Machine Learning, HW1 - Compiler provenance

Riccardo Caprari, ID number 1743168

Fall 2019

Contents

1 Introduction

One of the most common problems in binary analysis is to find compiler provenance. In particular, the aim of this homework is to identify the optimization level and the compiler who produced a given program. The importance of predicting the compiler is given mainly by aspects related to cybersecurity. In detail, it can provides a more accurate disassembly of a malware.

In order to try to identify a compiler and an optimization level it's convenient to use machine learning techniques. The objective is to learn a function $f : X \mapsto Y$ from a given data set $D = \{(x_n, t_n)_{n=1}^{N}\}$ in which x_n are the domain points, so called features, and t_n their relative labels. This process is done by initially parsing a fairly large data set and then extracting features from it, which permits us to proceed with the analysis. The set of features and their labels are divided now into two sets, one for the training and the other one for the test. The first one will be used to train a learning algorithm which has a set of hyper parameters to be set. On the second set, instead, will be applied the model generated in the previous step in order to predict it and obtain an evaluation of the classifier. There will be actually two variants of model for the evaluation. In the final state the evaluation will produce the performance results which are important to understand the effectiveness of the model. In Fig. 1 it's possible to visualize all the proceedings.

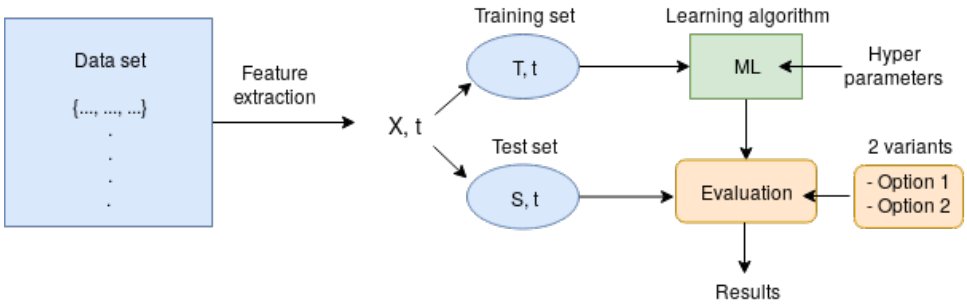


Figure 1: Homework structure.

Having two main characteristics (compiler and optimization) for every program, it's necessary to split the problem into two subproblems which

means two different models using classification algorithms. In particular, the classification of optimization is done through a binary classifier with classes $HIGH$ or LOW. The compiler classification, instead, can be done using a multi-class classifier which classes are *gcc*, *icc* and *clang*. To better understand how good the prediction of the model will be it's necessary to use evaluation metrics. In this homework, the metrics computed in the evaluation step will be log loss, precision, recall and f1-score.

The prediction of the test set, besides being a means of evaluation, it's not the main objective of the problem. In fact, together with the initial data set there is a blind data set composed of 3000 samples. Each sample is a single function in x86 which means there won't be an optimization level and its compiler. Therefore, the main objective is to predict the labels of a certain function using the trained model generated in the previous steps.

2 Data set parsing

The data set that has to be parsed in order to extract the features contains 30000 functions of x86 instructions. Each function has a compiler and an optimization level associated to it and it's known that the distribution of the first class is very balanced while the distribution of the second is not balanced. Dataset is provided as a jsonl file in which each json object has the keys *instructions*, *opt*, *compiler*.

File parsing is relatively simple using *Python*. In particular, in the file parser, there are three lists, one for each json key. To allow a quick and targeted extraction of the features in the next step, only the mnemonic of each instruction has been selected from the first list, except for a small set of mnemonics which are considered *basic* and for which it also has been taken the target register. An x86 instruction is in fact generally composed by a mnemonic and one or more registers which are the parameters involved in the action. One example is *mov edi 0*: *edi* is the target register and 0 is the value that will be moved in it. The *basic* mnemonics I've chosen are *mov* and *lea* in order to include also their target registers in the final instructions list. This choice is justified from the fact that, adding also the registers of these two *commands*, I've obtained a better performance during the evaluation process (discussed in Subsection 6.1).

3 Features extraction

One of the most important choices in a machine learning model is certainly how to extract the features. After a good parsing of the dataset it's now the time to transform the instrunctions, which are set of strings, into something which can be easily usable by the learning algorithms.

A big amount of features can lead to a much slower execution of the model. In this case, thanks to an accurate analysis of the data set and to dimensionality reduction, there will be a total of 491 features (instead of 567 180). The process of feature selection, in this case, is done in part before the extraction through data set parsing. Selected variables from the data set are, like said before, strings. This means that it's possible to intepret them as a collection of text documents. The objective is to obtain a sparse matrix that has a dimension of *#functions* x *#features*. It's useful to use a sparse representation with most of the entries equal to zero because it optimizes the use of memory and permits to train the model much faster. In particular, I separately used two vectorizer functions from sklearn package:

$$CountVectorizer() \tag{1}$$

$$TfidVectorizer() \tag{2}$$

The first one, which is the best from an evaluation metrics point of view, literally counts the number of times that a mnemonic or a register appear in a x86 function, generating a dictionary of features. Also *TfidVectorizer* counts the number of occurences but, instead of *CountVectorizer*, there is another transformation process. In fact, all the occurences are transformed into frequencies which are basically the number of occurences of each word divided by the number of the total number of words. In the real world, for an x86 program, the frequency of a certain instruction means the time spent by the process to execute them. That's why Func. 2 could be a useful vectorizer to use for this type of problem. After a first test with Func. 1 and 2 I obtained 491 features. This was expected, because the number of features is equal to the number of terms in the initial vector if no selections were applied. Then, I decided to improve the selection and add a parameter in both the functions, which is *min_df* and permits to ignore terms that have a frequency strictly lower than a fixed value. The value I've chosen for this parameter is *min_df*=5 and this changed the total number of features to a total of 358.

Adding this parameter, there is a reduction of the model training time and a deletion of unnecessary features, maintaining the same performances (Subsection 6.1).

4 Learning algorithms and hyperparameters

Now that the sparse matrix of the samples with their features is ready it's possible to proceed with the most time-expensive part: learning algorithms.

First of all, as shown in the diagram 1, it's necessary to split the sparse matrix into random train and test subsets. The first subset will be actually used by the classifiers for the learning process and the second one to evaluate the efficacy of the learning algorithms. To easily split the set I've used *train_test_split()* function with parameters the matrix with his vector of labels[1], *test_size*=0.2 which is the portion dedicated for the test set and *random_state*=15 for the random number generator. Having X as set of samples and Y its set of labels and splitting both in two subsets means a total of four subsets: *X_train, y_train* and *X_test, y_test*. The first two are the input of the classifier that will proceed with the learning process.

Regarding the classifiers, I will proceed with the list of those I have used, including their hyperparameters, and present the final variants in Chapter 5. To search for the best classifiers (Subsection 6.3), in terms of performance and consistency with the problem, I've tested many of them: one of the best is the **Random Forest Classifier**. Its efficacy is due to its randomness. In fact, using decision trees, the algorithm is able to predict labels using the value that appears more often in the classes. Normally, a decision tree is synonym of overfitting. Instead, random decision forests correct, or at least decrease, this problem and this is one of the main reason why I've chosen it. The second reason is that it works efficiently with multi-class problems, like the one for the compiler provenance. I decided to leave by default the hyperparameters of this classifier except for the number of estimators, which are the number of trees in the forest. In fact, by setting *n_estimators*=200 instead of a default value of 10, it's possible to increase the precision of the model as shown in Subsection 6.2.

[1]The vector depends on the classification problem: opt or compiler.

The second classifier I've chosen for the homework is the **Gradient Boosting Classifier**. It also uses decision trees, which are weak prediction models, but generates a "wise" prediction model in the form of an ensemble of them. Thanks to a different way to build trees, GBC is considered better than RFC. The reason why I've chosen this classifier, in addition to its great performances, is that it's normally used in anomaly detection problems like the ones in cybersecurity. Another main reason to choose it is that it works well with unbalanced data like the one we have for the optimizations (binary classification problem). Despite this, GBC may request an higher execution time and, therefore, it's slower than RFC. The hyperparameters of this second classifier I've changed are again the number of estimators (*n_estimators*=200) and the maximum depth which limits the number of nodes in the tree (*max_depht*=7).

The assigned values of the hyperparameters are chosen manually in the range of the best ones. In Subsection 6.3 I will highlight the main differences and improvements that lead to these decisions. Another way I tried to choose the optimal hyperparameters is using *GridSearchCV()*, a function from sklearn package that iterates over a range of hyperparameters to find the best ones. It's a good method to reach the best performances but it requires a large computational time and, mostly, optimal hyperparameters aren't the objective of this homework. That's why i preferred to manually choose them.

After the choice of the classifier, it's time to train the model. To proceed with this step I've used the fit() function which takes as parameters the two training sets generated before. After the training is completed, the model can predict the test set and it's possible to calculate the performance in the evaluation process, in particular using *classification_report()* function.

5 Model variants

In this section are present all the variants for the classification problems of the homework. Each case is labeled as: M (model) + C/O (compiler or optimization) + initial classifier letters. The best hyperparameters and the best features extraction are reported for each case.

5.1 MCRF

Classification: compiler
Feature extraction: TfidVectorizer
Classifier: Random Forest
Hyperparameters: *n_estimators*=200

5.2 MCGB

Classification: compiler
Feature extraction: CountVectorizer
Classifier: Gradient Boosting
Hyperparameters: *n_estimators*=200, *max_depth*=7

5.3 MORF

Classification: optimization
Feature extraction: CountVectorizer
Classifier: Random Forest
Hyperparameters: *n_estimators*=200

5.4 MOGB

Classification: optimization
Feature extraction: CountVectorizer
Classifier: Gradient Boosting
Hyperparameters: *n_estimators*=200, *max_depth*=7

6 Results

This last chapter reports the performance outcomes of the discussed variants and the chosen strategies to complete the steps of the homework.

6.1 Ways to extract the features

As mentioned in Chapter 3, the naive features extraction, which means selecting all the variables inside the functions, produces a total number of 567 180 features. This way increases a lot the execution time of the program but gives a good contribution on the evaluation metrics if the dataset is balanced (like the compiler one). For example, running the naive extraction with MORF (5.3) produces these outcomes:

	precision	recall	f1−score	support
H	0.83	0.69	0.75	2388
L	0.81	0.91	0.86	3612

The second strategy I've tried out is the one of selecting all the mnemonics with the addition of the target register for the "special" ones. This significantly reduces the execution time and the number of features (491), increasing also the level of performance. Testing again MORF with the new settings:

	precision	recall	f1−score	support
H	0.82	0.79	0.81	2388
L	0.87	0.88	0.87	3612

The third component to finish the extraction of the featuers is adding a features selection, which consists on discarding the less frequent ones. Let's run MORF with the last edit:

	precision	recall	f1−score	support
H	0.82	0.79	0.80	2388
L	0.86	0.88	0.87	3612

There are no evident score differences in the last two steps, using a different amount of features, which means it has been obtained a better and optimized strategy.

6.2 Finding the best hyperparameters

The values for the hyperparameters of the models have been chosen manually, looking for the best score. For example, these are the metrics scores for MCRF using default parameters:

	precision	recall	f1−score	support
clang	0.80	0.85	0.82	2009
gcc	0.82	0.82	0.82	1994
icc	0.86	0.81	0.83	1997

And these are from the same model with a much larger number of estimators, from 10 to *n_estimators*=200:

	precision	recall	f1−score	support
clang	0.88	0.90	0.89	2009
gcc	0.90	0.89	0.89	1994
icc	0.90	0.89	0.89	1997

It's evident that this parameter has an high impact on the performance of the model. Nevertheless, I noticed that continuing to increase it doesn't lead to an improvement.

Regarding the Gradient Boosting classifier, I've find out that also here the number of estimators has an important impact. Analyzing all the parameters I also discovered that the higher contribute on the performance is given by the maximum depth of the generated trees which is equal to 3 by default. These are the scores of MCGB with default hyperparameters:

	precision	recall	f1−score	support
clang	0.82	0.82	0.82	2009
gcc	0.82	0.81	0.82	1994
icc	0.81	0.83	0.82	1997

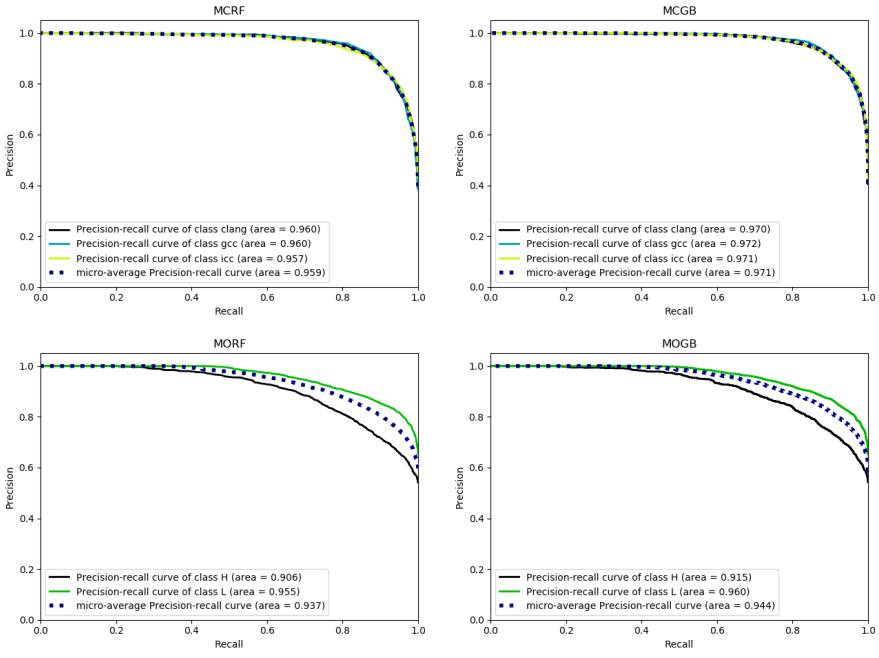
And now the scores, assigning the new values:

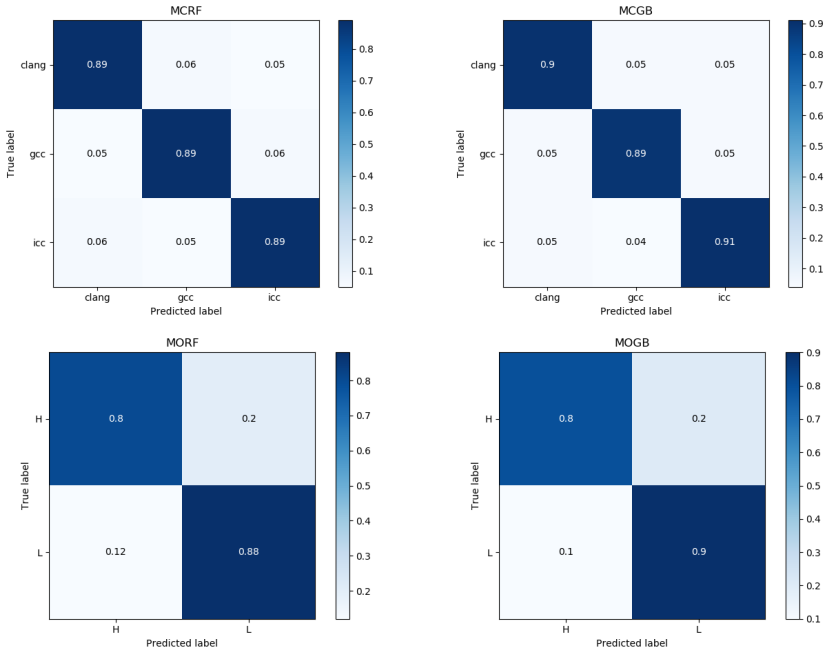
	precision	recall	f1−score	support
clang	0.91	0.90	0.91	2009
gcc	0.91	0.90	0.91	1994
icc	0.89	0.92	0.90	1997

6.3 A comparison between the variants

The good common point of the variants is that none of them suffers of a large *overfitting*. For example, using MCGB with *max_depth*=4 gives an average precision of 87% on the test set, which is the blind one, and 91% on the training set on which the model has been trained. Instead, using the classic MCGB produces an average precision of 91% on the test set and 97% on the training set. I preferred to use the second one for a more accurate prediction. Random Forest models, instead, suffer a bit more the overfitting but it's not so critical. For example, using MCRF, the average precision on test set is 90% and 100% on training set, which means the model is able to perfectly predict the samples from which it trained. In general, a "learn by heart" is not adequate for a machine learning algorithm. However, the precision on the test set is still high, which means the overfitting isn't dangerous for the model correctness.

Now I'll proceed with a graphic representation of the results produced from the four variants. In particular, the precision-recall curve of the models that permits to compare them calculating the AUC (area under the curve) and then the shape of the confusion matrices with the predictions.





It's possible to notice that the performances of both the classifiers RF and GB are quite similar. It's evident that, due to the unbalanced data for the optmization classification, the precision-recall curves for the compiler classes are more stable than the optimization ones. However, the models are working good, as showed in the confusion matrices, in which the predicted labels are almost equal to the true ones. This gives a good hope for the correctness of the final blind test of the homework.

Another important metric that can be used to compare the models is the log loss function, which takes into account the uncertainty of the predictions based on how much it varies from the actual label. These are the log loss values of the four variants:

MCRF: 0.4611359657999586
MCGB: 0.2512627024002483
MORF: 0.35343121698665475
MOGB: 0.3058825439004223

Also this comparison shows that GB performs better than RF, having a lower log loss. In fact, log loss is a value between 0 and 1 and lower is the better.

For the last comparison I decided to use the f1-score, which embed precision and recall into its computation. These are the computed f1-scores:

MCRF: clang 0.89
gcc 0.89
icc 0.89

MCGB: clang 0.90
gcc 0.91
icc 0.91

MORF: H 0.81
L 0.87

MOGB: H 0.82
L 0.88

In all the outcomes GB has the best scores and it was quite predictable since it is a classifier that works well with these types of problems .

6.4 Best models for the blind test

After a careful analysis of the models, I decided to choose MCGB (5.2) and MOGB (5.4) to predict the blind test of 3000 samples.