Microsoft Malware Classification Challenge
2nd place solution documentation

Marios Michailidis (London, United Kingdom), mimarios1@hotmail.com
Gert Jacobusse (Goes, The Netherlands), gert.jacobusse@rogatio.nl

1. Summary

In the Microsoft malware classification challenge, we were given over ten thousand malware files from 9 different labeled classes (the train set). The challenge was to predict the hidden class labels of another set of over ten thousand files (the test set). Each file was provided in two versions: as a bytes file (without header, in hexadecimal text) and as an asm file with information that the IDA disassembler extracted from the binary file. Our approach consisted of two steps that were repeated multiple times: feature extraction and model training.
We had features of three types:
1) on file properties: size, compressed size and ratios between them - for both bytes and asm files
2) on contents of the asm files: sections, dlls, opcodes and interpunction statistics by section
3) on contents of the bytes files: 1,2,and 4 grams, full lines and distribution of entropy
Model training consisted of two main steps: first to apply different classification models to different subsets of the features, and second to combine the models in a meta bagging model using different weights for each model. We ended up using Gradient Boosting and Extremely Randomized Trees as classification models. Performance was estimated using cross validation within the train set, so we did not rely on test set feedback to decide about the models' weights.

2. Features Selection / Extraction

File properties

Some very simple file properties are quite useful to distinguish between the 9 malware classes (see Figure 3). We used file size of both bytes and asm files, compression rate (using python zipfile) of both file types and some ratios between them:

ab_ratio: asm file size, divided by bytes file size
abc_ratio: asm file compression rate, divided by bytes file compression rate
ab2abc_ratio: ab_ratio, divided by abc_ratio

Note that these features only contain this much information within the closed set of 9 malware families. A lot of other (not malware) files types could share these properties.

Asm files contents

From the asm files, we extracted features by looking at the structure and meaning of specific elements. We also tried a more brute force approach (character or word ngrams) but this seems to work worse. We used the following features on each file:

Proportion of lines or characters in each ‘section’, that is recognized as the first word (before the colon) on each line, like: HEADER, .data, .rdata etc.
Number of occurrences of specific dll’s, recognized by the .dll extension in the text
Number of occurrences of specific opcodes, recognized by a pattern of spaces and lowercase a-z characters
Proportion of certain interpunction characters in each of the sections:
' ','?',',',',',':',';','+','-','=','[','(','_','*','!','\\','/','\"

Note that rare features were excluded based on the number of train files in which they occur, we required a minimal number of 5 files with sections, 30 with dlls and 30 with opcodes.

Finally, for one model, we reduced the set of asm features by keeping only features that are nonzero in at least 500 files. For that model, we also calculated statistics (number of unique, total number, etc.) on sections, dlls, opcodes and function calls.

Bytes files contents

From the bytes files, we extracted (1,2 and 4) byte ngrams and the distribution of entropy over the file. For the latter, we compressed each 4 kB block of bytes, and calculated percentiles on the distribution of those blocks, together with averages over the whole file and four quarters of the file. We have also generated full-line bytes as 1-gram, picking 40K most popular present from a sample in the training and test files.

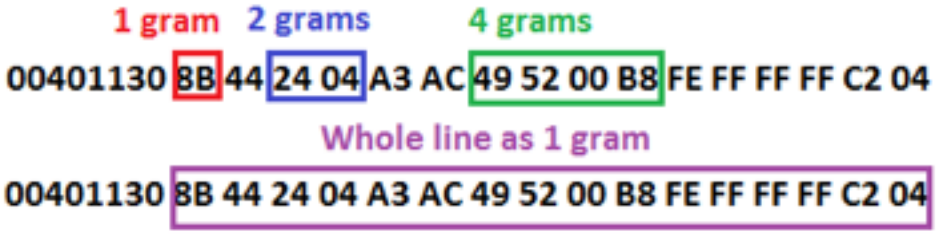

Figure 1: Different Byte's considered as ngrams, sample line from 0A32eTdBKayjCWhZqDOQ.bytes

3. Modeling Techniques and Training

We have generated many different models from different combinations of the initial datasets we created (that invloved file properties, asm characteristics and Bytes' ngrams). Most of them have been modelled using Xgboost[1] (Gradient Boosting Trees) with 80% contribution and scikit's ExtraTreesClassifier[2] with 20% contribution respectively. All these models generated predictions for hold-out sets of the train data to be used for meta-modelling.

Part of the modeling was done mainly to evaluate new sets of features, we call this part the 'untuned modeling'. It is a mix of 2/3 GradientBoostingClassifier[3] with 400 trees and 1/3 ExtraTreesClassifier with 400 trees, both from python sklearn. This model is untuned in the sense that it was kept the same throughout the competition. Predictions from the untuned model on the final feature set (45c) have been used as features in the meta bagging model.

We also used a meta-bagging model that takes the outputs' predictions of the single models (and their respective hold-out predictions) and combines them via another combination of models that consists of xgboost (60%) and ExtraTreesClassifier(40%). Many (10) such models were generated with different seeds and shuffling and we averaged their prediction, which also constitutes our final submission. The following table illustrates the data, model specifications and performance of all our models.

---

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

[2] http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html

[3] http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html

Model Name	Data	Model	Pr. LogLoss
fullline	whole-line Bytes + 1gram Bytes	80% Xgboost + 20% Extra	0.02851
2gram	2 gram Bytes + 1gram Bytes	100% Xgboost	0.01681
Gert202xtra115	asm fetaures and files' properties	80% Xgboost + 20% Extra	0.01423
pred_45	asm fetaures and files' properties	66.66% GBM+ 33.33% Extra	0.00762
4gram	4 gram Bytes + 1gram Bytes	80% Xgboost + 20% Extra	0.00702
Gert282xtra115k	more asm fetaures and files' properties	80% Xgboost + 20% Extra	0.00681
Gert282_258extra115k	asm fetaures and files' properties + 1 gram bytes	80% Xgboost + 20% Extra	0.00597
meta-model	fullline + 2gram + 4gram + pred_45 +…ALL	60% Xgboost + 40% Extra	0.00297

Figure 2: Different models' performance

4. Code Description

_fast_feature_extraction.py extracts three types of features from all of the train and test files: file properties, features from the asm files and features from the bytes files (except ngrams). Results are written line by line while the functions go through all the files. Finally, files are combined to facilitate training models on multiple feature sets together.

_untuned_modeling.py was used to evaluate new sets of features during the competition. It does cross validation on a (combined) set of features to test whether it improves the score compared to earlier feature sets, always using the same model.

tfidf1gram.py, tfidf2grams.py and tfidf4grams.py scan a sample of train and test files for n-gram bytes and then scans for a second time both files find out how many times each byte-gram appear (aka count frequency) in each file and writes it to oldNgramtrain.csv and oldNgramtest.csv (where N= the number of bytes next to tfidfN) . In the case of 4grams only the 40,000 most popular bytes are used. Note the idf part is not really used but was meant to initially.

fulllinegram.py scans the train and test files for whole-line Bytes and finds the 40,000 most popular ones from a sample and then scans for a second time both files find out how many times each of these whole-line-byte-grams appear in each file and writes it to fullinebytetrain.csv and fullinebytetest.csv

single_20.py, single_28.py, single_282_and_1gram.py, fullline.py, 2gram.py, 4gram.py, all have the same structure and are single models trained on different datasets as explained above . They consist of a 5-fold cross validation where its holdout predictions are being saved for meta-modelling as well as we make predictions for the test set.

5. Dependencies

Python 2.7.9, including the following libraries:

os
csv
zipfile
re
numpy, scipy
io: BytesIO
collections: defaultdict
sklearn: .cross_validation, ensemble, metrics
Xgboost

6. How To Generate the Solution (aka README file)

1. Have trainLabels.csv and sampleSubmission.csv in the execution directory, with subdirectories train and test containing the bytes and asm files. Make sure to provide the right path to xgboost.
2. run _fast_feature_extraction.py
3. run tfidf1gram.py
4. run tfidf2grams.py
5. run tfidf4grams.py
6. run fulllinegram.py
7. run single_20.py,
8. run single_28.py,
9. run single_282_and_1gram.py,
10. run fullline.py,
11. run 2gram.py,
12. run 4gram.py,
13. run _untuned_modeling.py
14. run metastacking_scipt.py

7. Additional Comments and Observations

Some interesting insights we got during the competition:
- linear models perform a lot worse than tree-based models that allow interaction
- using ngrams (more than one word) from the asm file does not seem to improve things
- sections in the asm files are very important to distinguish between malware types
- meta features in the asm files (reflected by interpunction) help the model a lot
- a quite acceptable accuracy can be obtained from different feature sets, but there is a limit that cannot be crossed by combining them
- while reducing the number of features improves the Gradient Boosting model from Python sklearn (also when we tune it using a grid search), xgboost is able to achieve a significantly better score using the whole unreduced feature set.

8. Simple Features and Methods

Using the untuned model, we calculated the performance of various subsets of features, see table below.

Feature set	CV performance (regular logloss on both positive and negative class)	public LB	private LB
File properties	0.0682	0.0597	0.0599
Asm files contents	0.0146	0.0145	0.0121
Asm file statistics	0.0348	0.0304	0.0344
4 kB compressed size distribution	0.0480	0.0405	0.0449

9. Figures

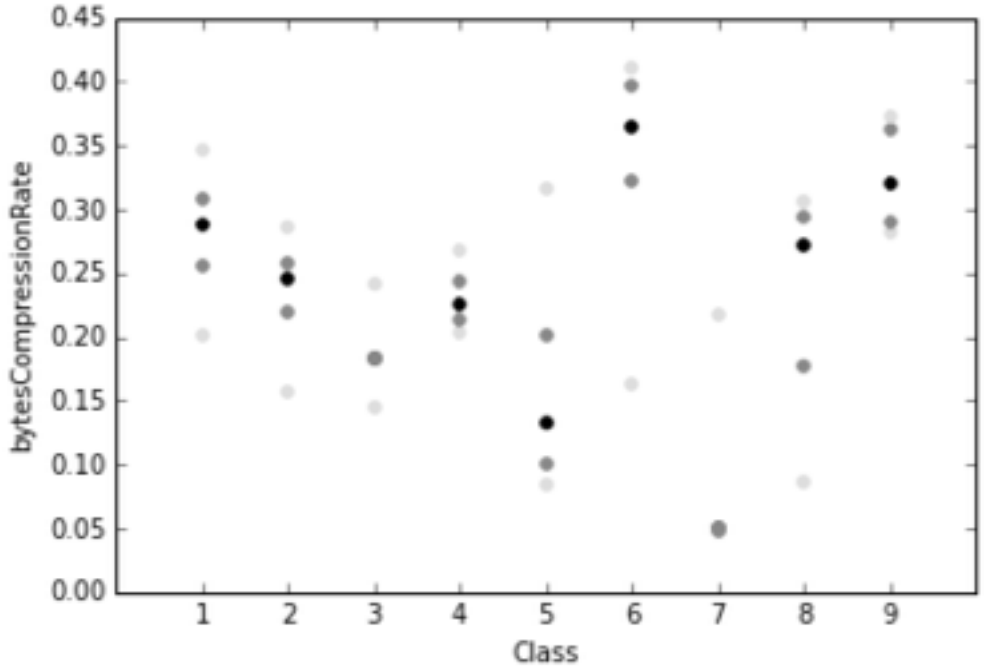

Figure 3. This plot shows that, as an example, the file property ‘compression rate of bytes file’ is quite strongly related to class membership (dots show 5%, 25%, 50%, 75% and 95% percentiles).

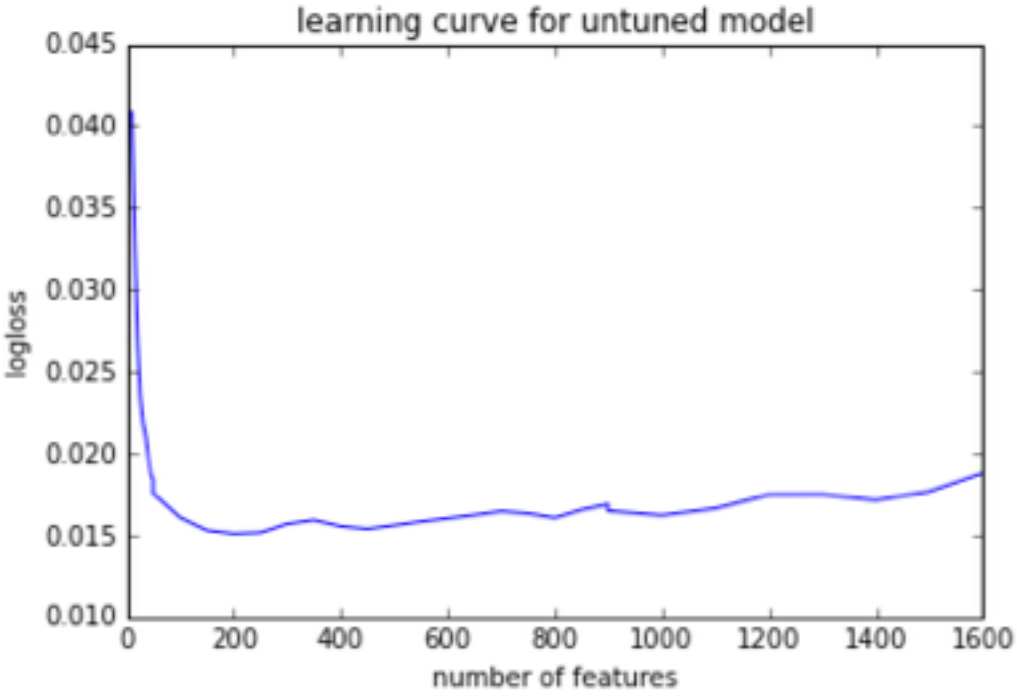

Figure 4. shows the learning curve for adding features to the Gradient Boosting (Python sklearn) part of the ‘untuned’ model, using only the features on asm contents. Optimal performance is attained with about 200 features - selected using feature importances. In contrast, the performance of xgboost is clearly better when the full 1600 (unreduced) asm features are used.