Practical 2: Classifying Malicious Software

Christian Junge

Abhishek Malali

Vinay Subbiah

I. Technical Approach

1. **Training and Testing**

   To overcome the limitation imposed by limited Kaggle submissions, we used a small portion of our training data as a test set for developing and comparing models. Separating a small test set was a contentious affair considering the the training set was smaller than the test set in the problem. We randomly separated out 90% of our training data to use to train all of our models as we developed them, and used the remaining 10% of the training data as a hold-out validation set. This allowed us to explore and tune many models without being constrained by the small number of submissions. To test that the 90/10 split was reasonable, we retrained the baseline models using the training subset and calculated categorization accuracy on the validation set. A concern during this process was to keep the class distribution similar while splitting the training and test datasets. Each time we found a promising model, we then retrained it using the full training set before making predictions on the actual test set and making submissions.

2. **Exploratory Analysis**

   Since there was no curated training and test dataset available for this practical, a considerable time was spent understanding the .xml files to understand how malware conducts function calls to cause inconsistencies in operating system behavior. Exploratory analysis was hence the key to deciding what features we would generate from the files such that we can build a strong prediction model on the features.

   We started by taking simple counts of the function calls in each .xml file in the training set. We produced these as features, and then grouped by malware class and examined summary statistics for ideas about differences between the classes. We observed that some classes had much higher average counts of particular function calls than others, and some malware classes contained function calls that were not present in any other malware class. Unfortunately, the variation of the count of function calls was very high within classes, and there were several anomalous counts for particular observations. This exploration did suggest some directions for deeper exploration of the parameters to the function calls, however. For example, certain malware classes had many create_process commands, and we expected that the items in the log files associated with those commands might be informative.

3. **Feature Construction**

   Looking manually through the .xml files, we identified a few patterns that suggested potentially informative features. For example, many of the malware files contained 'sleep'

commands with long duration. We parsed the keys of the items associated with the 'sleep' commands to create the total time spent sleeping in the function as a feature.

Similarly, we created features from the number of failed processes, the fraction of processes that failed, the total run-time of the program, and the average length of the dump line arguments.

4. **Logistic Regression**

   Given we had just learnt Logistic Regression in class, we decided to leverage it in our analysis. Despite the fact that its a linear approach and our prior belief that non-linear approaches like neural networks or random forests would perform better, we decided to use Logistic Regression to gain some intuition both about the method and our problem. Our best Logistic Regression model used an L1 penalty, a one-vs-the-rest multiclasss approach and the liblinear solver without any rebalancing of the class weights.

5. **Neural Networks**

   Since we were being exposed to solving multi-class classification problems using Neural networks, we decided to implement a neural network to solve the Malware classification problem. For this purpose, we use the PyBrain library where we construct a three layer neural network. We have a input layer the size of a feature vector, a hidden layer of size 400 and an output layer which implements soft-max and has a size of 15 which is the number of classes. The node which has the highest probability corresponds to the class prediction by the neural network. The model was trained multiple times over a set number of epochs. In our case neural networks didn't do a great job of classification which resonates with our results.

6. **Random Forest**

   Our final model of best fit was the Random Forest Classifier. We iterated upon the basic random forest method that was set as a baseline, tuning number of estimators to find the model that gave the highest categorization accuracy. The Random Forest Classifier produced the best accuracy score on our hold-out test set and performed best on our Kaggle submissions.

II. Results

Table 1: Feature groups

Feature group	Features
A	Counting all 106 unique XML tags across all files
B	Adding new features

Table 2: Feature versions

Feature version	Feature groups
1	Feature group A
2	Feature group A, B

Table 3: Results across various models and feature choices

Model	Feature version	Categorization accuracy	Kaggle score
Neural Network	1	0.73	0.584
Random Forests	1	0.89	0.807
Logistic Regression	1	0.85	0.7526
Random Forests	2	0.91	0.8110

Table 4: Logistic Regression parameter tuning

Model	Feature version	Parameters	Categorization accuracy
Logistic regression	1	penalty l2 class weight balanced	0.6375
Logistic regression	1	penalty l1 class weight balanced	0.7994
Logistic regression	1	penalty l1	0.8447

Table 5: Random Forest Regression parameter tuning for number of estimators across feature versions

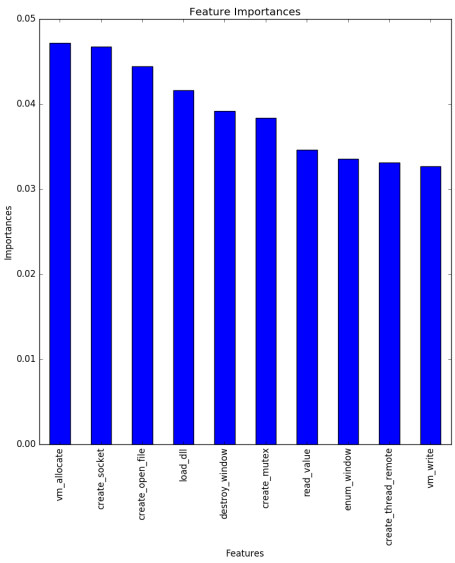
Model	Feature version	No of estimators	Out of Bag Score
Random Forest regression	1	30	0.8960
Random Forest regression	1	40	0.8976
Random Forest regression	1	50	0.8937
Random Forest regression	1	60	0.8944
Random Forest regression	1	70	0.8953
Random Forest regression	2	30	0.8889
Random Forest regression	2	40	0.8879
Random Forest regression	2	50	0.8879
Random Forest regression	2	60	0.8866
Random Forest regression	2	70	0.8914

III. Discussion

As described in the Technical approach section, the approach to classifying malware logs accurately began with a lot of exploratory analysis. From understanding the structure of XML logs to hypothesizing which elements of the log files would make good features, it was a much longer process than was initially anticipated. Our first set of features with considerable classification power were counts of all tags across the XML files. We used these features across a number of different model classes including Logistic Regression, Neural Networks and Random Forest Classifiers.

Across all three model classes, we tried tuning the model parameters to achieve the best fit. The metric that we chose to optimize for was categorization accuracy or classification accuracy. We used inbuilt metrics methods from the sklearn.metrics module to find methods that would work on multi-class classification problems and would evaluate the model based on actual predictions rather than say probabilitites prior to thresholding. We felt optimizing for this measure gave us the best shot on the Kaggle leaderboard. We tried tuning hyperparameters of the model classes but also experimented with flags like class-weight that helped the model rebalance among the classes. We felt this latter parameter might be especially insightful since our exploratory analysis pointed out numerous classes that were a tiny fraction of the overall dataset. However, as can

Figure 1: Feature Importance graph for Random forests for Features A

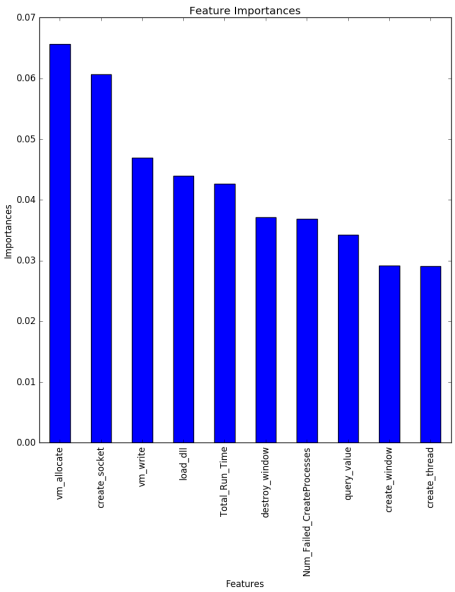


be seen from the results, balancing class weights did not necessarily give us the best predictive accuracy (something we would like to better understand going forward).

Our best results came from the Random Forest Classifier. Post that, we decided to spend time doing further feature engineering. From the leaderboard, it was quite clear that most teams were hovering around a very narrow band of classification accuracy scores and we felt new features might help us improve our score. As was discused in the Technical Approach, we engineered a few more complex features based on our hypotheses from analyzing the logs and that helped improve the model very marginally. With more time, we would look to perhaps understand some more on how malware works and see if we can map that understanding to what is observed in the logs to design better features.

References

Figure 2: Feature Importance graph for Random forests for Features A with B



IV. Appendix

gen_datasets.py

```
import os
import pandas as pd
import numpy as np
from collections import Counter
try:
    import xml.etree.cElementTree as ET
except ImportError:
    import xml.etree.ElementTree as ET
from scipy import sparse
import util

TRAIN_DIR = "../data/train"
call_set = set([])
def add_to_set(tree):
    for el in tree.iter():
        call = el.tag
        call_set.add(call)

#creating a set of features counting the number of tags
def call_feats(tree, good_calls):
    #Inputs
    #tree - tree object for every file
    #good_calls - list of tags for which we create the features
    call_counter = {}
    for el in tree.iter():
        call = el.tag
        if call not in call_counter:
            call_counter[call] = 0
        else:
            call_counter[call] += 1

    call_feat_array = np.zeros(len(good_calls))
    for i in range(len(good_calls)):
        call = good_calls[i]
        call_feat_array[i] = 0
        if call in call_counter:
            call_feat_array[i] = call_counter[call]
    return call_feat_array


###Creating function for loading data
def create_matrix(start_index, end_index, tags, direc="../data/train"):
    X = None
    classes = []
    ids = []
```

```
    i = -1
    for datafile in os.listdir(direc):
        if datafile == '.DS_Store':
            continue

        i += 1
        if i < start_index:
            continue
        if i >= end_index:
            break
        id_str, clas = datafile.split('.')[:2]
        ids.append(id_str)
        #adding target class to training data
        try:
            classes.append(util.malware_classes.index(clas))
        except ValueError:
            assert clas == "X"
            classes.append(-1)

        #parse file as an xml document
        tree = ET.parse(os.path.join(direc,datafile))
        add_to_set(tree)
        this_row = call_feats(tree, tags)
        if X is None:
            X = this_row
        else:
            X = np.vstack((X, this_row))

    return X, np.array(classes), ids


#List of unique tags
tags = []
for idx in range(numFiles):
    tree = ET.parse(os.path.join(TRAIN_DIR,fileList[idx]))
    for el in tree.iter():
        call = el.tag
        tags.append(call)
    tags = list(np.unique(tags))
unique_tags = np.unique(tags)
#Converting all tags to 'str' from  numpy.string_
unique_tags = [str(tag) for tag in unique_tags]
X_train, t_train, train_ids = create_matrix(0, numFiles,\
                                            unique_tags, TRAIN_DIR)


features_df = pd.DataFrame(X_train,columns=unique_tags)
features_df['class'] = t_train
```

```
features_df['id'] = train_ids
#Saving the features dataframe as a new file
features_df.to_csv('../outputs/features_v1.csv')


TEST_DIR = "../data/test"
testFileList = os.listdir(TEST_DIR)
numTestFiles = len(testFileList)
X_test, t_test, test_ids = create_matrix(0, numTestFiles,\
                                         unique_tags, TEST_DIR)

#Ignoring t_train since there is no response variable
features_test_df = pd.DataFrame(X_test,columns=unique_tags)
features_test_df['class'] = test_ids
features_test_df.to_csv('../outputs/features_test_v1.csv')
```

rf.py

```
from sklearn.cross_validation import train_test_split
from sklearn.ensemble import RandomForestClassifier

def genScore(RF, X_valid, y_valid):
    #generating the predictions
    pred = RF.predict(X_valid)
    #calculating classification accuracy
    return np.sum(pred == y_valid)/float(len(y_valid))


train = pd.read_csv('../outputs/features_v1.csv')
y = train['class']
train_ids = train['id']
X = train[train.columns[1:-2]]
X_train, X_valid, y_train, y_valid = train_test_split(X, y, \
                                test_size=0.10, random_state=1)
RF = RandomForestClassifier(n_estimators=10)
RF.fit(X_train, y_train)


#Loading the test data
test = pd.read_csv('../outputs/features_test_v1.csv')
test_ids = test['Id']
X_test = test[test.columns[1:-1]]
pred = RF.predict(X_test)
out_df = pd.DataFrame(test_ids, columns=['Id'])
out_df['Prediction'] = pred
out_df = out_df.set_index('Id')
out_df.to_csv('../outputs/RF_prediction.csv')
```

```
#Code for Logistic regression
X_train1, X_test_train, Y_train, Y_test = train_test_split(X_train, t_train, test_size=


LR = LogisticRegression(penalty='l1', multi_class='ovr',
                        n_jobs = -1, warm_start = True, solver = 'liblinear')
LR.fit(X_train1, Y_train)
LR_pred = LR.predict(X_test_train)

print(accuracy_score(Y_test, LR_pred))

LR_pred = LR.predict(X_test_real)

out_df = pd.DataFrame(test_ids, columns=['Id'])
out_df['Prediction'] = LR_pred
out_df = out_df.set_index('Id')

out_df.to_csv('../outputs/LR_prediction_V_LR.csv')
```