Microsoft Malware Classification Challenge third place solution

Mikhail Trofimov, Dmitry Ulyanov, Stanislav Semenov

April 21, 2015

1 Introduction

We present our solution to the Microsoft Malware Classification Challenge. In this competition we were asked to classify malware executables into nine families based on their hex dump and IDA disassembled representation. Our team extracted numerous of features from both hex dump and IDA code and fit GBDT on it. We augmented the train set using test set and used per-class model mixing scheme in order to improve the result.

2 Features

As opposed to the majority of the competitions at Kaggle raw data was provided and no pre-extracted features were available. Therefore, the most challenging task was to find discriminative attributes of the malware. We started with the easiest to extract: byte counts, byte bigrams, assembler instruction counts, section counts, files length and size. It was a surprise, that one can achieve the accuracy of 0.96 just by using counts of '00', 'FF' and '??' .

Up to that moment every feature we added improved our score significantly but it became difficult to find a new feature, which was not only good by itself, but also contained additional information to already extracted.

We built the features, which should have characterized the executables logic: opcodes n-grams, names of the imported functions, libraries, the number of procedures. We computed the "contitionality" of the executable: the number of "loc_*" references in *.asm* file. We noticed that the malware content is often encrypted inside the binary, so we wanted to introduce some encryption related features. We computed entropy over a sliding window of half-byte sequences, extracted some statistics of its distribution (20 quantiles, 20 percentiles, mean, median, std, max, min, max-min), statistics of first order differences distribution and parts of the entropy sequence. We computed compression ratio (as an approximation of Kolmogorov complexity).

We extracted strings but ended up not using them directly, instead we calculated characteristics of string length distribution for each object. Based on [1], [2] we extracted 4-grams and even 10-grams, which we found to perform well.

3 Feature selection and dimensionality reduction

All our features can be broadly divided into sparse (e.g. imported functions names counts) and dense (statistics of distributions, etc.). The overall dimensionality of sparse features was much higher than the dimensionality of the dense features, hence the sparse features would be considered more often when using both directly in Random Forest. This degrades the performance since there are a lot more non-informative features among the sparse ones.

We used two approaches to dimensionality reduction: transformation and selection. We observed that a lot of our features with dimensionality more than 100 can be easily reduced to ≈ 10 dimensions preserving most of the information. Surprisingly, Non-Negative Matrix Factorisation (NMF) worked much better than PCA or ICA, which can be explained by the nature of features – the counts are non-negative. We used NMF on the raw data as well as on the data transformed by $\log(1 + x)$.

Another technique we used a lot is the following: we omit rare features, applied SVM with L1 regularization to roughly separate irrelevant features. Then we used RF feature importances for better selection. Using this pipeline we selected 131 4-grams (out of $(256 + 1)^4$ possible) and could get 0.01473 cross-validation score. See list of of the most important 4-grams in table 1.

1	????????
2	04000000
3	5DC30000
4	F0F0F001
5	00100000
6	00F0F000
7	0D2F0600
8	5DC38BFF
9	8BFF558B
10	840D2F06

Table 1: 10 most important 4-grams

Finally, we used even more feature selection for 10-grams. We applied the pipeline from above, and tried to select 10-grams, which will help to classify exactly the objects with the biggest loss. We generated out of fold predictions for train set and sorted objects by true class probability, labeled as '1' top 100 worse, and run RF on 500 preselected 10-grams. We selected the most important features, and it turned out we can have 1.0 accuracy for this binary task just with 14 10-grams.

4 Learning

We were fitting two models on two distinct features set in order to combine them effectively afterwards (see appendix for detailed description of feature sets). Mixing two classifiers worked out better for us than fitting one with

all the features. During the competition we use 10-folds cross-validation for evaluating our performance.

4.1 Model

We mainly used GBDT classifier (XGBoost implementation [1]). Extremal bagging was very helpful. Instead of using just a train set as is we created a set with 8x times more objects: we took all L train objects and sampled αL more objects with replacement. Alpha can be found by grid search and was set to 7. We averaged the predictions over several runs.

4.2 Semi-supervised trick

Since the classification accuracy was pretty high (> 0.998) we could use test set to get more data for training. The test set was divided into K folds. We used train set and all but one folds at a time for training and predicted the other fold. We sampled labels for test objects based on our best submission on public LB. We were not sampling test objects when using bagging with this scheme.

4.3 Per-class weighting

When comparing two models it turned out they make mistakes differently for different classes, that is why we were mixing predictions for each class separately. For each class the prediction was built as a linear combination of models predictions for that class. We also added 2nd and 3rd order interactions between the base predictions into mix.

4.4 Trimming predictions

We were bold enough to trim the predictions to 0 and 1 for 1,3 and 7th classes. We got lucky and moved from 0.0041 to 0.0039 private score, although we knew that LogLoss does not forgive mistakes at all.

[1] https://github.com/dmlc/xgboost

5 Appendix

5.1 Feature set 1

- line count for every sections ('.bss', '.data', '.rsrc' and so on) in *.asm, entropy of the counts distribution
- 30 most frequent asm instruction counts
- *.asm and *.bytes file sizes and their ratio.
- sys calls (grepped by "__stdcall" from *.asm) + TF-IDF + NMF(n_components=10)
- like previous, but for functions (grepped by "FUNCTION")
- 131 selected 4-grams
- 14 selected 10-grams

5.2 Feature set 2

- single byte counts, transformed using NMF and NMF(log(x+1))
- byte pairs counts, transformed using NMF and NMF(log(x+1))
- 98 assembler instructions counts, some registers counts
- various bags: dll names, imported functions, extern functions, std calls
- keywords counts and registry-like strings ('proc', 'cookie', 'loadlibrary', 'installdir', 'HKEY_LOCAL_MACHINE', 'HKEY_CURRENT_USER', 'sp-analysis failed', '___security_cookie' and so on)
- line counts of every sections
- number of disassembled subroutines
- statistics of entropy distribution
- strings length statistics

References

[1] Ekta Gandotra, Divya Bansal, and Sanjeev Sofat. Malware analysis and classification: A survey. *Journal of Information Security*, 2014, 2014.

[2] J. Zico Kolter and Marcus A. Maloof. Learning to detect and classify malicious executables in the wild. *J. Mach. Learn. Res.*, 7:2721–2744, December 2006.