CS 181 - Machine Learning (Spring 2016)
Practical 2: Classifying Malicious Software

Team BridgingtheGAK

Gioia Dominedò (40966234) Amy Lee (60984077) Kendrick Lo (70984997)

March 4, 2016

For this project we used machine learning methods to identify classes of malware (or the lack of malware) in executable files. Our analysis was based on XML logs of the executables' execution histories, which we parsed in order to create features associated with particular malware classes.

At the time of submission, we had uploaded 18 different sets of predictions to Kaggle and achieved a best categorization accuracy of 84%, which put us in first place on the leaderboard. This is subject to change, pending release of the withheld final test data. Our best results were achieved using a tuned random forest classifier and features based on counts of sequences of system calls.

1 Technical Approach

The primary modeling techniques used were a **random forest classifier** (which generated our preferred Kaggle submissions) and a **linear support vector machine (SVM) classifier**. We also spent a considerable amount of time on feature engineering, including both simple counts and percentages of unique system calls, and more complex counts of sequences of system calls. Excerpts of the relevant Python code are included in Appendix E.

1.1 Validation

We carried out two main forms of validation. First, we first set aside a portion of the training data for use as a validation set (or "test set" in the context of our training set). This not only allowed us to check how well our models generalized before submitting them to Kaggle, but it also allowed us to compare performance across models on a consistent test-train split. We used a common 80-20 split for this purpose. We also performed five-fold cross-validation when tuning our models, to avoid the potential for over-fitting. For both forms of validation, we tested our model performance with a simple **categorization accuracy** measure, i.e. the percentage of correctly classified examples.

1.2 Exploratory Data Analysis

We started by examining the distribution of classes, in order to gain a better understanding of each label's frequency and to make sure that the distribution was consistent across both the training and validation sets. Figures 1 and 2 show that just over half of the executable files have no form of malware. Among the infected files, the most common malware classes are Swizzor (18%), VB (12%) and Agent (4%). All other malware classes account for less than 2% of the samples. While we did not correct for the imbalance in class labels at this early stage, we noted it as a potential area of later investigation.

1.3 Feature Engineering

Untuned model results for all the feature combinations that we attempted are included in Table 1. The two main approaches are described in more detail below.

1.3.1 Number and Percentage of System Calls

The sample code provided included two features based on the number of counts of two particular system calls. As a first step, we extended this code to count the number of all system calls. This resulted in a significant improvement in categorization accuracy on the validation set across model classes (e.g. 53% to 74% using untuned logistic regression, 75% to 88% using an untuned random forest classifier). The untuned random forest model performed slightly worse on the Kaggle dataset (80%), but still significantly better than both the baseline (39%) and the two-feature model (65%).

We then considered whether the percentage of calls might be a more useful metric than a simple count. We have all (unfortunately!) experienced malware in the past, and a common symptom is the way in which it takes over all system resources. We speculated that this might be visible in an unusual distribution of system call types, which would not necessarily be captured by the previous count measure. We also thought that this could be a way of normalizing for executables that are more or less active than others. Unfortunately, our intuition turned out to be incorrect, as this model performed roughly in line with the previous one.

1.3.2 Sequences of System Calls (N-Grams)

Malware often contains a package of exploits through which it might probe a system in search for vulnerabilities. Unusual behavior might therefore be detected by analyzing sequences of system calls. To achieve this, we encoded each of the 110 system calls as a two-digit hexadecimal string, and created sequences of hexadecimal strings representing all the system calls made by the executables. This encoding allowed us to preserve the order of system calls, but in a more compact form. We then processed the strings by dividing them into sequences of continuous characters ("n-grams") and counting how many times specific sequences appeared in each string. More precisely, we computed how many times patterns of length n were found in the hexadecimal strings for $n = 2, 4, 6,$ and 8, which resulted in nearly $200,000$ features.

1.4 Modeling Approaches

Compared to the previous practical, where feature engineering resulted in substantial improvements in model performance, we were disappointed to see that adding incremental features did not have as significant an effect in this case. In particular, we had expected to see a larger improvement in performance when adding features based on the sequences of system calls. We therefore decided to focus our attention on testing many different model classes, in order to determine whether this could be an alternative source of (large) performance improvements. All our model implementations were based on "one hot encoding" of the malware classes.

1.4.1 Majority Classifier

Given that about 52% of the training data did not contain malware, we first implemented a basic **majority classifier** as a baseline measure. This classifier - which simply predicts that all executable files do not contain malware - resulted in categorization accuracies of 52% on the training set, 53% on the validation set, and 39% on Kaggle. This highlighted a large discrepancy between the training and test data, with the test data showing a much smaller proportion of uninfected software compared to the training data.

1.4.2 Logistic Regression

We started our analysis of each new batch of features with **logistic regression**, but found that it consistently underperformed the other models. This indicates that the functional forms being generated by the classifier are not sufficiently expressive to capture the information encoded in the design matrix. The gap between logistic regression and other models only narrowed when using features based on sequences of system calls, as the sheer number of features was able to offset the restrictions imposed by the functional form.

1.4.3 Random Forest Classifier

Unlike logistic regression, **random forest classifiers** are a non-parametric model class. As with other ensemble learning methods, random forest models tend to do very well in model fitting where accuracy of fit (and predictive power) is prioritized over model interpretability. While decision trees typically over-fit the training set, random forest models reduce variance by training on subsets of both the training data and the design features and, consequently, usually generalize better to out-of-sample data. As we began to see them significantly outperform the other model classes, we focused on random forest classifiers in later stages of the project. Our parameter tuning is described in more detail below.

1.4.4 Extremely Randomized Trees Classifier (ExtraTrees)

Extremely randomized trees (ExtraTrees) are an extension of random forest classifiers. In addition to using a random subset of the features and training data, the ExtraTrees algorithm also randomly chooses a feature/split combination at each node of the tree. This reduces the likelihood decision trees getting stuck in local optima and can potentially lead to splits that would not otherwise be seen. For this particular dataset; however, the performance of the ExtraTrees classifier was roughly the same as that of the random forest classifier. This indicates that, on the whole, the

locally optimal decisions being made by the random forest classifier also represented globally optimal decisions.

1.4.5 Adaptive Boosting (AdaBoost) Classifier

As a last extension to decision trees, we also implemented an **AdaBoost classifier**. Over multiple iterations, AdaBoost creates a weighted sum of individual decision trees, with each new decision tree tweaked to focus on data points that were misclassified at the last step. We hoped that by directly targeting misclassified software, we would be able to improve the predictive power of our model. However, this turned out not to be the case – in fact, this classifier performed significantly worse than the random forest classifier in most cases. For this particular problem, the use of random subsets of the data and features turned out to be a more powerful tool than a direct focus on misclassified data points.

1.4.6 Linear Support Vector Machine (SVM) Classifier

The last model class we tried was a **multi-class linear SVM classifier**, as this model class has been reported to generalize better than other classifiers – particularly in high dimensional spaces. The multi-class training strategy that we used was one-vs-rest, which involved fitting one classifier per malware class (i.e. fitting a given class against all other classes). The advantage of this strategy over others, such as one-vs-one and crammer singer, is the improved computational efficiency and interpretability. As a whole, we found that the untuned one-vs-rest linear SVM's predictive performance was comparable to that of the untuned random forest classifier, but with the additional benefit of being much faster and lightweight to train due to the kernel trick.

While we focused our attention on the one-vs-rest strategy, largely due to its computational benefits, we hope to have additional time in future practicals to also experiment with a one-vs-one classification strategy. Models that use this approach are more time-consuming to train, due to the requirement of one classifier per pair of classes; however, they can be more effective in addressing imbalanced class distributions similar to the one in this problem.

1.5 Re-balancing Classes

As noted in our exploratory data analysis, some of the malware classes were relatively under-represented in the training data. In particular, the 'None' class accounted for over half of the executables. We were concerned that this might have a negative effect on our model results, so we experimented with various techniques to address the imbalance.

We first considered either **undersampling** the over-represented class (by randomly selecting a sub-set of records from the 'None' class and only including those in the training set) or **oversampling** the under-represented classes (by duplicating rare records and adding them to the training set). We chose to implement the latter technique as it does not introduce bias. Our expectation was that the previously under-represented classes would contribute more to the feature/split decisions when growing the random forest, and that this would lead to more accurate classifications. Unfortunately, our hoped-for improvement failed to materialize on the untuned random forest classifier, though it is possible that it could have improved the performance of one of our tuned models.

As an alternative, we also experimented with built-in weighting mechanisms in sklearn's random forest implementation. Once again, this failed to significantly improve the classification accuracy of our untuned random forest model.

1.6 Parameter Tuning

The relatively low number of features created by measuring the count and percentage of all system calls meant that we were able to tune five different model classes that used these features. The results in Table 2 show that the greatest improvements were obtained in the logistic regression and SVM models. The random forest and ExtraTrees classifiers did not improve much after parameter tuning, but still remained the best-performing models for these features.

Turning the parameters for the models based on the sequences of system calls was considerably most time-consuming and computationally intensive due to the very large number of features. The computational advantages of our linear SVM model were on full display at this point, as we were able to complete a full cross-validation grid search despite the very large number of features (see Table 3 for the results). However, even the best-performing SVM model continued to underperform our random forest classifier, so we focused our attention on the latter model despite the considerably longer runtimes[1]. Unsurprisingly, the overall results were consistent with the previous practical: increasing the number of trees resulted in higher accuracy and lower overfit, and limiting the number of features considered when choosing the best split resulted in lower variances as long as we did not choose too small a subset of features. The results for the runs that we were able to complete are included in Table 4.

2 Results

We had uploaded 18 sets of predictions to Kaggle at the time of submission. We selected the models to be tested on Kaggle based on their performance on the validation set that we carved out from the training data; however, we generated submissions based on models trained on the entire training set in order to make full use of the training data available. The best-performing model performed significantly better than the baseline (84% vs 39%) and achieved the top spot on the leaderboard, as of the time of submission. The performance of our models on the withheld data remains to be seen, and we expect to see variations in the final rankings due to the similar categorization accuracy of many of the top scores.

We selected our two best-performing tuned random forest models that used features based on the number of system call sequences as our final entries for the Kaggle competition. We note that the categorization accuracy on the validation set was consistently higher than on the Kaggle dataset. This may indicate the presence of one or more of the following: over-fitting to the training set; significant differences between the training and test sets (we noted this when testing our majority classifier); insufficient data to generalize well to out-of-sample data.

Table 5 summarizes the performance of the models that we built, including their categorization accuracy scores on the training and validation sets, and on Kaggle if submitted. Given the relatively large number of malware classes, we also found it useful to refer to **confusion matrices** and **misclassification reports** when assessing the results of our various models. These performance metrics allowed us to focus on the most common type of classification errors that our models

[1] Even when using a Google virtual machine, each random forest model took approximately two hours to run.

were generating. For example, we found that our earlier models were misclassifying a relatively large proportion of class zero ("Agent") malware as uninfected software. We have included these metrics for our best model in Figures 3 and 4.

3 Discussion

The discussion above details our thought process and the techniques that we employed through the project. Our main takeaways are listed below.

- **Asymmetrical classes:** We noticed early in the process that the training data was heavily asymmetrical, with just over half of the executables not being infected, and most malware classes accounting for just 1-2% of the data points. This led to the concern that it might be difficult to identify "common" traits of particular types of malware when only being able to examine relatively few instances of infected software. On a related note, we were also concerned that the features of the uninfected software might end up taking precedence when fitting models (e.g. when identifying feature/split pairs when growing a random forest). We experimented with techniques such as over-sampling in an attempt to correct for these possible pitfalls; however, they did not yield the performance improvements that we had anticipated.

- **Identifying features:** The disappointing effect of oversampling on model performance suggested that the imbalanced classes were probably less of a problem than we had originally thought, and that it was more likely that we were still missing features that would allow our models to distinguish between classes. The sheer amount of information encoded in the XML files made for a very open-ended feature creation process. We ultimately focused on sequences of system calls; however, it is possible that we were throwing away useful information by ignoring the parameters/metadata encoded in each of those calls. Had we had additional time, we would have attempted to also build features based on these additional parameters.

- **Dimensionality reduction:** The fact that seemingly endless features can be created from the XML files leads straight to the problem of dimensionality. Even by "just" using system call sequences, we quickly accumulated nearly $200,000$ features. This led to computational bottlenecks and very slow model tuning procedures. We plan to use dimensionality reduction techniques in feature practicals to avoid falling into this trap again.

- **Performance measures:** As a more general question, we also considered whether categorization accuracy is the best measure of performance for this problem. For example, there might be cases where there is a high cost associated with failing to detect malware, so we might want to focus on building models that minimize the false negative rate. We could also imagine that certain types of malware are more dangerous than others, so we might want to weight our performance measure more heavily toward the malware classes that we are especially concerned about, possibly even at the expense of misclassifying others.

A Figures

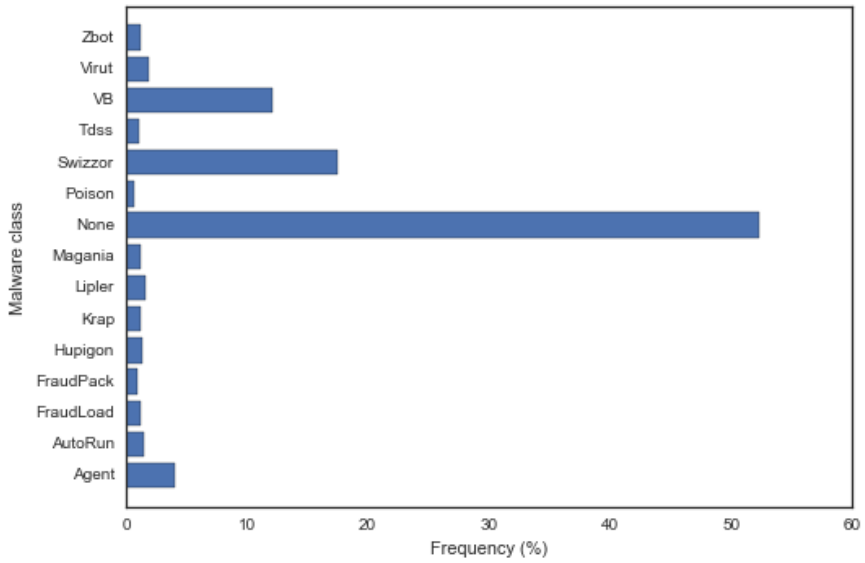

Figure 1: Distribution of Malware Classifications - Training Set

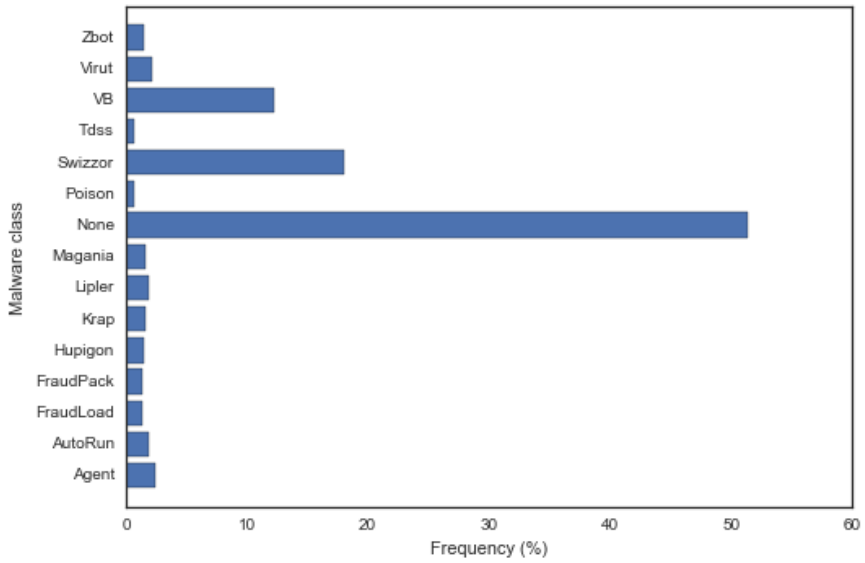

Figure 2: Distribution of Malware Classifications - Validation Set

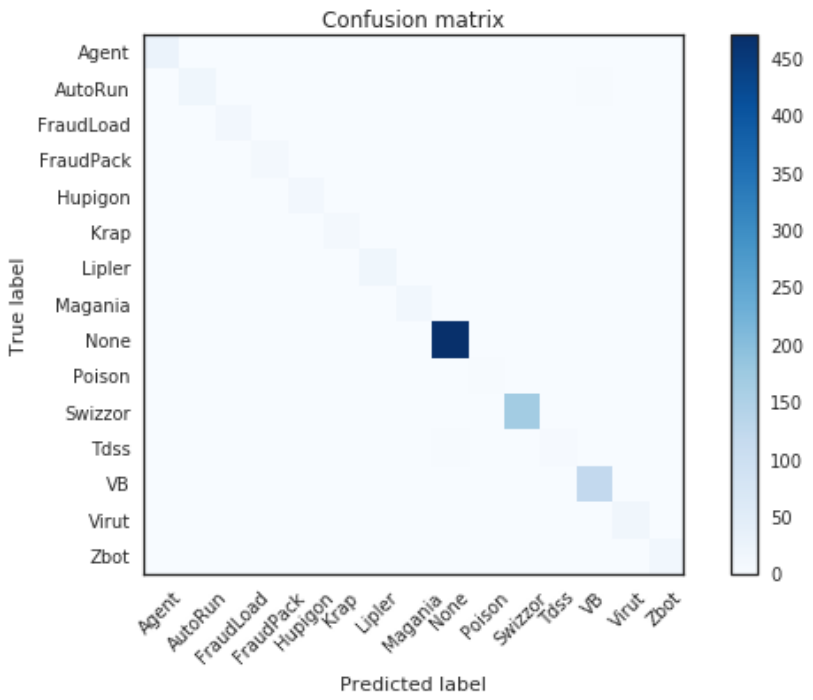

Figure 3: Confusion Matrix of Malware Classifications - Validation Set - Random Forest ngram 250 trees, 0.5 max features

	precision	recall	f1-score	support
Agent	1.00	1.00	1.00	29
AutoRun	1.00	0.90	0.95	20
FraudLoad	1.00	1.00	1.00	12
FraudPack	1.00	1.00	1.00	10
Hupigon	1.00	1.00	1.00	12
Krap	1.00	1.00	1.00	10
Lipler	1.00	1.00	1.00	17
Magania	1.00	1.00	1.00	14
None	0.99	1.00	0.99	473
Poison	1.00	0.67	0.80	3
Swizzor	1.00	1.00	1.00	168
Tdss	0.83	0.71	0.77	7
VB	0.98	1.00	0.99	122
Virut	1.00	1.00	1.00	15
Zbot	1.00	1.00	1.00	14
avg / total	0.99	0.99	0.99	926

Figure 4: Misclassification Report of Malware - Validation Set - Random Forest ngram 250 trees, 0.5 max features

B Feature Engineering

Table 1: Untuned Model Comparison

Features	Logistic Regression	Random Forest	Extra Trees	Ada-Boost	Linear SVM
Majority classifier	0.5258				
Given features (count of 2 system calls)	0.5356	0.7527	0.7527	0.4968	-
Count of all system calls	0.7441	0.8866	0.8758	0.4968	-
Percentage of all system calls	0.7268	0.8801	0.8801	0.7084	-
Count and percentage of all system calls	0.7495	0.8834	0.8790	0.5594	0.8326
Sequences of system calls (n-grams)	0.8715	0.8931	0.8952	-	0.8726

(Validation Set Categorization Accuracy)

C Model Tuning

Table 2: Model Tuning - Count and Percentage of all System Calls

Model Class	Untuned Accuracy	Tuned Accuracy	Best Parameters
Logistic regression	0.7495	0.8521	Penalty = L1 C=10000
Random forest	0.8834	0.8888	Number of trees = 250 Maximum features = log2 Minimum samples = 1
ExtraTrees	0.8790	0.8898	Number of trees = 150 Maximum features = 0.1 Minimum samples = 1
AdaBoost	0.5594	0.5594	Number of trees = 150 Learning rate = 1.0
SVM	0.7394	0.8326	Loss = squared hinge C = 0.01

(Validation Set Categorization Accuracy)

Table 3: SVM Model Tuning - Sequences of System Calls

Loss	C	Mean Accuracy
Squared hinge	0.01	0.8653
Squared hinge	0.1	0.8551
Squared hinge	1.0	0.8222
Squared hinge	10.0	0.7903
Squared hinge	100.0	0.7773
Hinge	0.01	0.8546
Hinge	0.1	0.8593
Hinge	1.0	0.8352
Hinge	10.0	0.7574
Hinge	100.0	0.7574

(Five-Fold Cross-Validation Set Categorization Accuracy)

Table 4: Random Forest Model Tuning - Sequences of System Calls

Number of Trees	Maximum Features	Minimum Samples	Train	Validation	Kaggle
100	0.6	1	0.9931	0.8942	0.8295
150	0.6	1	0.9930	0.8963	-
200	0.4	1	0.9931	0.8931	0.8316
200	0.5	1	0.9931	0.8996	0.8374
200	0.8	1	0.9931	0.8931	-
250	0.4	1	0.9931	0.8963	-
250	0.4	10	0.9139	0.8737	-
250	0.5	1	0.9931	0.9006	0.8358
250	0.6	1	0.9931	0.8974	-

(Validation Set Categorization Accuracy)

D Kaggle Submissions

Table 5: Kaggle Submission Comparison

Features	Model Class	Train	Validation	Kaggle
Baseline	Majority classifier	0.5203	0.5258	0.3900
Given features (count of 2 calls)	ExtraTrees (untuned)	0.7963	0.7495	0.6453
Count of all system calls	Random forest (untuned)	0.9819	0.8866	0.8016
Percentage of all system calls	Random forest (untuned)	0.9815	0.8801	0.8016
Count and percentage of all system calls	Random forest (untuned)	0.9810	0.8834	0.8026
Count and percentage of all system calls	Linear SVM (untuned)	0.8583	0.8326	0.7421
Count of system call sequences	Random forest (tuned)	0.9931	0.8942	0.8295
Count of system call sequences	Random forest (tuned - 100 trees, 0.6 max features)	0.9931	0.8942	0.8295
Count of system call sequences	Random forest (tuned - 200 trees, 0.5 max features)	0.9931	0.8996	0.8374
Count of system call sequences	Random forest (tuned - 200 trees, 0.4 max features)	0.9931	0.8931	0.8316
Count of system call sequences	Random forest (tuned - 250 trees, 0.5 max features)	0.9931	0.9006	0.8358

(Categorization Accuracy)

E Code Base

We have included below sample excerpts of code that was used to generate our features and models.

```
=========================================================================
CROSS VALIDATION & MODEL FITTING

def cv_optimize(model, parameters, train_x, train_y, n_folds=5, score_func=None):
   """
   Function
   --------
   cv_optimize

   Inputs
   ------
   model: an instance of a scikit-learn classifier
   parameters: a parameter grid dictionary that is passed to GridSearchCV
   train_x: a samples-features matrix in the scikit-learn style
   train_y: the response vectors of 1s and 0s (+ives and -ives)
   n_folds: the number of cross-validation folds (default 5)
   score_func: a score function we might want to pass (default python None)

   Returns
   -------
   The best estimator from the GridSearchCV, after the GridSearchCV has been used to fit the model.

   Notes
   -----
   see do_classify and the code below for an example of how this is used
   """

    if score_func is not None:
       gs = GridSearchCV(model, param_grid=parameters, cv=n_folds, scoring=score_func)
   else:
       gs = GridSearchCV(model, param_grid=parameters, cv=n_folds)
       gs.fit(train_x, train_y)

   print "BEST", gs.best_params_, gs.best_score_, gs.grid_scores_
   best = gs.best_estimator_

   return best



def fit_model(model, train_x, train_y, test_x=None, test_y=None, title=None, tracker=None):
    """
    Generic function that is used to fit a model and compute out-of-sample accuracy.

    Inputs
    ----------
    model: Any Scikit Learn predictive model (with or without custom parameters)
    train_x: Dataframe containing training set X-variables
    train_y: Dataframe containing training set Y-variables
    train_x: Dataframe containing test set X-variables
    train_y: Dataframe containing test set Y-variables

    title: title to use as key in tracker
    tracker: dictionary to store results
    """
    # fit model
    model.fit(train_x, train_y)
```

```
    # calculate training set accuracy
    train_acc = accuracy_score(train_y, model.predict(train_x))

    if tracker is not None:
        tracker[title] = [model, train_acc]

    # calculate test set accuracy
    if test_x is not None:
    test_acc = accuracy_score(test_y, model.predict(test_x))

    print '#############'
    print model
    print '----------'
    print 'Training set accuracy = %0.4f' % train_acc
    print 'Validation set accuracy = %0.4f' % test_acc
    print '----------'
    print '#############'

    if tracker:
        tracker[title].append(test_acc)

    else:
        print '#############'
        print model
        print '----------'
        print 'Training set accuracy = %0.4f' % train_acc
        print '----------'
        print '#############'

    # return fitted model and training/test accuracy
    return model

=====================================================================
FEATURE GENERATION


def call_feats_all_count_perc(tree):
   '''
   Returns the number of system specific calls made by the programs in count and percentage of total
   '''

    # keep track of calls
    call_counter = {}

    # loop through all calls/tags in the XML file
    for el in tree.iter():

        # extract the call/tag name
        call = el.tag

        # count the number of calls to each tag
        if call not in call_counter:
            call_counter[call] = 0
        else:
            call_counter[call] += 1

    # initialize the feature array (1 x 2D)
    call_feat_array = np.zeros(2 * num_calls)

    # loop through the calls we are looking for
    for i, call in enumerate(p2.CALL_SET):

        # update counter with the number of times the call was seen
        if call in call_counter:
```

```
            call_feat_array[i] = call_counter[call]
        else:
            call_feat_array[i] = 0

    # add percentages
    call_feat_array[num_calls:] = (call_feat_array[:num_calls] / call_feat_array[:num_calls].sum()).copy()

    # return feature array (1 x D)
    return call_feat_array

# generate features
X_train_all, Y_train_all, train_ids_all = p2.create_data_matrix(p2.call_feats_given, direc='/home/shared/practica
#  start_index=0, end_index=5

# split out randomly into train_test_split 70-20
X_train, X_valid, Y_train, Y_valid = train_test_split(X_train_all, Y_train_all, train_size=0.7, random_state=1004

X_test, Y_test, test_ids = p2.create_data_matrix(p2.call_feats_given, direc='/home/shared/practical2/data/test')

---------------------------

# OPCODE SEQUENCES

hexrep = {}
for i in xrange(len(opcodes)):
    if len(hex(i))==3:
        hexrep[opcodes[i]] = "0"+hex(i)[-1:]
    else:
        hexrep[opcodes[i]] = hex(i)[-2:]
hexrep["unknown"]="ff"

def create_call_string(tree):
    '''
    Generates sequence of hexadecimal digit pairs (e.g. "6a") for each
    system specific call and creates a string feature that encodes the ordering
    of all calls made for a given file. Returns the string.
    '''

    # holds string of hex pairs
    callstring = ""

    # ignore section headers
    ignore = ['processes', 'process', 'thread', 'all_section']

    # loop through all calls/tags in the XML file
    for el in tree.iter():

        # extract the call/tag name
        call = el.tag

        if call not in ignore:
            # append hex code to string
            if call not in hexrep:
                callstring += hexrep['unknown']
            else:
                callstring += hexrep[call]

    return callstring

...

codes2 = {}
codes4 = {}
codes6 = {}
codes8 = {}
```

```
for i in xrange(df.shape[0]):
    s = df.iloc[i, 0]
    assert (len(s) % 2) == 0

    for key in range(0, len(s), 2):
        if (key + 4) <= len(s):
            ss = s[key: key+4]
            if ss not in codes2:
                codes2[ss] = 1  # dummy value
        if (key + 8) <= len(s):
            ss = s[key: key+8]
            if ss not in codes4:
                codes4[ss] = 1  # dummy value
        if (key + 12) <= len(s):
            ss = s[key: key+12]
            if ss not in codes6:
                codes6[ss] = 1  # dummy value
        if (key + 16) <= len(s):
            ss = s[key: key+16]
            if ss not in codes8:
                codes8[ss] = 1  # dummy value

seqdata2 = np.zeros((df.shape[0], len(codes2)), dtype=int)
seqdata4 = np.zeros((df.shape[0], len(codes4)), dtype=int)
seqdata6 = np.zeros((df.shape[0], len(codes6)), dtype=int)
seqdata8 = np.zeros((df.shape[0], len(codes8)), dtype=int)
seqdf2 = pd.DataFrame(seqdata2, columns=codes2.keys())
seqdf4 = pd.DataFrame(seqdata4, columns=codes4.keys())
seqdf6 = pd.DataFrame(seqdata6, columns=codes6.keys())
seqdf8 = pd.DataFrame(seqdata8, columns=codes8.keys())

for i in xrange(df.shape[0]):
    s = df.iloc[i, 0]
    assert (len(s) % 2) == 0

    for key in range(0, len(s), 2):
        if (key + 4) <= len(s):
            ss = s[key: key+4]
            seqdf2[ss][i] += 1
        if (key + 8) <= len(s):
            ss = s[key: key+8]
            seqdf4[ss][i] += 1
        if (key + 12) <= len(s):
            ss = s[key: key+12]
            seqdf6[ss][i] += 1
        if (key + 16) <= len(s):
            ss = s[key: key+16]
            seqdf8[ss][i] += 1

================================================================================
CLASSIFIERS


# RANDOM FOREST
start = time.time()

# update model class and/or parameters to search over here
model = RandomForestClassifier()
estimators = [10, 50, 100, 150, 200, 250] # default = 10
features = [0.1, 0.25, 0.5, 0.75, 1.0, 'sqrt', 'log2'] # default = 'sqrt'
depths = [None] # default = None (i.e. ignored)
samples = [1, 5, 10, 25, 50, 100] # default = 1

# cross-validation on training set to identify optimal parameters
```

```
model = p2.cv_optimize(model, {'n_estimators': estimators, 'max_features': features,
'max_depth': depths, 'min_samples_leaf': samples}, X_train, Y_train)

# fit model on training set with optimal parameters
# check out-of-sample performance using validation set
model = p2.fit_model(model, X_train, Y_train, test_x=X_valid, test_y=Y_valid, title='Random Forest Features provi
tracker=tracker)

# fit model on entire training set with optimal parameters and make predictions
model = p2.fit_model(model, X_train_all, Y_train_all)
Y_test_pred = model.predict(X_test)
p2.write_predictions(Y_test_pred, test_ids, 'predictions/f02_random_forest.csv')

print '%0.1f seconds runtime' % (time.time() - start)

----------------------------------------------------------------------

# SVM
start = time.time()

model = LinearSVC(random_state=0)

Cs=[0.01, 0.1, 1.0, 10.0, 100.0] # default = 1
loss = ['squared_hinge', 'hinge'] # default=squared_hinge; 'hinge' is another option but cannot be combined with


# cross-validation on training set to identify optimal parameters
model = p2.cv_optimize(model, {'C': Cs, 'loss': loss}, ngrams_X_train, ngrams_Y_train)


# fit model on training set with optimal parameters
# check out-of-sample performance using validation set
model = p2.fit_model(model, ngrams_X_train, ngrams_Y_train, test_x=ngrams_X_valid, test_y=ngrams_Y_valid,
title='LinearSVC ngrams', tracker=tracker)

# fit model on entire training set with optimal parameters and make predictions
# model = p2.fit_model(model, ngrams_X_train_all, ngrams_Y_train_all)
Y_test_pred = model.predict(ngrams_X_test)
p2.write_predictions(Y_test_pred, test_ids, 'predictions/svc_ngram.csv')

print '%0.1f seconds runtime' % (time.time() - start)


======================================================================

VISUALIZATIONS

def plot_confusion_matrix(cm, title='Confusion matrix', cmap=plt.cm.Blues):
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(p2.MALWARE_CLASSES))
    plt.xticks(tick_marks, p2.MALWARE_CLASSES, rotation=45)
    plt.yticks(tick_marks, p2.MALWARE_CLASSES)
    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

svm_ngram_cm = confusion_matrix(ngrams_Y_valid, tracker['LinearSVC ngrams'][0].predict(ngrams_X_valid))
plot_confusion_matrix(svm_ngram_cm)


from sklearn.metrics import classification_report

print(classification_report(ngrams_Y_valid, ngrams_Y_valid_predict, target_names=p2.MALWARE_CLASSES))
```