CS 181 Practical 2: Classifying Malware

Peer Group 15: No necesita memorizar formulas

Jonne Saleva, Rohan Thavarajah, Anthony Soroka

1) Introduction

The purpose of this practical was to identify the presence of malicious software in computer systems. This presence was to be detected by analyzing logs of computer activity in XML format. In particular, the task is to learn to classify a particular XMl document as representing a particular class of malware, or alternatively designate it as clean.

In addition to the `clean` class, there are 14 types of malware found in the data set:

Agent	Magania
AutoRun	Poison
FraudLoad	Swizzor
FraudPack	Tdss
Hupigon	VB
Krap	Virut
Lipler	ZBot

We obtain a near-80% classification accuracy on this task, using a logistic regression baseline and additional ensemble methods, random forests and xgboost.

2) Technical approach

We hypothesize that malware classes are characterized by a fingerprint-like set of system calls. We flesh our reasoning out in the discussion section but in brief, our technical approach is guided by searching for a model best suited to identifying these fingerprints.

Logistic Regression

First, to establish a sound baseline, we run a multiclass logistic regression using the unigram features along with the standard features, and select out L2 regularization parameter via 5-fold cross validation.

When regularization is very strong (low c), we attain a biased model with low accuracy but similar performance on the training vs validation sets. When regularization is weak (high c), model variance becomes an issue and we begin to overfit the data. We observe that accuracy on the validation set is compromised (though we had hoped it would look more parabolic in shape!).

However, a logistic regression on unigrams is unable to capture how a group of system calls interact to behave maliciously. We have two options: (1) we can explicitly generate features

to check for the simultaneous presence of different types of system calls (2) we can turn from logistic regression to random forests, a model that is suited to generating these interactions automatically.

Random Forests

With this in mind, we turn to random forests as a method. One of the clear advantages of using random forests over explicit feature generation is that RF takes care of some feature engineering for us. In particular, we do not know how many system calls interact to characterize malware. Indeed, the number of system calls required may differ by malware category (eg. AutoRun involves `{Order A, Order B}` but Swizzor involves `{Order A, Order B and Order C}`).

Therefore we opt for the second approach and abandon logistic regression in favor of random forests. When tuning the random forest classifier we run through a number of options. A random forest is an ensemble of multiple trees and we experiment with the maximum depth of each one in the range `[1,5,10,20,50,100]`.

As with very strong regularization for logistic regression, a random forest with low max depth is likely to be biased manifesting in low accuracy. Unlike logistic regression, when we ramp up max depth, although each tree is likely to be vulnerable to high variance, by averaging many such trees, we mitigate their individual variance. We opt for a `max depth = 20`.

We experiment with other random forest parameters, including the number of trees `[1,5,10,20,50,100]`, using `entropy` vs `gini` as the splitting criterion and the number of features used when looking for the best split `[n_features, sqrt, log2]`. We use 20 trees, `gini` and `n_features` finding that other configurations have a marginal impact on validation performance (omitted for conciseness). The parameter we thought would be most promising was "`class_weight`".

We observe that the class distribution is highly asymmetric. The "not malware" class appears over 50% of the time whereas "Poison" appears <1%. We experiment with the balanced mode which weights classes according to their frequency. This however caused our validation accuracy to dip from 0.89 to 0.88.

Given that when our performance is judged, correctly classifying the most frequent class is given the same weight as correctly classifying a rare one we did not explore further. However, if for instance identifying a particularly pernicious form of malware is of interest, this parameter would warrant further investigation.

Finally, we consider counting pairs of adjacent calls (bigrams). We juxtapose performance with bigrams and performance with interactions. We hypothesize that interactions are better suited to malware classification as order is probably unnecessarily constraining (it doesn't so much matter if order A immediately follows order B. What matters is if order A and order B occur anywhere in the same file). Indeed, we yield better results on the kaggle test set with interactions (0.794 vs 0.784) and favor them over bigrams in the final model.

Extreme Gradient Boosting (xgboost)

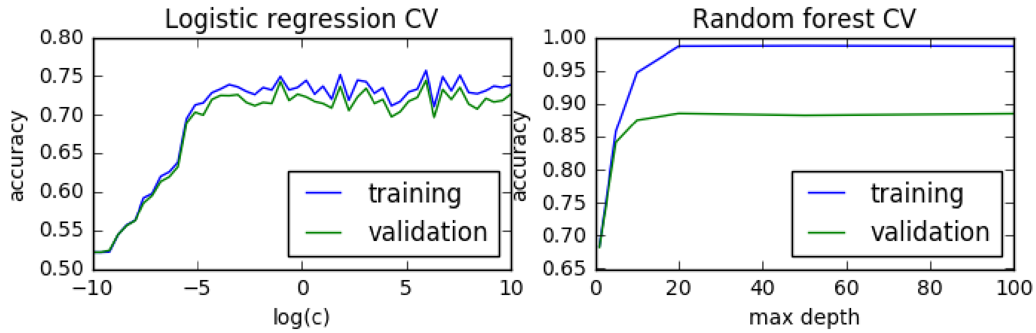

Figure 1: 5-fold cross-validation training vs validation accuracy for parameter tuning

Finally, inspired by our results using an ensemble method, we turn to another famous learning algorithm – `xgboost`. Our motivation for using this algorithm comes from its ability to both (i.) reduce variance without increasing bias too much, and (ii.) since the algorithm automatically re-weights misclassified points – a trait certainly desirable for us, given that our data set contains very rarely appearing data.

For tuning, we simply opt to tune the number of boosted trees, and learning rate:

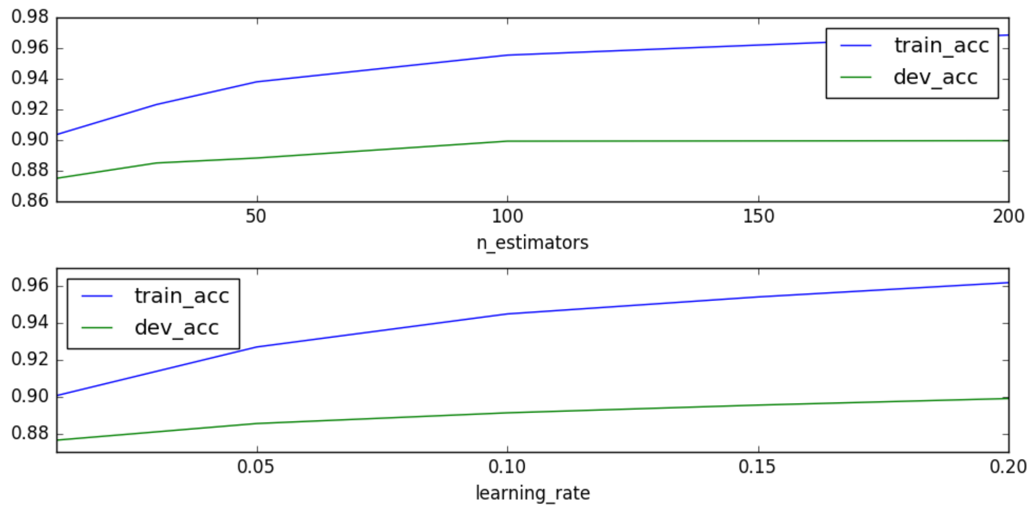

Figure 2: xgboost. (a) Accuracy vs. learning rate (b) Accuracy vs. no. of parameters

Overall, xgboost does not improve a whole lot with tuning on the dev set – in hindsight `max_depth` would have been a good parameter to examine more. Perhaps the unresponsiveness to tuning also results from our accuracy already being so high!

3) Results

Happily, both Random Forest and XGBoost outperformed the Bigram Baseline Model, as can be seen below:

Model	Leaderboard score
Random Forest (enhanced features)	0.794
xgboost (enhanced)	0.788
Bigram baseline	0.781
Logistic Regression (enhanced)	0.722
Logistic Regression (given features)	0.601
Most Frequent Class Baseline	0.414

4) Discussion

We begin by working with the three features given to us i) `first_call_feat` (a one-hot encoded vector of the first system call), ii) `last_call_feat` (a one-hot encoded vector of the last system call), and iii) `num_system_call` (count of system calls). For a baseline model, we apply logistic regression with 5-fold cross validation to tune the L2 regularization parameter. Using this simple feature space and logistic regression achieves 57% test accuracy.

We next explore more sophisticated feature extraction. First, after reviewing the Global Feature Dictionary, it becomes evident that the `first_call_feat` is adding no value as the first call is always "load image" in the training data. We amend this feature to capture the second system call instead. Additionally, besides knowing the total number of system calls, having the counts of each specific system call should prove valuable as well (the unigrams discussed in the technical approach).

With our improved feature space, we reapply Logistic Regression. This leads to an improved test accuracy of 72.2%. At this point we start to consider investigating alternative classification methodologies.

Given that logistic regression is linear, and we expect non-linearity in our basis we consider non-linear classifiers. Hence we next apply Random Forest - a decision tree-based classifier. For the reasons discussed in our technical approach, Random Forest lives up to our expectations, and improves our test accuracy to nearly 80%, surpassing the Bigram Baseline.

Seeing the significant improvement the classification technique can lead to, we investigate another non-linear classifier: XGBoost. The algorithm fits a new model to the residuals of the previous one, re-weighting the points on which there was high error. After tuning, XGBoost performed similarly to Random Forest, surpassing the Bigram Baseline and achieving a test accuracy of nearly 79%.

With more time we would like to investigate enhancing our feature space. Specifically, we considered using a deque data structure to record the first and last n features rather (for example recording the first 5 and last 5 features) and localizing system calls by thread. Finally, we would like to apply deep learning to this problem given their noteworthy ability to detect patterns from complicated and/or imprecise data similar to this problem.