Practical 2: Classifying Malicious Software

Yi Ding, Linying Zhang, Danny Zhuang
Kaggle: LYD

March 9, 2017

1 Technical Approach

- **Feature Engineering**

  First, we extract 3 sets of features for a total of 133 features, where n is the number of training points in the "train" directory (n = 3086):

  - **30 features**: A 0/1 matrix of n rows x 30 columns indicating the **first and last function calls of the executable with 1 and all other values 0**.
  - **1 feature**: A n-length vector denoting the **total number of function calls made by the executable**.
  - **102 features**: A matrix of n rows x 102 columns denoting the **counts of each function made by the executable** - similar to "bag-of-words" idea

  We store these features into a sparse matrix to minimize our memory usage for our features which have a large number of 0's. This is important when we don't know how large our data may get. We didn't have enough features in the end to make using sparse matrices absolutely necessary, but we have the capability to scale our feature set if we so chose.

  The CSR sparse matrix implementation describes the matrix **M** by 3 one-dimensional arrays (**A**, **IA**, **JA**)

  - **A** holds all the nonzero entries in **M**
  - **IA** holds the index of the first nonzero element in each row of **M**, plus a final term denoting the number of nonzero entries in **M**
  - **JA** holds the column index in **M** of each element in **A**

- **Model Selection**

  We explored classification models such as logistic regression, LDA/QDA, Decision Trees, Random Forests, and SVMs. We also explored priors and balancing the weights of our training data to account for the class imbalances, as some forms of malware were rare.

  Our results after feature engineering and weighted/unweighted classes are as follows (BOW = Bag-of-Words):

Model	Before BOW weighted	After BOW weighted	After BOW unweighted
LOGISTIC REGRESSION L1	0.471	0.800	0.839
LOGISTIC REGRESSION L2	0.469	0.659	0.733
LDA	0.596	0.780	0.780
QDA	0.077	0.796	0.796
DECISION TREE	0.779	0.844	0.857
RANDOM FOREST	0.781	0.873	0.887
LINEAR SVM	0.546	0.702	0.796

Table 1: Mean accuracy of models before and after including the Bag-Of-Words (BOW) features, for weighted and unweighted models.

We further explore the logistic regression models and random forest models through k-fold cross validation because they had the best performances in our default models.

- We explore the logistic regression because of its regularization capabilities to prevent over-fitting. C penalizes large coefficients per the loss functions below:

$$LogisticRegression(\ell_1)Regularization \qquad \min_{w} \ \mathcal{L}(\boldsymbol{w}) + \frac{1}{C}||\boldsymbol{w}||_1$$

$$LogisticRegression(\ell_2)Regularization \qquad \min_{w} \ \mathcal{L}(\boldsymbol{w}) + \frac{1}{C}||\boldsymbol{w}||_2^2$$

- We explore the random forest because it has low bias and low variance.

2 Results

All seven models were trained/tested before and after custom feature engineering using both weighted and unweighed classes. The performance of these models were summarized in Table 1.

Overall, random forest had the best mean accuracy compared to the other methods under the same condition. With our custom features added, accuracies improved for all models: the logistic regression L1 regularization increased from 0.47 to 0.80, and random forest increased from 0.78 to 0.87. The increase in mean accuracy after our custom "bag-of-words" features indicates that these additional features had substantial predictive power for identifying malware.

Additionally, we find that using weighted classes to fit our training models almost uniformly reduce our models' testing accuracy. This suggests that the imbalances in the data that we trained our models on (70% of the "train" directory) was similar to the imbalances in the data that we tested our trained models on (remaining 30% of "train" directory).

We ultimately use the unweighted models because we don't have any prior belief that the imbalances present in our training data will be significantly different than the imbalances in our out-of-sample data. Thus, we think that it is lower bias and lower variance to use unweighted models in which we implicitly assume that the imbalances in the training data will be close to

those in the population and out-of-sample data.

- **Logistic Regressions**
  We tuned the penalty parameter C from 10^{-7} to 10^{8} for both L1 and L2 regularizations through 5-fold cross-validation. The best mean accuracy achieved was about the same as that from the default logistic regression models. We noticed that as C decreased (more regularization), all coefficients became zero and mean accuracy also became zero (Figure 1).

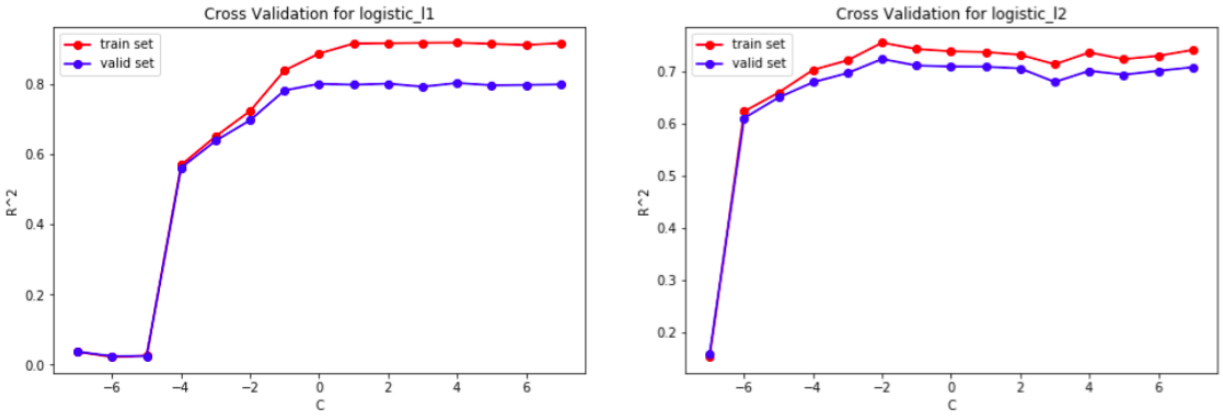

Figure 1: Logistic Regression CV accuracy from parameter tuning: L1 (left) and L2 (right)

- **Random Forest**
  We used a 5 fold cross validation to tune our random forest regression for three parameters: `n_estimators` the number of trees, `max_depth` the maximum depth of each tree, and `max_features` the number of features to be considered at each node.

  Because the number of trees in the random forest affect the variance (For number of trees B: if identical but not independent and pair correlation ρ is present, then the variance is: $\rho = \sigma^2 + \frac{1-\rho}{B}\sigma^2$ and the second term disappears as B increases) and not the bias of the model, we only tuned over two sufficiently large values to confirm they produced similar testing errors. We tuned `max_depth` up to 200 to be certain we were seeing a full range of model flexibilities (as we increase depth, bias goes lower). Lastly, we tuned `max_features` from 10 to 133 in order to test all values of features (total 133).

  We ultimately found that our random forest peaked at a CV-testing mean accuracy of 0.88. We applied the principle of Occam's Razor to choose the parameters of our final model: the smallest parameter values for the given testing mean accuracy of 0.87. These parameters were `n_estimators` = 110, `max_depth` = 100, `max_features` = 100.

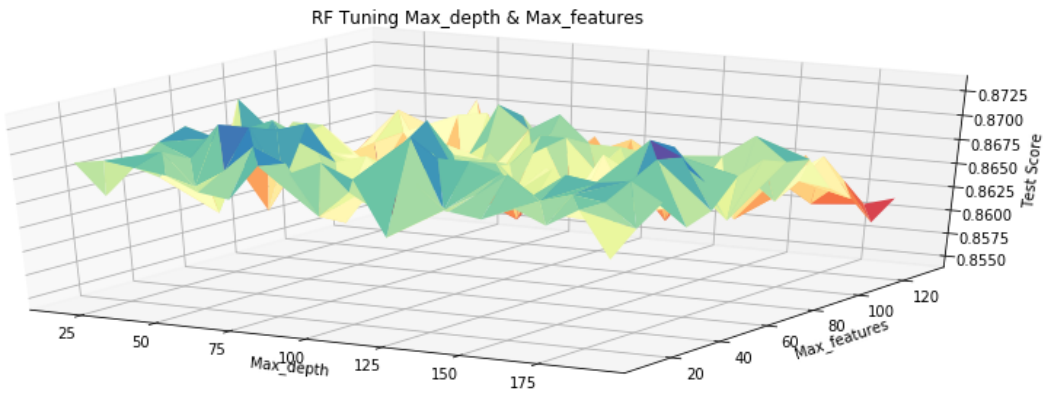

Figure 2: Random Forest tuning across all parameters

3 Discussion

We use a "bag-of-words" approach, whereby we built our own custom feature-function to count the number of each function call made by a given executable file. We combine our custom features with the features generated by the given feature-functions that extract the first and last function calls and the total number of function calls. We thought this approach would be helpful because it makes intuitive sense to us that different types of malware would call different functions in different frequencies.

With 133 features and 3,086 training points, we weren't too worried about over-fitting. Regardless, we still explored regularization. Furthermore, we tried to experiment with using balanced and unbalanced classes because of the class imbalances in our training data. Ultimately, we find that a tuned random forest had the best accuracy (0.88).

However, our approach has theoretical shortcomings. By relying on function call counts, we ignore the order of the function calls and efectively assume that the order in which functions are called does not matter. Having the first and last function calls as features slightly mitigates this issue of order, but by and large we ignore order.

4 Reference

[1] "How Does the Class weight Parameter in Scikit-learn Work?" Python How Does the Class weight Parameter in Scikit learn Work? Stack Overflow. N.p., n.d. Web. 09 Mar. 2017.
[2] Singer, By, Phillip. Handling Huge Matrices in Python. Philipp Singer. N.p., n.d. Web. 09 Mar. 2017.
[3] "Weights in Glm, Logistic Regression, Imbalanced Data." Cross Validated. N.p., n.d. Web. 09 Mar. 2017.