Practical 2

Samuel Daulton, Taylor Killian, and Andrew Petschek (ML Marauders)

March 2, 2016

1 Technical Approach

1.1 Feature Engineering

We were very excited to work on this Practical, as many of us have experience in the tech industry, and malware is a relevant topic in that area. In the first Practical, we learned that feature engineering played a critical component in our model's predictive success. Thus, we were keen on discovering new features here that could push our results above and beyond the baseline. When we first looked at the executables provided to us, our first approach was to track which different calls were used and count how many of them were employed in each sample (referred to as Stage I features below). This resulted in lots of new columns added to our design matrix. However, initial modeling attempts using all of these calls did not lead to great results. Thus, we decided to only include calls that seemed most important. We filtered through the calls and selected the ones that we felt likely had the most malicious intent/potential (referred to as Stage II features). For example, we included calls such as "impersonate user", "get computer name", "load", "get system directory", and many more. This feature engineering, when tested with our models, led to a significant increase.

Next, we wanted to see if we could build upon the success of our Stage II features. Thus, within each call that we tracked, we also analyzed which attributes were specifically called and the counts of each respectively (referred to as Stage III features). In this way were were able to gain better insight into what a potential Malware was executing, which was helpful in our classification results (as seen below).

1.2 Modeling Techniques

We approached our classification modeling from a couple of different angles. Specifically, we focused our efforts on Random Forest Classifiers, Logistic Regression Classifiers, and Naive

Bayes methodologies. In each method used, we leveraged cross-fold validation to evaluate optimal hyper parameter values for our classifiers. Cross-fold validation is helpful when you have a limited dataset, and don't want to further reduce sample size when splitting off both a Test and Validation set. Thus, we were able to retain the samples from what would have been a discrete Validation set. To carry out cross-fold validation, we used the library packages from scikit-learn and passed in the arrays of hyperparameters we wanted to test. Cross-validation is necessary because we do not know which hyperparameter values will minimize our loss function. Thus, the cross validation permits us to test an array of values and determine which one is best. In this way, our Test set is not used in this tuning step, and information from our Test set is prevented from leaking into our model fitting.

With the Logistic Regression Classifier, for example, we passed in an array of potential values that corresponded to the hyper-parameter of inverse of the regularization strength. We noted that smaller values tested specified stronger regularization (and potentially a better generalization to out of sample data). With Random Forest classifiers, we were able to better tune our model by searching over both the number of features used as well as number of estimators. This dual-loop cross validation proved to be very useful in our modeling efforts.

Another important aspect within our model fitting was ensuring the distribution of malware classes across folds/sets was consistent with the distribution provided to us in the Practical 2 specification. We were able to check this by employing histograms at each stage. We found that the distribution was relatively consistent, which allowed us to be more confident in our results. Finally, one last fix included starting our call counter at 1 instead of 0. Once we did that, our modeling ran better.

2 Results

We attempted many permutations of the above modeling techniques spanning the inclusion of various feature sets, models, and hyperparameters. We found our best results to be when we used our full feature set (Stage III) and a Random Forest Classifier. See Table 1 for our Accuracy results.

3 Discussion

This was a very interesting practical and led to some intriguing results. Our first pass at feature engineering generated no incremental gain. We found this to be quite puzzling, but presumed that we had not provided the model with enough information to beat its current baseline mark. If anything, this reinforced the notion that good feature engineering

Model	Accuracy
Random Forest with Stage III Features	0.82105
Logistic with Stage III Features	0.76053
Naive Bayes with Stage III Features	0.07040
Random Forest with Stage II Features	0.78526
Logistic with Stage II Features	0.75632
Naive Bayes with Stage II Features	0.07040
Random Forest with Stage I Features	0.39000
Logistic with Stage I Features	0.39000
Naive Bayes with Stage I Features	0.05010
Most Common Label Baseline	0.39000

Table 1: Model Results

is critical to model success. Data Science is the intersection between statistical knowledge, computer science, and domain knowledge. It is the domain knowledge in this case that helped with our feature engineering. Our team had some intuition surrounding what would make for a good feature. As one can see from our Stage II and Stage III results, better features (specifically those which were generated based on domain knowledge) led to higher accuracy.

Once we added new features we started to see improvement in our scores. However, it is worth noting that our Naive Bayes approach did not perform very well at all (even when compared to baseline). We continued to run it as a control group, but toward the end of our workflow we were focused on the Logistic and Random Forest Classification models.

Another aspect that was reinforced in this practical was the importance of cross-validation and hyper parameter tuning. As we have learned, one can reduce variance in a predictor by increasing sample size. Thus, cross-validation allows us to avoid losing sample to a discrete Validation set. Moreover, cross-validation tunes our models optimally before submitting against our Test set. Without cross-validation, we feel we would often over fit and get poor accuracy scores as a result.

If we had additional time to work on this project, we would have liked to implement a generative classifier. This method is advantageous in that it removes the need of a validation set. Instead, we leverage class conditional distributions to compute a posterior probably that is used for prediction against the Test set.

4 Code

```
# Example Feature Extraction from XML Files
# We count the number of specific system calls made by the programs, and use
# these as our features.

# This code requires that the unzipped training set is in a folder called "train".

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

import utils

TRAIN_DIR = "train"
TEST_DIR = "test"

call_set = set([])

def add_to_set(tree):
    for el in tree.iter():
        call = el.tag
        call_set.add(call)

def read_attributes(filename):
    fp = open(filename,'r')
    return [line.strip() for line in fp]

def create_data_matrix(start_index, end_index, good_attributes, good_calls,direc="t
    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

        # extract id and true class (if available) from filename
        id_str, clazz = datafile.split('.')[:2]
        ids.append(id_str)
        # add target class if this is training data
        try:
            classes.append(utils.malware_classes.index(clazz))

        except ValueError:
            # we should only fail to find the label in our list of malware classes
            # if this is test data, which always has an "X" label
            assert clazz == "X"
            classes.append(-1)

        # parse file as an xml document
        tree = ET.parse(os.path.join(direc,datafile))
        if training:
            add_to_set(tree)

    # 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
        this_row = call_feats(tree,good_attributes,good_calls)
        if X is None:
            X = this_row
        else:
            X = np.vstack((X, this_row))

    return X, np.array(classes), ids

def call_feats(tree, good_attributes, good_calls=None):
    #good_calls = ['sleep', 'dump_line', 'impersonate_user','revert_to_self','kill_
    # good_calls = list(good_calls)

    call_counter = {}
    att_counter = {}
    for el in tree.iter():
        call = el.tag
```

```
            if call not in call_counter:
                call_counter[call] = 0
            else:
                call_counter[call] += 1
            for att in el.attrib:
                if att in good_attributes:
                    if att not in att_counter:
                        att_counter[att] = 0
                    else:
                        att_counter[att] +=1

        feat_array = np.zeros(len(good_calls)+len(good_attributes)+1)
        for i in xrange(len(good_calls)):
            call = good_calls[i]
            feat_array[i] = 0
            if call in call_counter:
                feat_array[i] = call_counter[call]

        for i in xrange(len(good_calls),len(good_calls)+len(good_attributes)):
            att = good_attributes[i-len(good_calls)]
            feat_array[i] = 0
            if att in att_counter:
                feat_array[i] = att_counter[att]

        feat_array[len(good_calls)+len(good_attributes)-1] = len(call_counter.keys())+l

        return feat_array

    ## Feature extraction
    def main():
        num_of_train_files = len(os.listdir(TRAIN_DIR))
        num_of_test_files = len(os.listdir(TEST_DIR))

        # Read in attribute names from file
        good_attributes = read_attributes('attributes.txt')
        good_calls = read_attributes('calls.txt')

        X_train, t_train, train_ids = create_data_matrix(0, num_of_train_files, good_at
        X_test, t_test, test_ids  = create_data_matrix(0, num_of_test_files, good_attri

        #
        calls = set(calls_train) | set(calls_test)
        fp = open("calls.txt",'w')
        for call in calls:
            fp.write(call + '\n')
```

```
    fp.close()

    print len(call_set)
    print X_train.shape, t_train.shape
    print X_test.shape, t_test.shape

    # From here, you can train models (eg by importing sklearn and inputting X_trai

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

import utils
import TWK_feat_eng
import matplotlib.pyplot as plt
import numpy as np

from sklearn.preprocessing import StandardScaler
from sklearn.cross_validation import train_test_split

from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.grid_search import GridSearchCV


TRAIN_DIR = "train"
TEST_DIR = "test"

def create_submission(ids,predictions,filename):
        with open(filename, "w") as f:
                f.write("Id,Prediction\n")
                for i,p in zip(ids,predictions):
                        f.write(str(i) + "," + str(p) + "\n")

if __name__ == "__main__":
        num_of_train_files = len(os.listdir(TRAIN_DIR))
        num_of_test_files = len(os.listdir(TEST_DIR))
        good_attributes = TWK_feat_eng.read_attributes('attributes.txt')
        good_calls = TWK_feat_eng.read_attributes('calls.txt')
        X_train, t_train, train_ids = TWK_feat_eng.create_data_matrix(0, num_of_tra
        full_test, _, test_ids = TWK_feat_eng.create_data_matrix(0, num_of_test_fil
```

```
xX_train, xX_valid, xY_train, xY_valid = train_test_split(X_train, t_train,

# Quickly check to see if distributions are well mixed... first pass showed
# plt.figure()
# plt.hist(xY_train,bins=20,alpha=0.5)
# plt.hist(xY_valid,bins=20,alpha=0.5)
# plt.show()

print "We've compiled the training data and have split it into training/val
print "Train set dims: ", X_train.shape, "Number of training files: ", num_
print "Test set dims: ", full_test.shape, "Number of testing files: ", num_

# Standardize the data!
scaler = StandardScaler(copy=True, with_mean=True, with_std=True)
xX_train = scaler.fit_transform(xX_train)
xX_valid = scaler.transform(xX_valid)

X_train = scaler.fit_transform(X_train)
full_test = scaler.fit(full_test)

n_folds = 8
n_jobs = 2

# Initialize different classifiers
logReg_clf = LogisticRegression()
rf_clf = RandomForestClassifier()
gb_clf = GradientBoostingClassifier()

# Pass logistic regression through GridSearchCV, just cause
Cs=[0.001, 0.01, 0.1, 1.0, 10.0, 100.0]
parameters = {"C": Cs}

gs_logReg_clf = GridSearchCV(logReg_clf, param_grid=parameters, cv=n_folds,
gs_logReg_clf.fit(xX_train,xY_train)
print "BEST", gs_logReg_clf.best_params_, gs_logReg_clf.best_score_, gs_log
best_logReg_clf = gs_logReg_clf.best_estimator_
best_logReg_clf = best_logReg_clf.fit(xX_train,xY_train)

logReg_trainingAcc = best_logReg_clf.score(xX_train,xY_train)
logReg_validationAcc = best_logReg_clf.score(xX_valid,xY_valid)

print "############# LOGISTIC REGRESSION RESULTS################"
print "Accuracy on training data:    %0.2f" % (logReg_trainingAcc)
print "Accuracy on validation data:  %0.2f" % (logReg_validationAcc)
print "#########################################################\n"
```

```
# Just to finely tune our Random Forest, I'm going to pass it through a lar

rf_params = {"max_depth": [3, None], "max_features": [1, 3], \
"n_estimators": [50, 100, 200, 300], "min_samples_split": [1, 3], \
"min_samples_leaf": [1, 3], "bootstrap": [True, False], \
"criterion": ["gini", "entropy"]}

gs_rf_clf = GridSearchCV(rf_clf,param_grid=rf_params,cv=n_folds,n_jobs=n_jo
gs_rf_clf.fit(xX_train,xY_train)
print "BEST RF:   ", gs_rf_clf.best_params_, gs_rf_clf.best_score_
best_rf_clf = gs_rf_clf.best_estimator_
best_rf_clf = best_rf_clf.fit(xX_train,xY_train)

rf_trainingAcc = best_rf_clf.score(xX_train,xY_train)
rf_validationAcc = best_rf_clf.score(xX_valid,xY_valid)

print "############### RANDOM FOREST RESULTS ####################"
print "Accuracy on training data:    %0.2f" % (rf_trainingAcc)
print "Accuracy on validation data:  %0.2f" % (rf_validationAcc)
print "##########################################################\n"

# Now running a gradient boosting classifier, through grid search to finely

gb_params = {"max_depth": [3, None], "max_features": [1, 3], \
"n_estimators": [50, 100, 150], "min_samples_split": [1, 3], \
"min_samples_leaf": [1, 3], "learning_rate": [0.001,0.01,0.1], \
"loss": ["deviance", "exponential"], "subsample": [0.25, 0.5, 1.]}

gs_gb_clf = GridSearchCV(gb_clf,param_grid=gb_params,cv=n_folds,n_jobs=n_jo
gs_gb_clf.fit(xX_train,xY_train)
print "Best GB:  ", gs_gb_clf.best_params_, gs_gb_clf.best_score_
best_gb_clf = gs_gb_clf.best_estimator_
best_gb_clf = best_gb_clf.fiti(xX_train,xY_train)

gb_trainingAcc = best_gb_clf.score(xX_train,xY_train)
gb_validationAcc = best_gb_clf.score(xX_valid, xY_valid)

print "############# GRADIENT BOOSTING RESULTS ##################"
print "Accuracy on training data:    %0.2f" % (gb_trainingAcc)
print "Accuracy on validation data:  %0.2f" % (gb_validationAcc)
print "##########################################################\n"

# Now predict against full_test set
print "Now predicting against test set"
```

```
if gb_validationAcc >= rf_validationAcc:
        best_gb_clf.fit(X_train,t_train)
        preds = best_gb_clf.predict(full_test)
else:
        rf_clf.fit(X_train,t_train)
        preds = rf_clf.predict(full_test)

# Just out of curiosity, I want to check the histogram of the predictions a
plt.figure()
plt.hist(t_train,bins=20,alpha=0.5)
plt.hist(preds,bins=20,alpha=0.5)
plt.show()

# Now creating submission
create_submission(test_ids,preds,'third_pass_fullSJDcalls_n_attributes_TWK.
```