CS181 Practical 2

Team [J.Q.G]
Jeewon (Grace) Hwang, Qing Zhao, and Jing Wen

March 2016

1 Technical Approach

In this practical, we are interested in classifying malicious software. Around 3000 XML tag files from people's computers were collected. These files are used to train models to make future predictions on the class of malware that an executable belongs to. Since this is a classification problem, we will explore several different kinds of classification models.

1) Features Engineering

From last practical, we learned that feature engineering is the crucial to enhance the model performance, therefore, we also came up with several ways of generating new features.

Frequency of Different Tags

Malware in various classes may have distinct sets of commands or execute certain commands with different frequencies. Thus, we extracted all different tags appeared in the beginning of each line of XML files and added distinct tags (indicating a command) as new features. Then we counted the occurrence of each tag in a file, for example, the number of "vm write" and the number of "open file". As a result, we are able to identify 106 different tags from 3086 XML files, and we will use the count of these tags as the features for our models.

u'accept_socket' u'add_netjob', u'all_section', u'bind_socket',
u'trimmed_bytes', u'unload_driver', u'vm_allocate', u'vm_mapviewofsection',
...
u'vm_protect', u'vm_read', u'vm_write',u'write_value'

After we generated the 100 features by simply extracting tags, we were able to beat the baseline. However, simply counting the number of tags would not be enough to improve our models' performance further. Therefore we looked at the correlation among features to see the reason behind the limited performance.

The highest correlation is around 0.3 to -0.3, which is relatively small. This implies that features are not very correlated with each other, and hence we do not need to worry about co-linearity problem. Therefore we decide to generate more features to enhance our model performance. **(refer Figure 1)**

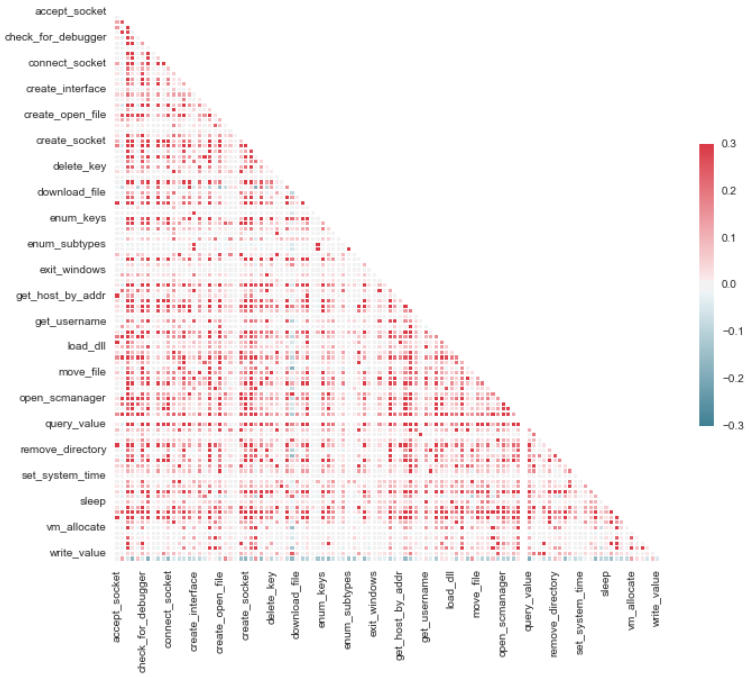

Figure 1: Pearson Correlation: last row is the response variable

New Features : Sequence of Tags
We thought the flow of XML file is imperative at classification since this can represent what program commands will do. Accordingly, we came up with an idea using 'Sequence of Tags', and figured out the way to mine the most frequently appeared sequence of tags (commands) and identify them as patterns.

To implement it, we parsed each XML file into a list of distinct-tags-flow and then turn them into **4-consecutive-tags pairs**. After applying this for each files, we combined all of them in our train set to select the most frequent tag-flow-patterns. Then we added these **100** tag-flow-patterns as our new features, and set the number of occurrence of this pattern among each file. Figure 2 shows the the top frequent tag-patterns based on their occurrence among the whole train set.

```
[['open_key query_value open_key query_value', 2142],
 ['query_value open_key query_value open_key', 1836],
 ['vm_protect vm_write vm_allocate vm_protect', 549],
 ['process thread load_image load_dll', 499],
 ['dump_line trimmed_bytes recv_socket dump_line', 445],
 ['recv_socket dump_line trimmed_bytes recv_socket', 434],
 ['trimmed_bytes recv_socket dump_line trimmed_bytes', 434],
 ['dump_line trimmed_bytes send_socket dump_line', 399],
 ['open_key enum_keys open_key query_value', 393],
 ['send_socket dump_line trimmed_bytes send_socket', 387],
 ['open_key query_value load_dll create_window', 380],
 ['trimmed_bytes send_socket dump_line trimmed_bytes', 370],
 ['load_dll open_key query_value open_key', 361],
 ['open_file open_key query_value open_file', 340],
 ['open_file open_key query_value open_key', 302],
 ['open_key query_value open_file find_file', 298],
 ['query_value open_file find_file open_key', 298],
 ['get_file_attributes read_value get_file_attributes read_value', 288],
 ['open_key query_value load_dll open_key', 288],
 ['read_value get_file_attributes read_value get_file_attributes', 284],
```

Figure 2: Most Frequent Patterns

2) Feature Reduction

PCA
So far, we have generated 106 + 100 features, whereas we only have 3086 observations. By including all features, we are likely to encounter over-fitting issues, as well as data sparsity problem. Therefore we concluded that we needed to reduce features.
We first decided to carry out PCA to reduce the number of features. Followings are the top 'explained_variance_ratio', which are the percentage of variance explained by corresponding component.

[3.91315276e-01, 2.63259786e-01, 1.45155855e-01,
9.41070436e-02, 4.22026597e-02, 1.50369668e-02,
1.35901802e-02, 1.08299298e-02, 9.02474844e-03,
4.79322777e-03, 3.51131656e-03, 1.46616478e-03,
1.30781114e-03, 1.09285478e-03, 8.78713852e-04,
4.77860237e-04, 4.44813602e-04, 3.01532820e-04]

Regularization
Instead of PCA, we looked for other method to avoid over-fitting caused by too many features. Here we believe that regularization will take care of over-fitting problem by reducing some elements in our weight vector that does not contribute much to the formation of decision boundary. The way how we selected options of regularization will be more discussed in the 'Modeling' and 'Discussion' part.

3) Model Selection

Grid Search Cross Validation
Similar to the first Practical, we divided the given training dataset additionally into training and validation datasets to address the over-fitting problem. Then, for each model, we apply k-fold (we usually tried 5-fold) cross-validation using the new training data. In addition, for models that require extra tuning, we iterated cross-validations through different values of parameters and select the parameters that gives the best score. For example, for Logistic Regression, we iterated different values of C, the inverse of regularization strength, i.e. "$alpha$" $= [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]$, and we then chose the alpha that gives us the smallest mean squared error. In addition, we also fix the **random state**, which allow us to compare models when different set of features are employed.

Thus, to compare different models (In this paper we mainly discussed Logistic, Generative Gaussian, Random Forest, SVM, and Gradient Tree Boosting), we looked at the accuracy score of the validation data-set.

Confusion-Matrix
Additionally, we also looked up the confusion-matrix since accuracy is not a reliable metric for the real performance of a classifier, because it will yield misleading results if the data set is unbalanced (that is, when the number of samples in different classes vary greatly). Indeed, our dataset has substantially more number of None-class files compared to the other class-files, which makes it crucial for us to examine the distribution False-negative and False-positive. We'll discuss in details later in 'Result' and 'Discussion' part.

4) Models

Logistic Regression
Logistic Regression is one of the most classical model to solve classification problem. It is a probabilistic discriminative model and performs non-linear transformation with **softmax** function. There are two approaches to solve the regression model: Newton-Raphson Algorithm and Stochastic Gradient Descent. Newton-Raphson Algorithm is an iterative method that updates parameter with the ratio of first to second gradient descent of likelihood. Stochastic Gradient Descent, on the other hand, updates parameters with the gradient descent of an error function, in this case, the negative likelihood, at certain learning rate.

Since the Logistic Regression from the sklearn package allows us to pick solvers, we train our model with both algorithms discussed above.

In addition, the function is also flexible enough to accommodate assumptions on loss function. We can either consider the classification of each label a binary problem: i.e. belong to a label or not, or assume that classes follow multinomial distribution and minimize the multinomial loss. When we make the second assumption, only Newton-Raphson Algorithm can handle the model fitting process.

Before we manually reduce any features, we place "L2" penalty on weights with a list of "C" values summarized in the table below.

Gradient Boosting Classifier

To address over-fitting issues, we tune the "learning_rate" of Gradient Boosting Classifier. By putting learning rate (shrinkage) relatively small, we are able to improve the model's generalization ability, thereby avoiding over-fitting. Moreover, by fixing "max_depth", we limit the minimum number of observations in tree's nodes, which helps to reduce model's variance. Gradient Boosting Classifier builds the model in a stage-wise fashion like other boosting methods do, and it generalizes them by allowing optimization of an arbitrary differentiable loss function.

2 Results

Feature Engineering

In addition to the 106 features generated from the tags of the XML files, we extracted extra 100 features by looking at the sequence of tags. Building Random Forest model with both set of features, we got a boost in both training and testing score as shown in Table 1.

Table 1: Accuracy Score Before / After Adding Flows of Tags

	Before (106 features)	After (206 features)
Train Score	0.92	0.98
Test Score	0.86	0.90

Model Selection

In this section, we display our results of cross-validation for different models: [Logistic Regression, Gaussian Generative Model, SVM, Gradient Boosting Classifier, Random Forest, KNN]. Among them, we'll only display the detailed result of 2 models that gave us the best result, which are the Random Forest and Gradient Boosting Classifier.

Logistic Regression

We tried 2 approaches to solve the regression model: Newton-Raphson Algorithm and Stochastic Gradient Descent.

Table 2: Logistic Regression (All Features)

	One v.s. the Rest				Multinomial	
	Stochastic Avg. Gradient		Newton		Newton	
C	mean	std	mean	std	mean	std
0.01	0.664	0.009	0.850	0.019	0.860	0.017
0.1	0.664	0.009	0.854	0.019	0.859	0.015
1.0	0.664	0.009	0.856	0.018	0.857	0.018
10.0	0.664	0.009	0.854	0.021	0.855	0.019
Best	C		0.01		1.0	
	Train Accuracy		0.92		0.91	
	Validate Accuracy		0.86		0.85	

Random Forest

Table 3: Random Forest Result

N_Estimator	Max_Depth	Mean	Std
20	30	0.88426	0.01486
50	30	0.88426	0.01180
100	30	0.88704	0.01172
20	50	0.88426	0.01486
50	50	0.88426	0.01180
100	50	0.88704	0.01172
Best	**(100, 30)**	**0.887037037037**	

Gradient Descent Boosting

Table 4: Gradient Descent Boosting

Learning Rate	Max Depth	Mean	Std
0.05	3	0.879	0.019
0.05	5	0.880	0.016
0.05	10	0.873	0.013
0.1	3	0.879	0.019
0.1	5	0.877	0.011
0.1	10	0.879	0.013
0.2	3	0.873	0.012
0.2	5	0.874	0.014
0.2	10	0.876	0.013
Best	**(0.05, 5)**	**0.880**	

Model Comparison

Table 5: Model Comparisons

Models	Training Accuracy Score	Testing Accuracy Score	Kaggle Score
Logistic	0.91	0.84	0.767
Gaussian Generative Model	-	-	0.652
SVM	0.93	0.86	0.798
Gradient Tree Boosting	0.98	0.90	0.81
Random Forest	0.99	0.91	0.80
kNN	0.99	0.87	0.75
Best	**Gradient Boosting Classifier**		0.81

3 Discussion

Feature Engineering

PCA

After training different classifiers with PCA, we learned that PCA did not provide better models. For example, when we train the Generative Gaussian Model using codes from homework2,the reduced feature decreased the Kaggle score significantly (The results are summarized in Table 2). We suspects the reason to be, as PCA chooses the directions in which the variables have the most spread, not the dimensions that have the most relative distances between clustered subclasses. Since all the classification algorithms generally tries to define a hyperplane to differentiate different classes of our objects in multi-dimensional space, it is susceptible to a range

Table 6: Generative Gaussian Model

	Full (106 features)	Reduced (20 features)
Kaggle Score	0.65158	0.46263

of artifacts in the data. Therefore we concluded not to use reduced-features, and stick to the original variables for all models.

Confusion matrix

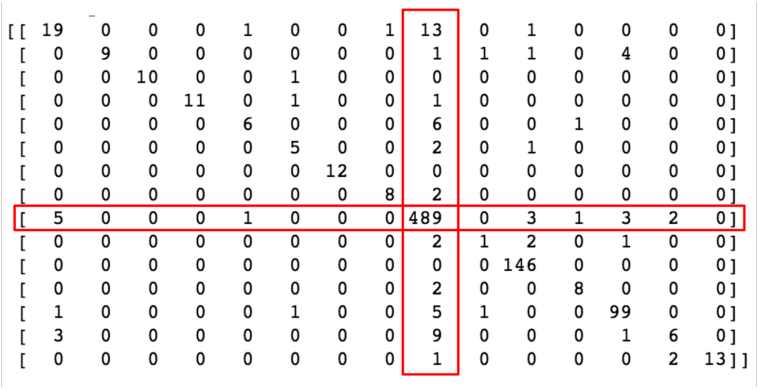

Figure 3: Confusion-Matrix: the red cells are for None-class

Figure 3 represents the confusion-matrix generated by the prediction results of a Random Forest model. As we described earlier, we also used confusion-matrix to evaluate other models. Across models, we learn that most of inaccuracy comes from the prediction of software that is good. For example, as shown in the figure 3, we notice that a lot of software in class "None" are classified into other malware classes.

In order to address this issue, we tune the "class_weight" parameter of Logistic Regression to place more weight on software in class "None." For example, we place the weight based on the frequency of a malware class appeared in the train dataset. However, we do not observe a better result. In addition, by doing so, we cannot generalize because we already learn some information about the data before training the model. However, this problem is definitely worth more exploration.

Models

Logistic Regression

As we observed from the table, Newton method always provides a better result than Stochastic Average Gradient. This is because Newton method takes consideration of second derivative of log-likelihood and hence includes more information. However, it take much longer to compute. In general, we observe that the method "One vs the Rest" provides a slightly higher accuracy result than the "Multinomial" method. This is because "One vs the Rest" is more flexible but at the same time more prone to overfit. Thus, we focus on the "Multinomial" method for the rest of analysis. However, we encounter over-fitting issues even we have add regularization. Therefore, we also explore other models.

Random Forest

Since Random Forest gave us good result in previous Practical, it was natural for us to try Random Forest as well in this Practical. We specifically control the number of trees in Random Forest (n_estimator) and the maximum depth (max_depth) of Decision Tree. We tuned those parameters by running cross-validation and we selected the best parameters based on both accuracy score itself and having minimum gap between train and test accuracy score.

Gradient Boosting Classifier

We noticed the over-fitting problem of Random Forest classifier. To address this problem, we introduced another

Ensemble models, Gradient Boosting classifier, which has nice properties in terms of regularization as discussed above in 'Technical Approaches' part. And judging from our result, it is safe to say that it is the best model that gave us the most accurate prediction as well as addressing the over-fitting problem.

SVM
As Finale emphasized during the class that SVM revolutionized the way people deal with classification. In addition, we noted that in our train data set, the classes were significantly unbalanced. For instance, None-classified files dominate the whole data set mainly. So we thought that there could be improvement if we deal with the balance. Therefore we chose SVM specifically since it implements a keyword **class_weight** in the **fit** method. The parameter was set as follows : a dictionary form class_label : value, where value is a floating point number > 0 that sets the parameter C of class class_label to C $\times value$.

KNN
Apparently, KNN did not give us good result. This is because, in the case of KNN, dimension reduction usually performed prior to applying the KNN algorithm in order to avoid the effects of the curse of dimensionality. However, since we did not adopt PCA, it is inappropriate to run KNN with such high number of features (100 + 106).

Potential Improvements

Improving Accuracy Score by Feature Engineering
Even though we spent quite a lot of time to do feature engineering, by making the sequence of 4-consecutive-tags, unfortunately, it did not give us huge improvement. Therefore, we think we should try other than this, such as exploiting number of system calls or consider different types of process, or even mining parameters inside every execution command.

Over-fitting Problem
Apparently, there are still plenty of rooms to deal with over-fitting problems, as suggested by the disparity among accuracy score of the train data, test data and the Kaggle score. One thing we regard as important was to deal with the unbalance between different types of classes. We wanted to use biased weight by setting the options in our models, rather than the original distribution of target values.