Feature consists in transforming arbitrary data, in this case text, into numerical features usable for machine learning. It convert a collection of text documents to a matrix of token counts. This implementation produces a sparse representation of the counts using *scipy.sparse.csr_matrix*.

```
vectorizer = CountVectorizer(ngram_range=(2,2))
X_all = vectorizer.fit_transform(X)
```

CountVectorizer, without specifying any parameters, simply counts the occurrences. This is wrong for me, since the order is important for training the model.

To keep the order of the instructions I added the ngrams option. An n-gram is just a string of n words in a row. For example, the sentence 'I am Leonardo' contains the 2-grams 'I am' and 'am Leonardo'. The sentence is itself a 3-gram. Set the parameter ngram_range=(a,b) where a is the minimum and b is the maximum size of ngrams you want to include in your features. The default ngram_range is (1,1). Finally, the function *fit_transform(X)*, learn all instructions passed and return term-document matrix.

3.3 Dataset split

It's necessary to divide the dataset into two parts: a subset to train a model and a subset to test the trained model.

```
X_train, X_test, y_train, y_test = train_test_split(X_all,
 y_all, test_size=0.3, shuffle=True)
```

It split matrix created before into random train and test subsets. I chose 70% of the dataset for training and the remaining 30% for the test. I added the option *shuffle=True* to shuffle data before splitting.

4 Classification models

I have selected two classification models on which to make predictions: `Support Vecto Machine`, `Multinomial Naive Bayes` and `Random Forest`.

4.1 Support Vector Machine

The objective of a Linear SVC (Support Vector Classifier) is to fit to the data you provide, returning a "best fit" hyperplane that divides, or categorizes, your data. Intuitively, a good separation is achieved by the hyper-plane that has the largest distance to the nearest training data points of any class (so-called functional margin), since in general the greater the margin, the lower the error of generalization of the classification. The distance between the support vector and the hyperplane is referred to as margin. Generally, hyperplane can take as many dimensions as we want. The goal is to minimize the Training Error + Complexity term. So, we choose the set of hyperplanes, so f(x) = (wx)+b.

```
model = svm.LinearSVC()
model.fit(X_train, y_train)
```

4.2 Multinomial Naive Bayes

Naive Bayes is the classification machine learning algorithm that relies on the Bayes Theorem. The main point relies on the idea of treating each feature indipendently. Naive Bayes method evaluates the probability of each feature independently, regardless of any correlations, and makes the prediction based on the Bayes Theorem.
Multinomial Naive Bayes simply lets us know that each p(fic) is a multinomial distribution, rather than some other distribution. This works well for data which can easily be turned into counts, such as word counts in text.

```
model = MultinomialNB()
model.fit(X_train.todense(), y_train)
```

4.3 Random Forest

Random forest, like its name implies, consists of a large number of individual decision trees that operate as an ensemble. Each individual tree in the random forest spits out a class prediction and the class with the most votes becomes our model's prediction. The low correlation between models is the key, uncorrelated models can produce ensemble predictions that are more accurate than any of the individual predictions. The reason is that the trees protect each other from their individual errors (as long as they don't constantly all err in the same direction). While some trees may be wrong, many other trees will be right, so as a group the trees are able to move in the correct direction.

```
model = RandomForestClassifier(n_estimators=100, n_jobs=-1)
model.fit(X_train, y_train)
```

n_estimators is an optional integer (default=10) that indicates the number of trees in the forest. *n_jobs* indicates the number of jobs to run in parallel (-1 means using all processors).

5 Evaluation

5.1 Optimization

5.2 Compiler

6 Result

7 Conclusion

This report presents a performance evaluation of selected symmetric encryption algorithms. The selected algorithms are AES, DES, RC2 and Blowfish.

As we have seen, the relationship between encryption and decryption speed depends on the operating mode used. The results obtained are in accordance, except for ECB, with the theoretical lines studied.
It is easy to notice that the operating modes, that allows the parallelization, improve performances of the algorithm (in most of the studied cases this concerns only the decryption).
It was interesting to note the behavior of the algorithms, with different operating modes, as the input size increases.
The presented simulation results showed that AES has a better performance than other common encryption algorithms used, but its speed is more subject to variation when the input size changes, compared with the other algorithms.

References

[1] https://wiki.openssl.org/index.php/Enc

[2] https://en.m.wikipedia.org/wiki/Block_cipher_mode_of_operation