Malware Analysis

November 18, 2017

by Ibis Prevedello (1794539)

1 Introduction

The goal of this project is to train different classifiers in order to separate malware from non malware aplications running in an Android OS, given features extracted from the `manifest.xml` file and from the disassembled code.

The dataset that is being used here is the The Drebin Dataset from the Braunschweig University of Technology.

The dataset contains 5,560 files from 179 different malware families. The samples were collected in the period of August 2010 to October 2012. More details on the dataset can be found in the paper describing Drebin and the corresponding evaluation.

2 Development

Below the development of the project is detailed, commenting each of the functions and steps given to achieve the desired goals for the project.

2.1 Import necessary libraries for the project

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.

```
In [1]: # Difine libraries
        from collections import defaultdict, OrderedDict
        from urllib.parse import urlsplit
        from tqdm import tqdm
        from sklearn.model_selection import GridSearchCV
        from sklearn.model_selection import train_test_split
        from sklearn.model_selection import KFold
        from sklearn import svm
        from sklearn.metrics import classification_report
        from sklearn.naive_bayes import BernoulliNB

        import csv
        import os, os.path
```

```
import pickle
import random
import time
import matplotlib.pyplot as plt

# Set seed for random numbers
random.seed(1)

# Define folder where the log files are located
folder = 'drebin/feature_vectors/'
```

2.2 Understanding the dataset

As described above, the dataset is composed of examples of malware and non-malware data, each file contains features of requested hardware components, requested permissions, app components, network addresses and so on. An example of one file is presented below.

```
In [2]: sample_file = \
            'd4e03d6e85b2f45a754240136d7544351b733fb12d9bb7511227da31bc709399'

        print('File: ', sample_file, '\n')

        # Print its content
        with open(folder + sample_file) as f:
            content = f.read().splitlines()
            for line in content:
                print(line)
```

```
File:  d4e03d6e85b2f45a754240136d7544351b733fb12d9bb7511227da31bc709399

activity::.PictureMapper
feature::android.hardware.touchscreen
intent::android.intent.action.MAIN
api_call::android/content/ContentResolver;->query
activity::Screen2
permission::android.permission.INTERNET
api_call::android/app/Activity;->startActivity
intent::android.intent.category.LAUNCHER
real_permission::android.permission.READ_CONTACTS
```

Just to start to have a better understanding about the dataset, it is interesting to find out the total number of malware and non-malware files.

```
In [3]: # Define list for the malware and non malware files
        non_malware = list()
        malware = list()

        # List of all the files
```

```
        dataset = os.listdir(folder)

        # Create malware dictionary with the file name and the type of each one
        with open('drebin/sha256_family.csv') as csvfile:
            reader = csv.reader(csvfile)
            next(reader)
            malware_dictionary = {row[0]:row[1] for row in reader}

        # Separate the file names between malware and non malware
        for i in dataset:
            if i in malware_dictionary:
                malware.append(i)
            else:
                non_malware.append(i)

        print('Size of dataset: ', len(malware) + len(non_malware))
        print('Number of non malwares:\t', len(non_malware))
        print('Number of malwares:\t', len(malware))
```

```
Size of dataset:  129013
Number of non malwares:         123453
Number of malwares:         5560
```

It is possible to notice that the dataset is not balanced, in this dataset, the number of non malware files is much larger than the number of malware, and this can influence on the classification. So, before to start it is good to pick randomly the same number of files from the non malware class.

2.3 Balancing the dataset

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

```
In [4]: # Generate random numbers
        index = random.sample(range(0, len(non_malware)-1), len(malware))

        # New list with malware and non malware examples divided
        non_malware = [non_malware[i] for i in index]

        # Merged list containing malware and non malware
        data = malware + non_malware

        # Vector with class of each example
        y = [1]*len(malware) + [0]*len(non_malware)

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

```
Number of non malware:  5560
Number of malware:  5560
```

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

2.4 Malware categories

Just to have an idea of the dataset, below is presented a list of how many samples there are in each malware class. This data will not be used for this project, but further development can be done in order to classify the class each malware belongs to.

```
In [5]: print('\nNumber of entries in each class of malware (values above 20):')

        v = defaultdict(list)
        for key, value in sorted(malware_dictionary.items()):
            v[value].append(key)
        ordered_v = \
            OrderedDict(sorted(v.items(), key=lambda x: len(x[1]), reverse=True))
        count_malware = 0
        for k in ordered_v:
            if len(v[k])>20: # Print only classes with more than 20 examples
                count_malware += len(v[k])
                print('\t', '{:>3}'.format(len(v[k])), k)

        print('\t', '{:>3}'.format(len(malware) - count_malware), 'Others')
```

```
Number of entries in each class of malware (values above 20):
         925 FakeInstaller
         667 DroidKungFu
         625 Plankton
         613 Opfake
         339 GinMaster
         330 BaseBridge
         152 Iconosys
         147 Kmin
         132 FakeDoc
          92 Geinimi
          91 Adrd
          81 DroidDream
          70 ExploitLinuxLotoor
          69 Glodream
          69 MobileTx
          61 FakeRun
          59 SendPay
          58 Gappusin
          43 Imlog
          41 SMSreg
```

```
 37 Yzhc
 29 Jifake
 28 Hamob
 27 Boxer
775 Others
```

2.5 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.

Below is presented the total number of categories found in the files.

```
In [6]: # Collect all the features found in the first 20 files of the dataset
        category_list = list()
        for file in dataset[0:20]:
            with open(folder + file) as f:
                content = f.readlines()
                for line in content:
                    category, string = line.split('::')
                    if category not in category_list:
                        category_list.append(category)

        print('The features can be divided in', \
              len(category_list), 'different sets:')
        print('\t','\n\t '.join(category_list))
```

```
The features can be divided in 10 different sets:
         api_call
         feature
         url
         service_receiver
         permission
         call
         intent
         real_permission
         activity
         provider
```

In order to make the problem easier or more difficult to solve, it is possible to chose which features will be considered in order to classify between the two classes.

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.

```
In [7]: features = {
            'api_call': True,
            'feature': True,
            'url': True,
```

```
        'service_receiver': True,
        'permission': True,
        'call': True,
        'intent': True,
        'real_permission': True,
        'activity': True,
        'provider': True,
    }
```

2.6 Functions to extract features

The functions listed below are used to extract the data necessary from the features. It receives each line of the files based on the category and returns a sequence of words that can be used to create the dictionary and/or construct the vector of features.

```
In [8]: # Extract url
        def extract_url(string):
            try:
                base_url = "{0.scheme}://{0.netloc}/".format(urlsplit(string))
                if len(base_url) > 10:
                    return [base_url]
            except:
                #print('Error html: ', string)
                return None

        # Extract api_call
        def extract_api_call(string):
            try:
                string = string.replace(';->', '/')
                api_call = string.split('/')
                return api_call
            except:
                return None

        # Extract feature
        def extract_feature(string):
            try:
                feature = string.split('.')[-1]
                return [feature]
            except:
                return None

        # Extract permission and real_permission
        def extract_permission(string):
            try:
                permission = string.split('.')[-1].lower()
                return [permission]
            except:
```

```
        return None

# Extract call
def extract_call(string):
    try:
        call = string.lower()
        return [call]
    except:
        return None

# Extract activity
def extract_activity(string):
    try:
        activity = string.split('.')[-1].lower()
        return [activity]
    except:
        return None

# Extract intent
def extract_intent(string):
    try:
        intent = string.split('.')[-1].lower()
        return [intent]
    except:
        return None

# Extract service_receiver
def extract_service_receiver(string):
    try:
        service_receiver = string.split('.')[-1].lower()
        return [service_receiver]
    except:
        return None

# Extract provider
def extract_provider(string):
    try:
        provider = string.split('.')[-1].lower()
        return [provider]
    except:
        return None
```

2.7 Map words to index

It is important to create a dictionary with all the words containing in the files and convert each file to a vector of the index of each word in the dictionary, as show in the example below:

`000068216bdb459df847bfdd67dd11069c3c50166db1ea8772cdc9250d948bcf`

```
[3, 55, 56, 11, 13, 57, 22, 58, 59, 52, 60, 61, 4, 5, 6, 62, 63, 64, 49, 50, 51]
```

The following function receives a file, reads its line, process using the functions to extract features and creates or the dictionary or the vector shown above.

```
In [9]: # Create dictionary or list of words
        def process_file(file, dictionary_creation = False):

            # List of words of each file
            words = list()

            # Read line by line of the file
            with open(folder + file) as f:
                content = f.readlines()

                # Divide each line of document in
                for line in content:
                    try:
                        split = line.split('::')
                        category = split[0]
                        string = split[1]
                    except:
                        break

                    # Only process the categories selected by the user
                    if (features[category]):
                        if (category == 'url'):
                            word_list = extract_url(string)
                        elif (category == 'api_call'):
                            word_list = extract_api_call(string)
                        elif (category == 'feature'):
                            word_list = extract_feature(string)
                        elif (category == 'permission' or \
                              category == 'real_permission'):
                            word_list = extract_permission(string)
                        elif (category == 'call'):
                            word_list = extract_call(string)
                        elif (category == 'activity'):
                            word_list = extract_activity(string)
                        elif (category == 'intent'):
                            word_list = extract_intent(string)
                        elif (category == 'service_receiver'):
                            word_list = extract_service_receiver(string)
                        elif (category == 'provider'):
                            word_list = extract_provider(string)

                        # If able to extract feature from line
                        if word_list != None:
```

```
                for word in word_list:
                    word = word.replace('\n', '')

                    # If flagged to create the dictionary
                    if dictionary_creation:
                        index = len(dictionary)-1
                        dictionary[word] = index
                    else:
                        index = dictionary[word]
                        if index not in words:
                            words.append(index)

    return words
```

2.8 Dictionary creation

In order to classify the files, first a dictionary needs to be created using the words of the files that it will classify. For the dictionary all the words containing in each file is being used for each of the categories showed above. As this creating takes time, because it is necessary to read all the words containing in each file, the dictionary is being save to a file called `dictionary.pkl`. It will only be create if it is not found on the directory folder.

```
In [10]: # Save dictionary to file
         def save_dic(obj, name):
             with open(name + '.pkl', 'wb') as f:
                 pickle.dump(obj, f, pickle.HIGHEST_PROTOCOL)

         # Load dictionary from file
         def load_dic(name):
             with open(name + '.pkl', 'rb') as f:
                 return pickle.load(f)

         # Check if dictionary exists
         if os.path.isfile('dictionary.pkl'):
             print('Dictionary file found!')
             dictionary = load_dic('dictionary')

         else:
             print('Dictionary file not found!')

             # Define the dictionary
             dictionary = {}

             # Colect words for malware
             pbar = tqdm(range(len(data)))
             pbar.set_description('Creating dictionary')
             for i in pbar:
                 process_file(data[i], True)
```

```
            # Save dictionary
            save_dic(dictionary, 'dictionary')
            print('\nDictionary saved to file!')

        # Print number of words in the dictionary
        dictionary_size = len(dictionary)
        print('Dictionary size: ', dictionary_size)
```

```
Dictionary file found!
Dictionary size:  28959
```

2.9 Features extraction

This is the function used to extract features of each file, it uses the same function to build the dictionary, but here a vector of zeros is created with the length of the dictionary, and its value is set to one only in the position of the words that contains in the file.

```
In [11]: # Construct the vector of features
         def features_extraction(file):
             indices = process_file(file)

             feat = [0] * dictionary_size

             for i in indices:
                 feat[i-1] = 1

             return feat

         # List of features
         X = list()

         # Extract features and append to the list of features
         pbar = tqdm(range(len(data)))
         pbar.set_description('Extracting features')
         for i in pbar:
             feat = features_extraction(data[i])
             X.append(feat)
```

```
Extracting features: 100%|| 11120/11120 [07:17<00:00, 25.39it/s]
```

2.10 Train test split

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 overfit the data and not be able to generalize for new examples.

```
In [12]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

3 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*.

3.1 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 'Naive'.

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*.

3.1.1 Pros and Cons associated with Naive Bayes

Pros

- It is easy and fast to predict class of test data set. It also perform well in multi class prediction
- When assumption of independence holds, a Naive Bayes classifier performs better compare to other models like logistic regression and you need less training data
- It perform well in case of categorical input variables compared to numerical variable(s). For numerical variable, normal distribution is assumed (bell curve, which is a strong assumption)

Cons

- If categorical variable has a category (in test data set), which was not observed in training data set, then model will assign a zero probability and will be unable to make a prediction (often known as "Zero Frequency"). To solve this, it is possible to use the smoothing technique, one of the simplest smoothing techniques is called Laplace estimation
- On the other side Naive Bayes is also known as a bad estimator, so the probability outputs from `predict_proba` are not to be taken too seriously
- Another limitation of Naive Bayes is the assumption of independent predictors. In real life, it is almost impossible that we get a set of predictors which are completely independent

```
In [20]: # 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 for the SVC
    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
```

3.2 Support Vector Machine

Support Vector Machine (SVM) is a supervised machine learning algorithm which can be used for both 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** trick, these are functions which takes 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*.

3.2.1 Pros and Cons associated with SVM

Pros

- It works really well with clear margin of separation
- It is effective in high dimensional spaces
- It is effective in cases where number of dimensions is greater than the number of samples
- It uses a subset of training points in the decision function (called support vectors), so it is also memory efficient

Cons

- It does not perform well, when we have large data set because the required training time is higher
- It also does not perform very well, when the data set has more noise, which means that target classes are overlapping

```
In [18]: # Implement Support Vector Machine
         def train_svm_classifer(X_train, X_test, y_train):

             # Define the classifier
             svc = svm.LinearSVC(C=1.0)

             # Check the training time for the SVC
             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
```

3.3 Metrics

Here is defined the function to calculate the metrics with 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

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

3.3.1 Precision

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

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

3.3.2 Recall

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

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

3.3.3 False positive rate

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

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

3.3.4 Accuracy

Percentage of files correctly classified.

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

3.3.5 F-measure

Weighted average of precision and recall.

$$FMeasure = \frac{2 * Precision * Recall}{Precision + Recall}$$

```
In [15]: # Function to calculate the metrics
         def metrics(y_test, y_predict):

             # Number of examples
             n = len(y_test)

             # Calculate true positives
             true_positive = \
                 sum([y_test[i] and y_predict[i] for i in range(n)])
             print('\nTrue positive', true_positive)

             # Calculate false positives
             false_positive = \
                 sum([not(y_test[i]) and y_predict[i] for i in range(n)])
             print('False positive', false_positive)

             # Calculate false negatives
             false_negative = \
                 sum([y_test[i] and not(y_predict[i]) for i in range(n)])
             print('False negative', false_negative)

             # Calculate true negatives
             true_negative = \
                 sum([not(y_test[i]) and not(y_predict[i]) for i in range(n)])
             print('True positive', true_negative)

             # Calculate precision
             precision = true_positive/(true_positive + false_positive)
             print('\nPrecision', round(precision*100,2), '%')

             # Calculate recall
             recall = true_positive/(true_positive + false_negative)
             print('Recall', round(recall*100,2), '%')

             # Calculate false positive rate
             false_pos_rate = false_positive/(false_positive + true_negative)
             print('False positive rate', round(false_pos_rate*100,2), '%')

             # Calculate accuracy
             accuracy = (true_positive + true_negative)/ \
                             (false_positive + false_negative + \
                              true_positive + true_negative)
             print('Accuracy', round(accuracy*100,2), '%')

             # Calculate accuracy
             f_measure = 2 *(precision * recall)/(precision + recall)
             print('F-measure', round(f_measure*100,2), '%')
```

4 Results

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.

Starting with the *Naive Bayes* algorithm, it is possible to check that it is was obtained a accuracy of approximately 92.4%, what is good, however, the number for false positive was a bit high, above 9%.

After obtaining this result, it was decided to train with a different classifier, in this case SVM, because this is a good application for this algorithm.

```
In [21]: y_predict = train_naive_bayes_classifer(X_train, X_test, y_train)
         metrics(y_test, y_predict)

24.91 Seconds to train Naive Bayes...

True positive 1066
False positive 102
False negative 67
True positive 989

Precision 91.27 %
Recall 94.09 %
False positive rate 9.35 %
Accuracy 92.4 %
F-measure 92.66 %
```

It is possible to notice that for the SVM, the numbers got much better with a smaller time for training. Here the accuracy is almost 97% with a false positive rate a little bit above 4%.

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

```
In [19]: y_predict = train_svm_classifer(X_train, X_test, y_train)
         metrics(y_test, y_predict)

32.76 Seconds to train SVM...

True positive 1107
False positive 44
False negative 26
True positive 1047

Precision 96.18 %
Recall 97.71 %
False positive rate 4.03 %
Accuracy 96.85 %
F-measure 96.94 %
```

5 Further improvements

For further improvements of this algorithm it would be interesting to better tune the parameters of the classifiers, try to discard features that does not add relevant information, therefore simplify the problem, and also, instead of just check if a word is in the file or not, count its number of occurrences in each file.

Besides, an improvement can be done on the features extraction, the algorithm presented here is using just the most important word of each line of the document, for example `touchscreen` and `main`, and discarding words less relevant (`android`, `hardware`, `intent` and `action`) as the lines shown below. It would be interesting to try using all the words and, after that, study if this words can add any useful information to the classifier or are irrelevant.

```
feature::android.hardware.touchscreen
intent::android.intent.action.MAIN
```