Practical 2: Classifying Malicious Software

Bolei Deng, Matheus C. Fernandes, and Nathan Wolfe
boleideng@g.harvard.edu, fernandes@g.harvard.edu, nwolfe@college.harvard.edu

March 10, 2017

1 Technical Approach

When directly using the naive skeleton file provided for this assignment we noticed that most of the 31 features are actually zeros and we only have 2 useful features. Then, by doing feature engineering, we were able to easily boost our features number by only considering the tags counts for all tags in the training set and in some instances even more features by considering the tag attributes. We began our numerical analysis by using Logistic Regression on the derived feature set and proceeded to try different classification methods, e.g., Neural Networks with RELU and Sigmoid as activation functions. Then by using Random Forest Method with an estimator number of 400, we are able to reach our highest accuracy on the public testing data.

In terms of feature engineering, our main approach was to produce integer features representing the count of a particular tag across the system log. Later, we added features integrating attributes of specific tags. We first tackled this practical by considering a given set of tag names that we thought was important for the distinction of the viruses. Through this first method, and by adding more tag categories and features we continuously saw performance improvement of our model. This led us to consider not only a select number of tags, but instead all tags in all of the training set. However, by comparing the tag names of the training set and the test set, we saw that there were more tags in the test set than there were in the training set.

Furthermore, we also noticed that there were tags in the training set that were not included in the test set, which served no purpose in predicting the test set. This led us to believe, that in order to improve our predictions, we had to consider perhaps the ordering of the tags but especially sub-features for each of the tags. Therefore, for select tags such as those that are both included in the training and test set, we decided to extract their sub-features, including further information on the execution of a particular command on Windows. We began this approach by first looking at a small set of tags' sub-features which we thought were important. Depending on how much including that sub-set of features improved our predictions through cross-validation, we either used that sub-set or not. Through a select few tags, we picked those that improved our predictions and included those as features totaling around $\sim$4000 to predict the test set. To obtain this process, we subdivided our feature generation function into various levels of of complexity ranging from level 1 to level 4 as subsequently described in the next section.

1.1 Feature generation levels

1.1.1 Level 1

This level is the most basic, simply generating a feature with the count of each system call tag. This level works well, except for the fact that some of the tags are not used in the test set and also the test set contains tags that are not used in the training set.

1.1.2 Level 2

This level adds features representing the attributes of all load_dll tags. To maximize the information for our model, each attribute and attribute value is encoded as a distinct count feature. This level increases our number of features from 133 features to over 4000. However, since this is a more sparse distribution of features, we must not only count it directly but proportionally introduce a weight factor such that the code is able to take it into consideration as a feature.

1.1.3 Level 3

This level adds 5 more tags for whose attributes we generate features. Here much like level 4, we must consider that the feature distribution of the added features become even more sparse. Because of that, we need to increase the weight factors proportional to the tags we think have more of an influence. Also, we make sure not to increase the weight too much in order to avoid saturating the original 133 features from the tag names.

1.1.4 Level 4

This level adds another 8 tags for whose attributes we generate features. The methodology in this level is the same as the previous levels only by factoring in even more tag attributes.

2 Results

In this section, we present the performance of our trials on Kaggles (public and private) as illustrated in Table 1. First, we use the 31 features extracted directly by the default code classification-starter.py without any modification and use Logistic Regression to classify the test data, which gives us only 44% accuracy. Then, by doing feature engineering, we boost our features number to 133, and Logistic Regression under this circumstance gives accuracy of 65%. Armed by these features, we proceed to try different classification methods, e.g., Neural Networks with RELU and Sigmoid as activation functions, producing accuracy of 73% and 80% respectively. Then by using Random Forest Method with estimators number 400, we are able to reach accuracy of 82.8% on public testing data.

After that, we decide to focus on Random Forest Method and taking even more features and sub-features into consideration. In Table 1 we present our results with different levels of features, i.e., 133 as level 1, 4848 as level 2, 20757 as level 3 and 26952 as level 4. Unfortunately, this boost in features number does not result in better performance on Kaggle. The redundance of features seems to result in noise to our model. Therefore, we try to explore another direction, i.e., decreasing the number of features. We found that not every feature in training files is also found

Model	Number of features	Perf. (public)	Perf. (private)
LOGISTIC REGRESSION	31	0.43579	0.44627
LOGISTIC REGRESSION	133	0.64895	0.65570
NEURAL NETWORK with relu	133	0.73579	0.72533
NEURAL NETWORK with sigmoid	133	0.80211	0.78399
RANDOM FOREST	133	0.82789	0.80208
RANDOM FOREST	4848	0.82158	0.79057
RANDOM FOREST	20757	0.81579	0.78454
RANDOM FOREST	26952	0.81263	0.78509
RANDOM FOREST	130	0.82947	0.80318
RANDOM FOREST	122	0.82895	0.80702
RANDOM FOREST	117	0.83053	0.80428

Table 1: Performance of different algorithms with different number of features.

in testing files. That means we are training on some features that never contribute to predictions. These features are not helpful and maybe even harmful to model performance. To deal with this, we find out of our 133 features which ones never appear in testing files (3 are found), and delete them before we train the model. By doing this we are able to increase the accuracy by 0.0015%. We then consider the two more cases where we delete the features that only appears 1 time and less than 5 times from 133 total features, result in 122 and 117 features. We can further increase our prediction accuracy by another 0.0020%, to 83.05%.

3 Discussion

Table 1 shows our level of success with different kinds of models and numbers of features. As described in Section 1, our set of features initially consisted of just counts of system call tags; these numbered up to 133, the so-called "level 1." The higher numbers of features in later trials resulted from adding attributes of certain tags as features.

The first thing we notice from our results is the difference in performance between the two neural network trials, one with the perceptron activation function and one using the sigmoid function. The significantly better performance using the sigmoid function demonstrates how important activation function selection is to neural network performance.

Looking at the results of the different models overall, we see significant success using random forest regression; random forest continues to prove to be a versatile and effective model, as we saw with the high performance of the random forest baseline in Practical 1.

After our success switching to the random forest model with our "level 1" of feature generation, we decided to add more features in an attempt to add more relevant information to the model. Incorporating specific tags' attributes as features significantly increased the number of features, as can be seen in some of the random forest trials in Table 1.

However, these features proved not to contain much relevant information, as their addition did not improve the performance of the random forest model. Adding more and more tag attribute features, going up to a total of 26952 features, did not help. In fact, each additional level of feature generation slightly decreased our performance, suggesting that the new features were adding variance to the data, thereby hindering the performance of the model by causing overfitting to the irrelevant information.

After this experience with the tag attribute features, we realized the benefit in concentrating our number of features to those with the most information, to cut out as much variance as possible. Thus, we removed features for tags that appeared in the training set but not in the test set. We also removed features for tags with very low frequency, ending up with our highest performance at 122 features. This result demonstrates the benefit of relatively few yet information-rich features.

Github Repository

`https://github.com/matheuscfernandes/cs181-s17-practicals/tree/master/p2`