Machine Learning for Malware Analysis

Simone Faricelli 1647406

November 2018

1 Introduction

The purpose of this project is to develop a valuable tool to recognize and classify different kind of Malware Application, running on AndroidOS, through the use of several Machine Learning Classifiers.

In order to train the above-mentioned classifiers, we're taking advantage of the Drebin Dataset, which provide us 5,560 malevolent files from 179 different malware families, collected in the period of August 2010 to October 2012.

From those applications we are extracting features through the manifest.xml file and the disassembled code.

2 A Classification Problem

Classification belongs to the category of supervised learning where the targets (or labels or categories) are also provided with the input data and it is, in fact, the process of predicting the class of given data points. Classification predictive modeling is the task of approximating a target function (f) from input variables (X) to discrete output variables (y).

The input variable chosen for this project is a list of vectors, each of which specify, for each malware, the features of the application itself, previously retrieved from a dictionary of all the features.

As expected, the output variable is a list containing the category of each malware.

3 Dataset

3.0.1 Selective Extraction

The Drebin Dataset mentioned above contains not only examples of features of malware files, but also a huge quantity of non-malware applications, which are not useful for our classification goal.

A selective extraction is therefore performed through a python script, picking only the malware files, specified in the csv file, from the 130.000 samples.

```
In [2]: import csv
        from zipfile import ZipFile
        from tqdm import tqdm

        csvName = 'sha256_family.csv'
        malware = list()
        with open('dataset/' + csvName) as csvFile:
            reader = csv.reader(csvFile)
            next(reader)
            for row in reader:
                malware.append(row[0])

        print("Extracting ",len(malware)," files from the Drebin Dataset")

        with ZipFile('drebin.zip') as myzip:
            for i in tqdm(range(len(malware))):
                myzip.extract('drebin/' + csvName)
                myzip.extract('drebin/feature_vectors/' + malware[i])
```

```
Extracting  5560  files from the Drebin Dataset


100%|| 5560/5560 [01:58<00:00, 47.09it/s]
```

4 Development

4.0.1 Libraries and Global Variables

In the following section the required libraries are imported and the global variables initialized. For this project, wide use of the python library for machine learning called “scikit-learn” is done. That particular library needs to be installed first.

```
In [3]: %matplotlib notebook
        import os, sys, csv, time, json
        import pandas as pd
        import seaborn as sns
        import matplotlib.pyplot as plt
        from sklearn.model_selection import train_test_split
        from sklearn.naive_bayes import MultinomialNB, GaussianNB
        from sklearn.metrics import classification_report
        from sklearn.svm import LinearSVC, SVC
        from sklearn.ensemble import RandomForestClassifier
        from sklearn.metrics import accuracy_score, log_loss
        from urllib.parse import urlsplit

        datasetFolder = 'drebin/'
        featuresFolder = 'feature_vectors/'
        csvName = 'sha256_family.csv'
        dictionaryName = 'dictionary.json'
```

4.0.2 Defining Malware Dataset & Dictionary

The provided dataset is composed of examples of malware and non-malware data, but in order to train the classifier the focus is moved onto the malevolent files. In the block below, those two categories are being separated using the cvs file as reference.

```
In [4]: # List of all the files in debrin dataset
        dataset = os.listdir(datasetFolder + featuresFolder)

        # Define list for malware and non-malware files
        malware = list()

        # Create malware dictionary from the csv file, with the file name and the type of each o
        with open(datasetFolder + csvName) as csvFile:
            reader = csv.reader(csvFile)
            next(reader)
            malwareDictionary = {row[0]: row[1] for row in reader}

        # Separate the file names
        for i in dataset:
            if i in malwareDictionary:
                malware.append(i)

        print('Number of malwares in the dataset:\t', len(malware))
```

```
Number of malwares in the dataset:         5560
```

4.0.3 Feature and Malware Categories

For a matter of knowledge, the features contained in the dataset and written into the files, are being collected, listed and printed. Also empty files are removed from the feature analysis.

```
In [5]: # Collect all the features from the files of the dataset
        categoryList = list()
        for file in malware:
            with open(datasetFolder + featuresFolder + file) as f:
                content = f.readlines()
                for line in content:
                    try:
                        category, string = line.split('::')
                        if category not in categoryList:
                            categoryList.append(category)
                    except:
                        print("Empty file: ", file)

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

```
Empty file:  3de513a148400b457dd8d8fa9238804db3ec031a0b526d4a04b77e5112aa2dcf
Empty file:  6c6eeed1b91913db0d6232edb1979c67d6fb48ca3da4f83dc49fb565a4e5f4fe
Empty file:  76e91e1f9cc3422c333e51b65bb98dd50d00f1f45a15d2008807b06c125e651a
Empty file:  df2c357f513c270cd1d06418e4eaf64aeb6b2d947149e83ed4f42c88286b76a7
Empty file:  f6239ba0487ffcf4d09255dba781440d2600d3c509e66018e6a5724912df34a9
The features can be divided in 10 different sets:
         api_call
         feature
         url
         service_receiver
         permission
         call
         intent
         real_permission
         activity
         provider
```

As for the features, also malware categories are being collected from the malware dictionary created above, and taken into account only if the number of their samples is greater that 20. From this list, the output vector for the training will be generated.

In [6]:
```
# Collect all the classes from the files of the dataset
classes = []
counter = []
for key, value in sorted(malwareDictionary.items()):
    if not value in classes:
        classes.append(value)
        counter.append(1)
    else:
        counter[classes.index(value)] += 1

# Considering only classes with more than 20 samples
classes.append("Other")
counter.append(0)
classesNew = list()
counterNew = list()

for i, el in enumerate(counter):
    if el > 20:
        classesNew.append(classes[i])
        counterNew.append(el)
    else:
        counter[len(counter)-1] += counter[i]

classes = classesNew
counter = counterNew

for i in range(len(classes)):
    print('\t', '{:>3}'.format(counter[i]), " ", classes[i])
```

```
339   GinMaster
613   Opfake
925   FakeInstaller
 27   Boxer
 29   Jifake
667   DroidKungFu
 41   SMSreg
 58   Gappusin
147   Kmin
 61   FakeRun
625   Plankton
 28   Hamob
132   FakeDoc
 91   Adrd
 37   Yzhc
 69   Glodream
330   BaseBridge
152   Iconosys
 70   ExploitLinuxLotoor
 59   SendPay
 92   Geinimi
 81   DroidDream
 43   Imlog
 69   MobileTx
775   Other
```

4.0.4 Chosen Ones

In order to reduce the input vector dimension and fine-tune the training, only a few features are chosen to be extracted. Deeper consideration will be done later about this specific selection.

```
In [7]: chosenFeatures = ["permission", "api_call", "service_receiver"]
```

4.0.5 Features Extraction

The following function has the purpose to, given as input the feature name and its subsequent string, identify the best function to extract the feature and return as output a vector of words which will be used to build the dictionary of words or the vector of features.

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.

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

The alghoritm perform that way also because the firsts are common words which do not improve the training results, because they add no more information to the input vector.

```
In [8]: def selfExtraction(feature, string):
            def extractUrl(string):
                try:
                    baseUrl = "{0.scheme}://{0.netloc}/".format(urlsplit(string))
                    if len(baseUrl) > 10:
                        return [baseUrl]
                except:
                    return None

            def extractApiCall(string):
                try:
                    string = string.replace(';->', '/')
                    apiCall = string.split('/')
                    return apiCall
                except:
                    return None

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

            def extractPermission(string):
                try:
                    permission = string.split('.')[-1].lower()
                    return [permission]
                except:
                    return None

            def extractCall(string):
                try:
                    call = string.lower()
                    return [call]
                except:
                    return None

            def extractActivity(string):
                try:
                    activity = string.split('.')[-1].lower()
                    if activity[0] == ":":
                        activity = activity[1:]
                    return [activity]
                except:
                    return None

            def extractIntent(string):
```

```
        try:
            intent = string.split('.')[-1].lower()
            return [intent]
        except:
            return None

    def extractServiceReceiver(string):
        try:
            serviceReceiver = string.split('.')[-1].lower()
            return [serviceReceiver]
        except:
            return None

    def extractProvider(string):
        try:
            provider = string.split('.')[-1].lower()
            if "\\" in provider:
                provider = provider[:-2]
            return [provider]
        except:
            return None

    switcher = {
        'url': extractUrl,
        'api_call': extractApiCall,
        'feature': extractFeature,
        'permission': extractPermission,
        'real_permission': extractPermission,
        'call': extractCall,
        'activity': extractActivity,
        'intent': extractIntent,
        'service_receiver': extractServiceReceiver,
        'provider': extractProvider
    }
    return switcher[feature](string)
```

4.0.6 Words Indices Vector

The network is fed with a set of vectors, which elements are the indices of each word belonging to the specific file, mapped from a dictionary of all the words obtained from the feature extraction of the entire dataset of files.

The dictionary is previously generated through the same function, as shown below.

For instance, considering the file

```
612aa197794bc4c88bd7dbed41a9b13b9969befb645761fa384b1a567a50ceee
```

Its own Word Vectore will be

```
[0, 25, 26, 126, 111, 22, 3, 7, 8, 10, 11, 33, 19]
```

```
In [9]: def processFile(file, dictionary, outputCreation = False, dictionaryCreation=False):
            # List of words of each file
            words = list()
            # Read line by line of the file
            with open(datasetFolder + featuresFolder + file) as f:
                content = f.readlines()

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

                    wordList = None
                    if category in chosenFeatures:
                        wordList = selfExtraction(category, string)

                    # If able to extract feature from line
                    if wordList != None:
                        for word in wordList:
                            word = word.replace('\n', '')
                            # If flagged to create the dictionary
                            if dictionaryCreation:
                                if not word in dictionary:
                                    index = len(dictionary)
                                    dictionary[word] = index
                            else:
                                index = dictionary[word]
                                if index not in words:
                                    words.append(index)
                #If flagged to create the output vector of categories
                if outputCreation:
                    if malwareDictionary[file] in classes:
                        y.append(classes.index(malwareDictionary[file]))
                    else:
                        y.append(classes.index("Other"))

            return words
```

4.0.7 Dictionary Generation and Loading

The above-mentioned dictionary is, at first, searched into the main folder and, if not found, generated and stored in a json file for a better readablity and due to its huge size.

```
In [10]: if os.path.isfile(dictionaryName):
             print('Dictionary file found!')
             with open(dictionaryName) as d:
                 dictionary = json.load(d)
             d.close()

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

             # Define the dictionary
             dictionary = {}

             # Colect words for malware
             print('Creating dictionary')
             for i in range(len(dataset)):
                 processFile(dataset[i], dictionary, dictionaryCreation=True)

             with open(dictionaryName, 'w+') as d:
                 d.write(json.dumps(dictionary, sort_keys=True, indent=4))
             d.close()


             print('\nDictionary saved to file!')

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

```
Dictionary file not found!
Creating dictionary

Dictionary saved to file!
Dictionary size:  1733
```

4.0.8 Building Input and Output

Through the above-mentioned function used to extract features from the malware files, the actual input (X) and output (y) that will feed the network are finally built.

In order to have a homogeneus size of input, the one-hot encoded version of the word vectors are built, starting from the length of the dictionary to generate a list containing only zeros, and replacing that values with ones at the indices specified by the output of the feature extraction function. One vector for file is then appended to the input list.

Moreover, for each file, the category index of that malware is appended to the output list, setting to True the boolean variable "outputCreation".

```
In [11]: #List of features
         X = list()
         #List of categories
         y = list()
         # Construct the vector of features and output vector at the same time
         def featuresExtraction(file, dictionary):
             indices = processFile(file, dictionary, outputCreation=True)
             feat = [0] * len(dictionary)
             for i in indices:
                 feat[i-1] = 1

             return feat
         # Extract features and append to the list of features
         print('Extracting Features')
         for i in range(len(malware)):
             feat = featuresExtraction(malware[i], dictionary)
             X.append(feat)
         print('Extraction Complete')
```

```
Extracting Features
Extraction Complete
```

4.0.9 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]: # Split randomly the dataset into train and evaluation
         X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

5 Classifiers

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

Follows a brief description of the above-mentioned classifiers, highlighting the pros and cons for each method.

5.0.1 Naive Bayes

Naive Bayes is a probabilistic classifier inspired by the Bayes theorem under a simple assumption which is the attributes are conditionally independent.

The classification is conducted by deriving the maximum posterior which is the maximal P(Ci | X) with the above assumption applying to Bayes theorem. This assumption greatly reduces the computational cost by only counting the class distribution. Even though the assumption is not valid in most cases since the attributes are dependent, surprisingly Naive Bayes has able to perform impressively.

Naive Bayes is a very simple algorithm to implement and good results have obtained in most cases. It can be easily scalable to larger datasets since it takes linear time, rather than by expensive iterative approximation as used for many other types of classifiers.

Naive Bayes can suffer from a problem called the zero probability problem. When the conditional probability is zero for a particular attribute, it fails to give a valid prediction. This needs to be fixed explicitly using a Laplacian estimator.

5.0.2 Support Vector Machine

SVM or Support Vector Machine is a linear model for classification and regression problems. It can solve linear and non-linear problems and work well for many practical problems. The idea of SVM is simple: The algorithm creates a line or a hyperplane which separates the data into classes.

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.

SVM performs similarly to logistic regression (really well) when with clear margins of linear separation.

On the contrary, it does not perform well when we have large data set, because the required training time is higher, and when the dataset has more noise, which means that target classes are overlapping.

Tuning Parameters?

5.0.3 Random Forest

Random Forest is an example of an ensemble method, in which we combine multiple machine learning algorithms to obtain better predictive performance. We'll run multiple models on the data and use the aggregate predictions, which will be better than a single model alone.

The idea behind a Random Forest is actually pretty simple: We repeatedly select data from the data set (with replacement) and build a Decision Tree with each new sample. It is important to note that since we are sampling with replacement, many data points will be repeated and many won't be included as well. This is important to keep in mind when we talk about measuring error of a Random Forest. Another important feature of the Random Forest is that each node of the Decision Tree is limited to only considering splits on random subsets of the features.

One big advantage of Random Forest is, that it can be used for both classification and regression problems, which form the majority of current machine learning systems.

On the contrary, Random Forest produces results not easy to visually interpret and it can overfit with some datasets with noisy classification/regression tasks.

6 Training and Results

6.0.1 Training Function

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.

```
In [13]: y_predict = list()
         classifiers = [
                 MultinomialNB(),
                 SVC(kernel="linear", probability=True),
                 RandomForestClassifier(n_estimators=100, max_depth=100)
         ]
         log_cols=["Classifier", "Accuracy", "Log Loss", "Time"]
         log = pd.DataFrame(columns=log_cols)

         for clf in classifiers:
             t=time.time()
             clf.fit(X_train, y_train)
             t2 = time.time()
             name = clf.__class__.__name__

             print("="*30)
             print(name)

             print('****Results****')
             train_predictions = clf.predict(X_test)
             y_predict.append(train_predictions)
             acc = accuracy_score(y_test, train_predictions)
             print("Accuracy: {:.4%}".format(acc))

             train_predictions = clf.predict_proba(X_test)
```

```
            ll = log_loss(y_test, train_predictions)
            print("Log Loss: {}".format(ll))

            timeSpent = round(t2-t, 2)
            print("Time to Train: {} s".format(timeSpent))

            log_entry = pd.DataFrame([[name, acc*100, ll, timeSpent]], columns=log_cols)
            log = log.append(log_entry)

        print("="*30)
```

```
==============================
MultinomialNB
****Results****
Accuracy: 85.8813%
Log Loss: 1.1335253415771336
Time to Train: 0.6 s
==============================
SVC
****Results****
Accuracy: 95.7734%
Log Loss: 0.20580616263825374
Time to Train: 48.0 s
==============================
RandomForestClassifier
****Results****
Accuracy: 95.6835%
Log Loss: 0.27335448783949584
Time to Train: 2.97 s
==============================
```

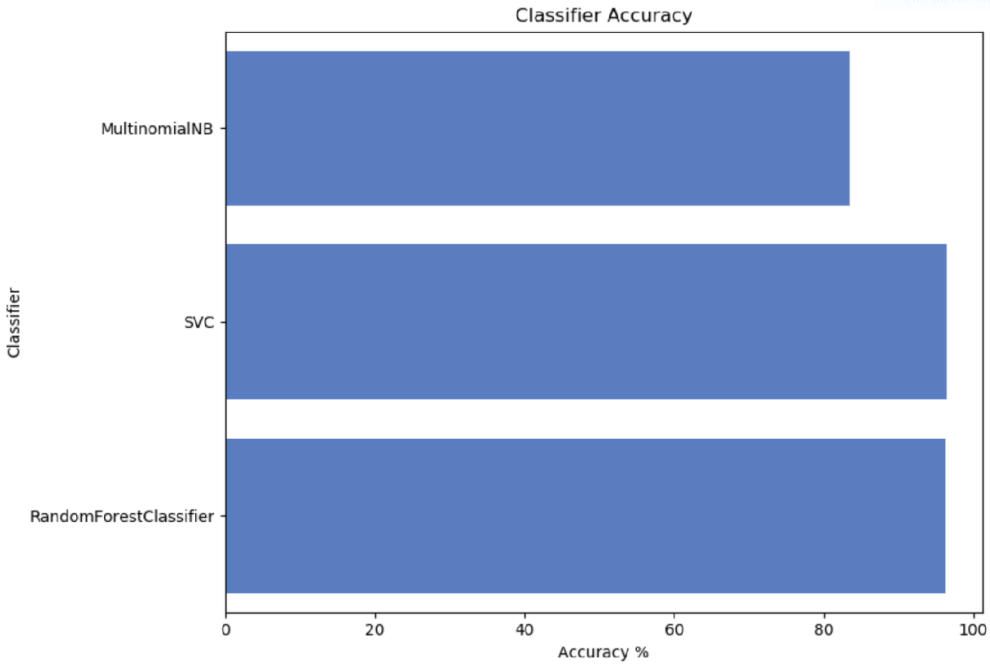
6.0.2 Graphical Results

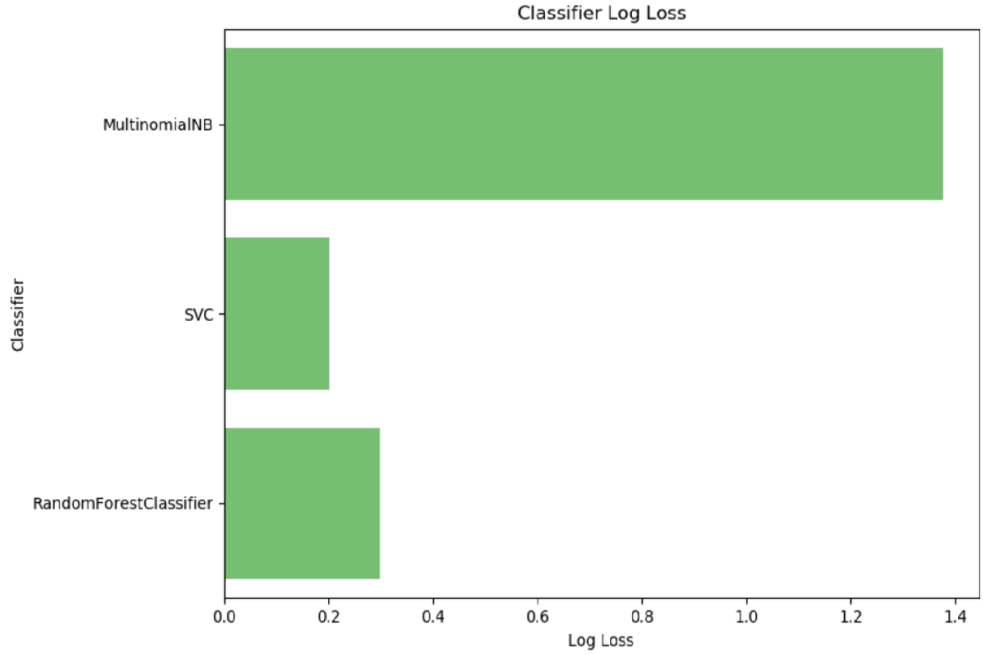
```
In [ ]: sns.set_color_codes("muted")
        sns.barplot(x='Accuracy', y='Classifier', data=log, color="b")

        plt.xlabel('Accuracy %')
        plt.title('Classifier Accuracy')
        plt.show()
```

```
In [ ]: sns.set_color_codes("muted")
        sns.barplot(x='Log Loss', y='Classifier', data=log, color="g")

        plt.xlabel('Log Loss')
        plt.title('Classifier Log Loss')
        plt.show()
```


Classifier Accuracy
MultinomialNB
SVC
RandomForestClassifier
Classifier
0
20
40
60
80
100
Accuracy %


Classifier Log Loss
MultinomialNB
SVC
RandomForestClassifier
Classifier
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
Log Loss


```
In [ ]: sns.set_color_codes("muted")
        sns.barplot(x='Time', y='Classifier', data=log, color="r")

        plt.xlabel('Time')
        plt.title('Classifier Time to Train')
        plt.show()
```

7 Final Considerations

7.0.1 Choosen Features and Classifiers

A wide range of attempts has been made in order to find the categories that performs better with this classification problem. Almost all the possibile combinations between the features have been tried, highlighting that, category such as “feature” or “provider”, which generates alone a dictionary containing less than 50 words, do not increase the accuracy when combined with others category, therefore they’have been deleted from the list.

On the other side, there are categories such as “permission” or “api_call” that perform amazing results all alone (an average of 90% of accuracy with the three classifiers). It has been easy to maximize the perfomance starting from those categories.

In all of the attempts, Random Forest and SVM performed a lot better than Naive Bayes, so the final goal of the task had been maximizing the perfomance of that specific classifiers. That combination has been found in the categories “permission”, “service_receiver” and “activity”.

7.0.2 From the Graph

The graphs showed above highlight streghts and weaknesses of each classifier, which is an extension of what it has been already told.

Besides the fact that Naive Bayes performs worse than the other two in most of the cases, also the time spent to train the classifier needs to be considered. It's clear that the Support Vector Machine is the slowest one and that's because its fit time complexity is more than quadratic with the number of samples, which makes it hard to scale to huge dataset (in this case, samples number equal to the number of malware files).

From the results of this project it's cristal clear that, among the studied three, the classifier that best fit this classification problem is the Random Forest.