Microsoft Malware Classification Challenge (BIG 2015):

First Place Team: Say No to Overfitting

Xiaozhou Wang, xiaozhou@ualberta.ca
Jiwei Liu, University of Pittsburgh, aixueer4ever@gmail.com
Xueer Chen, University of Pittsburgh, xuer.chen.human@gmail.com

Abstract

This document describes our approach to the Microsoft Malware Classification Challenge (BIG 2015). Our approach is based on intensive Feature Engineering, Gradient Boosting (Xgboost), Ensembling, Semi-supervised learning and calibration. We achieve the 1st place with 0.0028 multi-class logarithmic loss in the private leaderboard.

1. Introduction

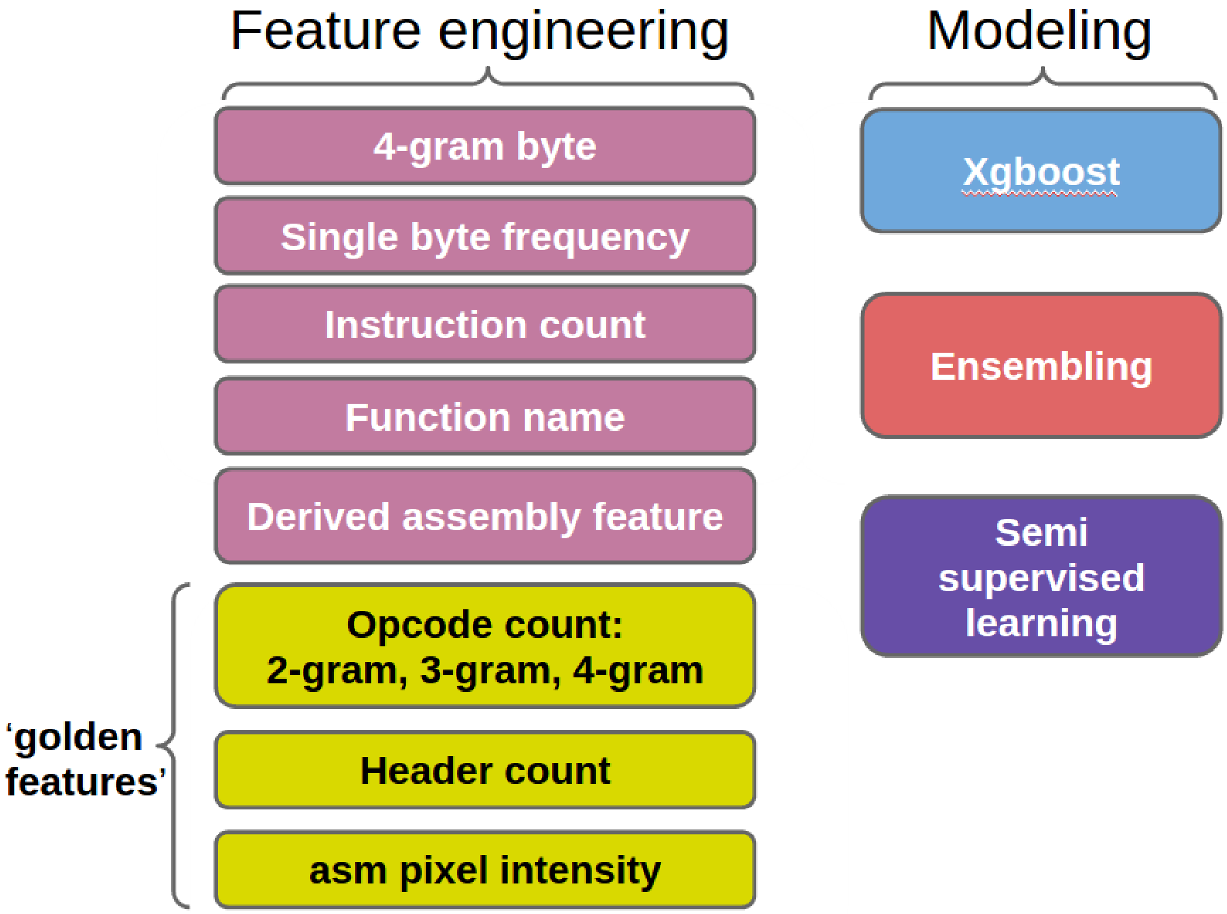
The goal of Microsoft Malware Classification challenge is to classify malwares into 9 classes. The total data size is 500 GB, including 10868 and 10873 malwares in train and test set, respectively. For each malware two types of files are provided: hexadecimal byte contents and asm files produced by a commercial Disassembler IDA pro. We observe that it is fairly easy to achieve a 0.02 log loss score with simple instruction or byte count features. Given the relatively small data set in terms of number of instances, the challenge is to avoid overfitting to outliers while minimizing log loss at a very low level.

Our approach contain 2 major steps:

- Feature engineering: we produce single-byte frequency, byte 4-gram, instruction count, function names and Derived assembly features [5]. In addition to that, we created three relatively novel features: **'interesting' opcode count, segment count and asm file pixel intensity**. These three new features are golden features to our models.

- Modeling: the main model is Gradient Boosting. We used the package Xgboost developed by Tianqi Chen and Bing Xu. We also come up with three techniques to further minimize the log loss, Ensembling, Calibration and Semi-supervised learning. Among them, Semi-supervised learning achieve the best performance.

It is worth mentioning that cross-validation plays a key role in our fight against overfitting. All the feature engineering and modeling techniques above are examined and selected using cross validation. Our final solutions are also selected based on local cross validation score rather than public Leaderboard. The best final model we selected used ensembling, calibration and semi-supervised learning, which achieves 0.0028 private LB and 0.0036 public LB.



Overview of our approach

2. Feature Engineering

The features we created can be divided into two parts:

1): Single-byte frequency, byte 4-gram, instruction count, function names and Derived Assembly Features (DAF). These features are inspired by paper [5] and beat benchmark code in the forum [2]. For 4-gram bytes, we use info gain to select the best 500 features for each class. For instruction count, common instructions like 'mov' and 'jmp' are counted. Function name features were supposed to be DLL features in [5]. We replace it by gathering only the function names instead for simplicity. For DAF, each feature is a standard representation of asm instructions, following the format: name.para1.para2, such as or.memory.register.

2): Novel features: ‘Interesting’ opcode N-gram, asm file segment count and asm file pixel intensity. These features also boost the performance greatly. We will elaborate on how these features are generated.

2.1 ‘Interesting’ opcode N-gram and segment features

For the instruction count feature above, we subjectively select a ‘common’ subset of X86 assembly instructions as candidates. Such an approach is very unlikely to capture the truly important information in the data set. To find the truly interesting opcodes, we followed a simple intuition: the 'interesting' opcode has to be frequent in at least one asm files. This idea is inspired by the Apriori algorithm: to extract N-gram features based on a certain set of opcodes, the 1-gram count of these opcodes have to be frequent. In practice, we count every token in asm file, except for bytes and addresses, and only select the tokens that appear more than 200 times in at least one asm file. 200 is threshold set manually.

In addition to a simple thresholding, we also consider the influence of endless loop caused by unconditional jump instruction such as ’jmp’ [5]. If an opcode live inside an endless loop, the time it is executed will be much more frequent than the time it appears in the asm file. Hence, when counting the tokens in the asm file, the counter follows the unconditional jump instruction to the address it points to. If an endless loop exits, the counters for the tokens within the loop will approach infinity. We set a threshold so that if the counter exceed the threshold, it breaks the loop and count the opcode outside the loop. This process is quite coarse since we ignore all the conditional jumps so it cannot reflect the real control flow order. However, we found the opcodes selected in this process is much more useful than the fixed instruction set in part 1.

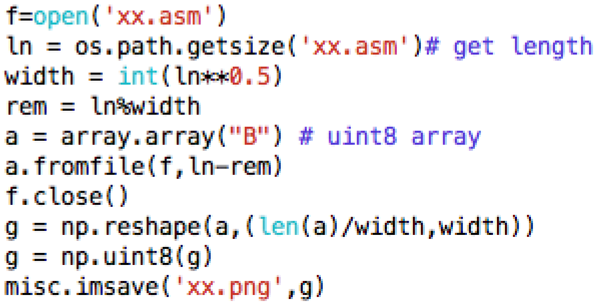
In the end, we extracted 165 such ‘interesting’ tokens and most of them turn out to be opcodes/assembly instructions. We extract 2-gram, 3-gram and 4-gram counts based on this ‘interesting’ opcode set. To reduce the number of features, we set different threshold at each step so only the most frequent N-grams are counted. It should be noted that unlike selecting interesting opcode above, when counting 2-gram, 3-gram, 4-gram features, the counter no longer follows the unconditional jump instruction, instead it follows the static order in the text only. The reason is that we lack the domain knowledge to reconstruct a precise control flow graph. We tested both but the performance of using a simple static order is better in terms of cross validation.

Segment count: the asm file can be divided into several segments based on the key word at the beginning of each line, such as text, code, data and idata. We found that a simple count of lines of each segment in the asm file is a good feature. This is also discovered by other teams in the competition [6]. There are 448 different segments in the training set in total and we count how many lines each segment has in each asm file.

We found that combining interesting opcode N-gram features and segment count features gives the best result. The total number of features is 71342 and it is very sparse. Since our classifier works better with dense features, we use the random forest classifier in scikit-learn[1] to select useful features. After this step we have roughly 4k features. Due to the random factor in random forest, the exact feature set varies in each run but the performance is pretty stable.

2.2 Asm file pixel intensity feature

Malwares can be visualized as gray-scale images using the byte file [8]. Each byte is from 0 to 255 so it can be easily translated into pixel intensity. However, we found the image processing techniques in [8] doesn’t work well with our features above. Inspired by the [7], we tried to extract a gray-scale image from asm file rather than the byte file. The code is shown in Figure 1. Figure 2 compares the byte image and asm image of the same malware.



```
f=open('xx.asm')
ln = os.path.getsize('xx.asm')# get length
width = int(ln**0.5)
rem = ln%width
a = array.array("B") # uint8 array
a.fromfile(f,ln-rem)
f.close()
g = np.reshape(a,(len(a)/width,width))
g = np.uint8(g)
misc.imsave('xx.png',g)
```

Figure 1. code to transform asm file to gray-scale image based on [7]

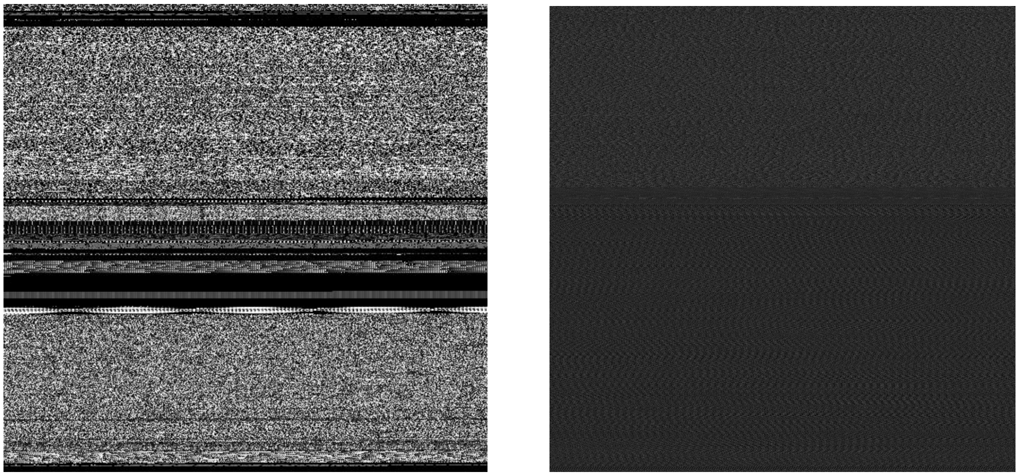


Figure 2. byte image (left) and asm image (right) of the same malware.

We found that the intensities of the first 800 or 1000 pixels in the asm image is a good feature for the malware. We don’t understand why it works for now. The performance of using this feature alone is not impressive but when it is used together with the previous features, the performance is improved significantly. The best single feature set is the N-gram opcode features and we add other features step by step. As shown in the table below, we observe that feature interaction is the key. Table 1 shows the importance of each feature set evaluated by cross validation and Leaderboard. We use a single mode with Xgboost to achieve these scores.

Feature set	Cross validation	Public Leaderboard	Private Leaderboard
N-gram opcode count (best single feature set)	0.014	0.014	0.010
N-gram opcode count segment count	0.010	0.0087	0.0079
N-gram opcode count segment count asm image pixel intensity	0.0058	0.00419	0.00419
All features	0.0052	0.0057	0.0030

Table 1 Log loss of a single model with different feature set

3. Modeling

We use Xgboost [2] to do multiclass classification with the softmax objective. Xgboost is a very efficient Gradient Boosting package that is widely used in kaggle competitions. There are 4 important parameters: eta, step size shrinkage, max_depth, maximum depth of a tree, num_round, the number of round for boosting and min_child_weight, minimum sum of instance weight(hessian) needed in a child. A nice document [3] can be found to explain the insight in Xgboost [2]. We also tried Random Forest, Naive Bayes and Neural Network but Xgboost’s performance is supreme. With selected features and some parameter tuning, we can reach 0.0030 log loss in private LB with our best single model, which could be the 1st place already.

We built three single models with different feature set and parameters. For ensembling we found weighted geometric mean gives the best result. The weights are obtained through grid search with cross validation.

Another important technique we come up is Semi-supervised Learning. We first generate pseudo labels of test set by choosing the max probability of our best model. Then we predict the test set again in a cross validation fashion with both train data and test data.

For example, the test data set is split to 4 part A, B, C and D. We use the entire training data, and test data A, B, C with their pseudo labels, together as the new training set and we predict test set D. The same method is used to predict A, B and C. This approach, invented by Xiaozhou, works surprisingly well and it reduces local cross validation loss, public LB loss and private LB loss. The best Semi-supervised learning model can achieve 0.0023 in private LB log loss, which is the best score over all of our solutions. Table 2 shows our final solution.

		cross validation	public LB	private LB
sub 1	ensemble without semi-supervised learning	0.0041	0.0035	0.0031
sub 2	ensemble with semi-supervised learning	0.0031	0.0036	0.0028

Table 2 Summary of our two final submissions.

4. Conclusion.

In this challenge, we discovered several novel features such as the segment count and asm file pixel intensity. We observe that feature interaction is the key and it is best exploited by the Gradient Boosting model (Xgboost). We also propose a semi-supervised learning technique to further optimize the log loss of the test set. Cross validation plays a critical role to overcome overfitting. Our parameter tuning and model selection is based local cross validation rather than the public leaderboard.

5. Acknowledgement

We would like to thank Tianqi Chen and Bing Xu for developing the great machine learning package Xgboost. We also thank Lakshman Nataraj for his great work on visualizing Malwares.

6. Hardware configuration, Dependencies and Code Description

- Hardware configuration: Google Compute Engine (instance with 104G memory with 16 cpus), 1 TB Disk
  - 100 GB memory is only required to generate byte 4 gram and DAF features
  - 32 GB memory is enough to the other features and do all the modeling.
  - it requires about 700 GB disk space. 500 GB space for the original data and 200 GB for the meta data generated. The final features used by the classifier are only 4 GB.
- OS: Debian 7 (wheezy). But as long as xgboost can run on it, any unix liked OS works.
- pandas 0.16.0
- sklearn 0.16.1.

- pypy 2.5.1
- python 2.7.6
- numpy 1.9.2
- xgboost-0.22 https://github.com/dmlc/xgboost/releases/tag/v0.22
- We provide two versions of code, a full version and a small version.
    - The small version is to test if everything is setup correctly. It takes 5 mins to generate a result.
    - The full version is to generate our solutions. Please refer to readme for more details.
        - To generate the best single model, run

          ```
          ./single_model.sh
          ```

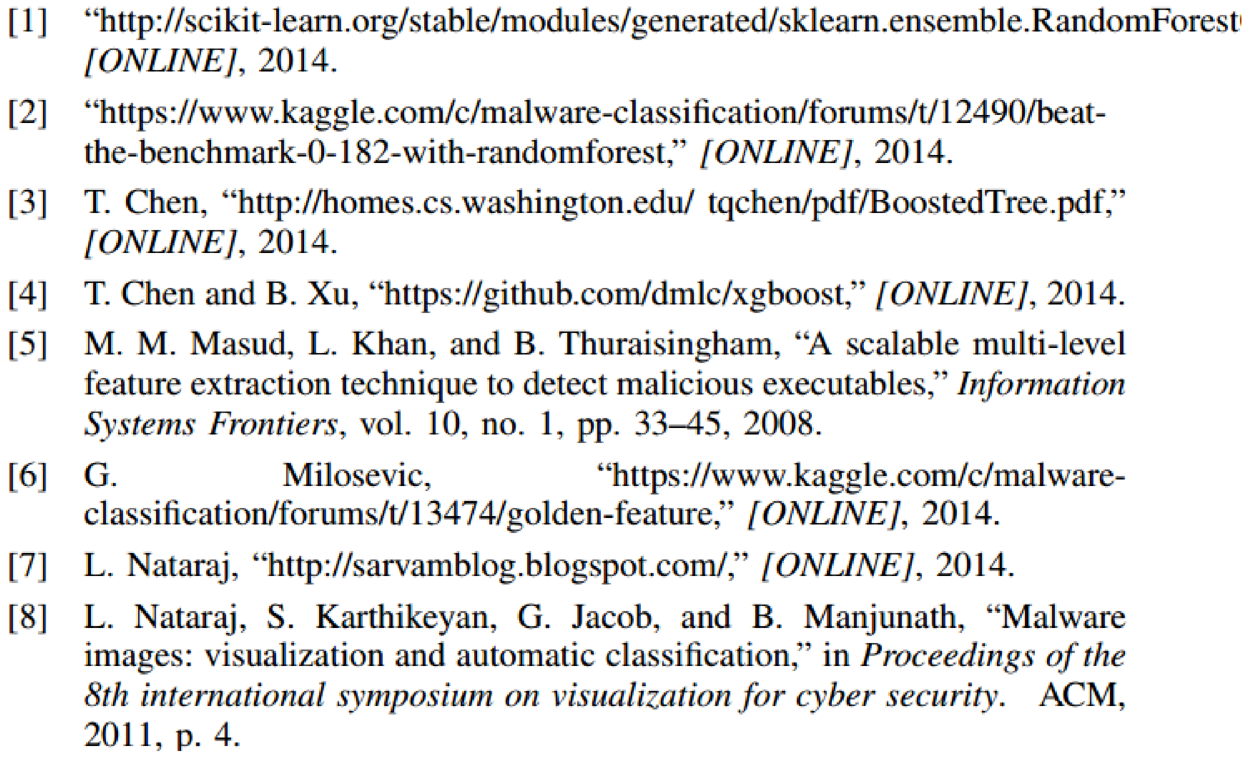
        - To generate the best ensemble model, run

          ```
          ./single_model.sh     # if not run yet.
          ./semi_ensemble.sh
          ```

    The best single model generates a file, ‘submission.csv’, which gets 0.002997894 in the private leaderboard. It takes about two days to run. Most time is spent on featuring engineering. The modeling part takes less than 1 hour.

    After the best single model finished, the best ensemble model can be run, to generate a file, ‘best_submission.csv’, which gets 0.002434706 in the private leaderboard. It will take another day to generate this submission.

7. Reference.



[1] “http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForest *[ONLINE]*, 2014.

[2] “https://www.kaggle.com/c/malware-classification/forums/t/12490/beat-the-benchmark-0-182-with-randomforest,” *[ONLINE]*, 2014.

[3] T. Chen, “http://homes.cs.washington.edu/ tqchen/pdf/BoostedTree.pdf,” *[ONLINE]*, 2014.

[4] T. Chen and B. Xu, “https://github.com/dmlc/xgboost,” *[ONLINE]*, 2014.

[5] M. M. Masud, L. Khan, and B. Thuraisingham, “A scalable multi-level feature extraction technique to detect malicious executables,” *Information Systems Frontiers*, vol. 10, no. 1, pp. 33–45, 2008.

[6] G. Milosevic, “https://www.kaggle.com/c/malware-classification/forums/t/13474/golden-feature,” *[ONLINE]*, 2014.

[7] L. Nataraj, “http://sarvamblog.blogspot.com/,” *[ONLINE]*, 2014.

[8] L. Nataraj, S. Karthikeyan, G. Jacob, and B. Manjunath, “Malware images: visualization and automatic classification,” in *Proceedings of the 8th international symposium on visualization for cyber security*. ACM, 2011, p. 4.