Malware Analysis

Machine Learning

Homework 1

Ahmed Tarik Mohamed Elhelow

1823229

Table of Contents

1 Introduction

The purpose of this homework is to train and compare different machine learning classifiers on the DREBIN dataset so that we have a mathematical model able to identify malware application determined by the features of the application itself. To accomplish this, the dataset has been preprocessed in order to make it possible to lessen the time of training and have an operating model in a small time. Particularly, the vocabulary size has been decreased to 7546, only a subset of all the features has been used, and the training set and the test set have size of 8892 and 2223 elements respectively. All of the malware applications have been used on all trials. Two different classifiers have been used and compared by different metrics. Specifically, Naïve Bayes and Support Vector Machine have been used. The project has been coded by Python programming language with the help of Scikit-learn as a machine learning library. The Support Vector Machine classifier achieved better results with an accuracy of 0.962, a precision of 0.97, a recall of 0.956, and a false positive rate of 0.032.

2 DREBIN Dataset

DREBIN is a dataset of Android malwares developed to study machine learning approaches for malware detection. It's composed of 123,453 benign applications and 5,560 malwares where each program is represented by a set of features which were extracted from the manifest file of Android (AndroidManifest.xml) and from the disassembled code of the application at runtime.

All the features are grouped in subsets. In particular, there are four groups from the manifest file:

- S1: Hardware components
- S2: Requested permissions
- S3: App components
- S4: Filtered intents

And four groups from the disassembled code:

- S5: Restricted API calls
- S6: Used permissions
- S7: Suspicious API calls
- S8: Network addresses

3 Preprocessing

In order to make it possible to train the classifiers on the dataset, all the data needs to be preprocessed. The features of each application will be inserted in a vector space. Each element of the vector can be either 1 which indicates the presence of the feature in the program or 0 if not present.

To accelerate the training time, only a subset of features and a subset of applications have been included in the training set. Specifically, categories Activity, Feature and URL have been discarded, and only a subset of 8892 files has been used for training so that the dimension of the vector space has been decreased.

4 Development

Below the development of the project is detailed.

4.1 Import Libraries

In the following block is listed all the necessary libraries for the project, the seed for the random numbers and the path for the folder containing.

```
from __future__ import division
from sklearn.model_selection import train_test_split
from sklearn import svm
from sklearn.naive_bayes import BernoulliNB
from sklearn.metrics import confusion_matrix
import csv
import os, os.path
import random
import time

folder = 'drebin/feature_vectors/'

# Set seed for random numbers
random.seed(5)
```

4.2 Dataset Balancing

Randomly select the same number of malware examples from the non-malware in order to have a balanced dataset.

```
# List of all the files
dataset = os.listdir(folder)
print 'done reading all files'

# Create malware list with each file name
with open(folder + '../sha256_family.csv') as csvfile:
    reader = csv.reader(csvfile)
```

```
    next(reader)
    malware = [row[0] for row in reader]
print 'done reading csv file'

# Generate random numbers to get a random set of
# non-malware files
index = random.sample(range(0,\
 len(dataset)-len(malware)-1), len(malware))

non_malware = [list(set(dataset)-set(malware))[i]\
 for i in index]
print 'done creating the non-malware list'

# Merged list containing malware and non malware
data = malware + non_malware
random.shuffle(data)

# Print number of examples in each class
print('Number of non malware: ', len(non_malware))
print('Number of malware: ', len(malware))
```

Now both classes have the same number of examples, both with 5560 files.

4.3 Features Categories

As explained above, each file contains all the features extracted from the application. These features are extracted from the manifest.xml file and from the disassembled code.

The features can be divided in 10 different sets:

- api_call
- permission
- intent
- call
- real_permission
- provider
- service_receiver
- activity
- feature
- url

In the list of features below it is possible to set True or False for each one of them, True to be considered by the classifier and False to be discarded.

```
# assign each feature to True or False
features = {
    'api_call': True,
    'permission': True,
    'intent': True,
    'call': True,
    'real_permission': True,
    'provider': True,
    'service_receiver': True,
    'activity': False,
    'feature': False,
    'url': False
}
```

4.4 Features Extraction

This is the code used to extract features of each file. It is also build the vocabulary vector in the process. Moreover, it generates the feature vector which contains one in the corresponding position of the feature in the vocabulary vector if this feature exists, otherwise zero.

```
# Creating a vocabulary from the dataset
# Creating X feature vectors
# Creating Y result vector
vocabulary = []
X = []
Y = []
cnt = 0
percent = 0

for file in data:
    cnt = cnt + 1
    if (1.0 * cnt / len(data)) - percent >= 0.1:
        percent = 1.0 * cnt / len(data)
        print 'Progress: {:2.1%}'.format(percent)
    try:
        features_vector = [0] * len(vocabulary)
        for feature in open(folder + file):
            if features[feature.split("::")[0]] == False:
                continue
            feature = feature.rstrip()
            if feature in vocabulary:
                features_vector[vocabulary.\
                index(feature)] = 1
            else:
                vocabulary.append(feature)
                features_vector.append(1)
        X.append(features_vector)
        Y.append(1 if file in malware else 0)
    except Exception as e:
        pass

print 'done constructing X, Y initial values'
```

```
# fix X to have the same length as vocabulary
for i in range(len(X)):
    X[i] = X[i] + [0]*(len(vocabulary)-len(X[i]))

print 'done fixing X to have same length as vocabulary'
```

4.5 Cross Validation

In order to have a good evaluation of the used algorithm, it is important to separate the dataset between training and test set. The training set will be used to train the classifier and the test set will only be used to calculate the metrics.

It is important for the classifier do not see the test set before, because it could over-fit the data and not be able to generalize for new examples.

```
# Cross Validation
X_train, X_test, Y_train, Y_test = train_test_split(X,\
 Y, test_size=0.2)

print('Training set size:', len(X_train))
print('Test set size:', len(X_test))
```

5 Classifiers

For this project, two classifiers were chosen to be trained with the same dataset. The first classifier is the Naive Bayes and the second one is Support Vector Machines.

5.1 Naive Bayes

Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature. Even if the features depend on each other or upon the existence of the other features, all of these properties independently contribute to the probability that this fruit is an apple and that is why it is known as 'Naïve'.

Naive Bayes model is easy to build and particularly useful for very large data sets. Along with simplicity, Naive Bayes is known to outperform even highly sophisticated classification methods.

Below is the implementation of the Naive Bayes.

```
# Implement Naive Bayes
def train_naive_bayes_classifer(X_train, X_test,\
 Y_train):
    # Define the classifier
    gnb = BernoulliNB(alpha=1.0, binarize=None)

    # Check the training time
    t=time.time()
    gnb.fit(X_train, Y_train)
    t2 = time.time()
    print(round(t2-t, 2),\
    'Seconds to train Naive Bayes...')

    # Check the score of the SVC
    Y_predict=gnb.predict(X_test)

    return Y_predict
```

5.2 Support Vector Machine

Support Vector Machine (SVM) is a supervised machine learning algorithm which can be used for either classification or regression challenges. However, it is mostly used in classification problems.

SVM is mostly useful in non-linear separation problems, because it has a technique called the kernel. These are functions which take low dimensional input space and transform it to a higher dimensional space, which means that it converts not separable problem to separable problem.

Below is the implementation of the Support Vector Machine.

```
# Implement Support Vector Machine
def train_svm_classifer(X_train, X_test, Y_train):
    # Define the classifier
    svc = svm.LinearSVC(C=1.0, max_iter=4000)

    # Check the training time
    t=time.time()
    svc.fit(X_train, Y_train)
    t2 = time.time()
    print(round(t2-t, 2), 'Seconds to train SVM...')

    # Check the score of the SVC
    Y_predict=svc.predict(X_test)

    return Y_predict
```

6 Metrics

Here is the code to calculate the metrics which will be used to compare both algorithms used. They will be compared using the following metrics:

- True positives (TP) - Malware correctly classified
- True negatives (TN) - Non malware correctly classified
- False positives (FP) - Non malware classified as malware
- False negative (FN) - Malware classified as non-malware

```
# Get metrics True Negative, False Positive,
# False Negative, True Positive
tn, fp, fn, tp = confusion_matrix(Y_test, Y_predict).\
ravel()

print 'True Negative: ', tn
print 'False Positive: ', fp
print 'False Negative: ', fn
print 'True Positive: ', tp
```

Besides, we can use this data to calculate some more interesting metrics:

6.1 Accuracy

Percentage of files correctly classified.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

```
# Accuracy
accuracy = (tp + tn)/(fp + fn + tp + tn)

print "accuracy: " + str(round(accuracy*100,2)) + '%'
```

6.2 Precision

Percentage of real malware detected in relation to all classified as malware.

$$Precision = \frac{TP}{TP + FP}$$

```
# Precision
precision = tp/(tp + fp)

print "precision: " + str(round(precision*100,2)) + '%'
```

6.3 Recall

Percentage of real malware detected in relation to all malware in the set.

$$Recall = \frac{TP}{TP + FN}$$

```
# Recall
recall = tp/(tp + fn)

print "recall: " + str(round(recall*100,2)) + '%'
```

6.4 False Positive Rate

Percentage of wrongly files classified as malware in relation to all benign files.

$$FalsePositiveRate = \frac{FP}{FP + TN}$$

```
# False Positive Rate
false_pos_rate = fp/(fp + tn)

print "false_pos_rate: " +\
str(round(false_pos_rate*100,2)) + '%'
```

6.5 F-measure

Weighted average of precision and recall.

$$FMeasure = \frac{2 * precision * reacll}{precision + recall}$$

```
# F-Measure
f_measure = 2 *(precision * recall)/(precision + recall)

print "f_measure: " + str(round(f_measure*100,2)) + '%'
```

7 Results & Conclusion

After having all the functions for the analysis explained and implemented, it is possible to train it with the training data and validate with the test data.

It was decided to start with the Naive Bayes algorithm.

```
# Train Naïve Bayes Classifier

print "Train Naïve Bayes Classifier..."
Y_predict = train_naive_bayes_classifer(X_train,\
X_test, Y_train)
```

An accuracy of approximately 86.36% was obtained, which is not bad, but not very good either.

```
(8.46, 'Seconds to train Naive Bayes...')

True Negative: 1011
False Positive: 67
False Negative: 214
True Positive: 931

accuracy: 87.36%
precision: 93.29%
recall: 81.31%
false_pos_rate: 6.22%
f_measure: 86.89%
```

After obtaining this result, it was decided to train with a different classifier, in this case SVM.

```
# Train SVM Classifier
print "Train SVM Classifier..."
Y_predict = train_svm_classifer(X_train, X_test, Y_train)
```

It is possible to notice that for the SVM, the numbers got much better with a smaller time for training. Here, the accuracy is about 96.18% with a false positive rate a little bit above 3%.

```
(6.09, 'Seconds to train SVM...')

True Negative: 1044
False Positive: 34
False Negative: 51
True Positive: 1094

accuracy: 96.18%
precision: 96.99%
recall: 95.55%
false_pos_rate: 3.15%
f_measure: 96.26%
```

In this case, the SVM algorithm presented a better performance for the classification of mal-ware and non-malware examples.