Machine Learning for Malware Analysis

Biagio Montaruli - biagio.hkr@gmail.com

16 November 2018

1 Introduction

Nowadays mobile devices are used everywhere in our daily life providing many valuable services, but on the other hand we are assisting to a rapid growth of malware designed for mobile devices. One of the most widespread operating system for mobile devices is Andorid which nowadays not only powers smartphones but also tablets, TVs and even wearable and IoT devices [2]. Due to the great threats and damage that malware can cause to users, Android malware detection has become increasingly important in cyber security and lots of studies and researches have been carried out in order to find new methods and approaches to discover malicious patterns in mobile applications. In this context, Machine Learning (ML) has became a very successful way to detect and classify malware. This is because, using standard ML classification algorithm is possible to automatically learn the characteristics that distinguish malware and automate the detection process [2]. In addition, as opposed to the standard signature-based approaches which can be evaded by attackers using code obfuscation or repackaging, ML methods allow for a very scalable detection and the possibility to adapt different malware variants. This reports describes an innovative approach to malware detection and family classification based on the Naive Bayes classification algorithm. Moreover, it also presents the evaluation procedure and the obtained results on the DREBIN dataset.

2 Android malware and the DREBIN project

One of the most important part in the machine learning projects is understanding the data you are working with and how it relates to the task you want to solve [3]. For this reason this section gives an overview on the main features of Android applications and describes the structure of the DREBIN dataset.

2.1 Overview of Android applications

An Android application consists of an Android apk file. Each apk file is a zipped file that contains the application source code, resources, assets, and manifest file. The application source code is encoded as dex files (i.e., Dalvik Executable Files) that can be interpreted by the Dalvik or the more recent ART Virtual Machines. Android applications are also characterized by many components of differing types, which are the essential building blocks of the application. Each component has an entry point that can be used by the system or an user to interact with the application. In addition, components can also be used by applications to communicate each other. For these reason, in malware analysis it is critical to analyze the components of an application [2]. The fundamentals component types of Android applications are:

- Activities: they serve as the entry point for a user's interaction with an app.
- Services: they are tasks that are executed in background without providing a user interface.
- Broadcast receivers: these components are used by Android apps to send and receive broadcast messages to/from the Android system and other apps when an event of interest occurs.
- Content providers: they are components that are used by an application to manage data sets and share data with other apps.

Feature engineering is the one of most important part for training a machine learning model and a right choice of features is a key point in malware analysis. In general, features extraction depends on two factors:

1. The target operating system. For example, in Android it is possible to easily extract most of the features of an application from its AndroidManifext.xml file.
2. The type of analysis:
    - Static: features are extracted from the source code of the application. In Android, the features that can be extracted from the apk of an application are:
        - Components: activities, services, content providers and broadcast receivers. Collecting the names of these components can to help to identify some variants of well-known malware families. In fact, as descibed in [2] the DroidKungFu family shares the name of particular services.

  - Intents: data structures exchanged as asynchronous messages that are used for inter-process and itra-process communication (IPC). As described in [1], intents are also used by some malware families such as GingerMatser and DroidKungFu. Malware that belong to these families use the BOOT COMPLETED intent to trigger malicious activity directly after rebooting the smartphone.
  - Hardware features: for security reasons if an Android app wants to access to some hardware components such as the GPS, Bluetooth or sensors, then those features must be declared in the manifest file. It is important to point out that requesting certain hardware components may have security implications, as the use of certain combinations of hardware features often reflects harmful behavior.
  - Permissions: Android system use a permission mechanism to protect the privacy of users. In fact, in order to access sensitive data (e.g. SMS, calls, ...), system features (e.g. camera) and restricted APIs, an app must request permission in the manifest file. Since malware usually tends to request a special set of permissions it is a good practice to consider permission when doing feature extractions. Sometimes is also useful to distinguish, among all permissions, those that are effectively used by the application (Used permissions).
  - Restricted APIs: special APIs related to sensitive data access are protected by permissions. Sometimes they are used by malware to try to gain root access.
  - Suspicious APIs: they are a special set of APIs that can lead to malicious behavior without requesting permission. An example of suspicious APIs are cryptography functions in the Java library that can be used by malware for code obfuscation.
- Dynamic: Features that can be extracted by executing the application in a secure environment and logging at the execution traces. In our case, the main dynamic features that can be extracted from an Android app are resource consumptions, system calls and download patterns such as URLs and network addresses.

2.2 The DREBIN dataset

DREBIN is a lightweight method for detection of Android malware that enables identifying malicious applications directly on the smartphone by performing a broad static analysis [1]. Malware detection and classification is performed using the Support Vector Machine (SVM) algorithm on a dataset (called the DREBIN dataset in the following) consisting of 123,453 applications (samples) among which 5,560 are malware.

Features, which are extracted from the manifest file and from the disassembled dex code, are divided into 8 categories (set) for which is also indicated the prefix (or the list of prefixes) used in the dataset to label the features of each application:

- From the manifest:
  - S1: Requested hardware components (feature)
  - S2: Requested permissions (permission)
  - S3: App components (activity, service_receiver, provider, service)
  - S4: Filtered intents (intent)
- From the disassembled code:
  - S5: Restricted API calls (api_call)
  - S6: Used permissions (real_permission)
  - S7: Suspicious API calls (call)
  - S8: Network addresses (url)

The DREBIN dataset, which is publicly available, consists of a directory named *features_vector* and a CSV file called *sha256_family.csv.* For each app there is a file in the *features_vector* directory whose name is the SHA256 hash of the related apk and it contains the list of all the extracted features. Each feature is composed by a prefix and a value. The prefix represent the set to which the feature belongs to. As for the *sha256_family.csv* file, it contains the list of malware. Each line contains the SHA256 hash of the apk and the family (represented by a label) to which the malware belongs to. The format adopted for each line is *<sha256_apk>,<family_label>*

3 Naive Bayes classifier

After analyzing the structure of the DREBIN dataset, the next step is to define a classification problem for malware analysis. This means defining the target function $f : X \to Y$ that, given in input a sample, will return the class to which the sample belongs to. The target function will be used for:

- Malware detection (binary classification problem): given a sample as input, determine if it is a malware or a benign app.
- Malware family classification (multi-class classification problem): given a malware application as input, determine the family to which it belongs to.

Among all supervised machine learning algorithm, the Naive Bayes classifier has been chosen for the following reasons:

1. Naive Bayes models are a group of extremely fast and simple classification algorithms that are often suitable for very high-dimensional datasets. In our case it is well-suited since the DREBIN dataset consists of many samples that are in turn characterized by many different features [3].

2. Naive Bayes algorithms (especially the Multinomial version) are often used is in text classification, where the features are related to word counts or frequencies within the documents to be classified. In our case both malware detection and family classification can be seen as text classification problem: each application consists of a file containing a list of features; so the application can be considered as a document and its features as the document's words.

Naive Bayes classifiers are based on the Bayes's rule, which is an equation describing the relationship of conditional probabilities of statistical quantities:

$$P(A \mid B) = \frac{P(B \mid A)P(B)}{P(A)}$$

Given the target function $f : X \to V$, $V = \{v_1, ..., v_k\}$, the dataset D and a new sample to classify $x \notin D$, the most probable class label to assign to x can be expressed using the Bayes Optimal classifier:

$$v_{OB} = \arg\max_{v_j \in V} \sum_{h \in H} P(v_j \mid x, h_i)P(h_i \mid D)$$

Bayes Optimal classifier provides the best results because it maximizes the probability that a new instance is classified correctly; however it is not a practical method when the hypotesis space is large. For this reason, in practical machine learning problems the Naive Bayes classifier is adopted since it approximate the solution by using the conditional independence assumption. Given the target function $f : X \to V$ and a dataset D, the most probable class value v_{MAP} for a new instance x, described by the tuple of attribute values $(a_1, a_2, ..., a_n)$, can be evaluated as:

$$\begin{aligned}
v_{MAP} &= \arg\max_{v_j \in V} P(v_j \mid x, D) \\
&= \arg\max_{v_j \in V} P(v_j \mid a_1, a_2, \ldots, a_n, D) \\
&= \arg\max_{v_j \in V} \frac{P(a_1, a_2, \ldots, a_n \mid v_j, D)P(v_j \mid D)}{P(a_1, a_2, \ldots, a_n \mid D)} \\
&= \arg\max_{v_j \in V} P(a_1, a_2, \ldots, a_n \mid v_j, D)P(v_j \mid D)
\end{aligned}$$

Then, since the naive Bayes classifier is based on the simplifying assumption that the attribute values are conditionally independent given the target value, we have: $P(a_1, a_2, \ldots, a_n \mid v_j, D) = \prod_{i=1}^{n} P(a_i \mid v_j, D)$ and so the most probable class value can be approximated as:

$$v_{NB} = \underset{v_j \in V}{\arg\max}\, P(v_j \mid D) \prod_{i=1}^{n} P(a_i \mid v_j, D)$$

Where $P(v_j \mid D)$ and $P(a_i \mid v_j, D)$ terms are estimated based on their frequencies over the training data.

One of the main application of Naive Bayes method is text classification. In this case the dataset is a set of documents and the features of each document are represented by the words contained in it. In text classification, the target function to learn is $f : Docs \rightarrow \{c_1, \ldots, c_k\}$ and we use a data structure called Vocabulary (V) to store all the words appearing in any document of the dataset. One of the most adopted representation of the documents is the so-called Bag of Words (BoW), where each document is represented as a n-dimensional feature vector (where $n = \mid V \mid$). There are two main methods for representing the features:

1. Boolean features: the value of the feature f_i is 1 if the word w_i is contained in the document otherwise f_i is 0.

2. Ordinal features: the value of the feature f_i is equal to the number of times the word w_i appears in the given document.

It is important to point out that the BoW representation has also some disadvantages. In fact, the words order is completely discarded and if a document only contain a small subset of the words in the vocabulary, then most entries in its feature vector are 0 (determining a waste of memory). The last problem can be avoided by using a sparse matrix data structure to store the document's features [3].

Given a new document d_i we can classify it by using the Naive Bayes method:

$$c_{NB} = \underset{c_j \in V}{\arg\max}\, P(c_j \mid D) P(d_i \mid c_j, D)$$

and by considering the Naive Bayed independence assumption we have that $P(d_i \mid c_j, D) = \prod_{i=1}^{n} P(a_i = w_k \mid c_j, D)$; where $P(a_i = w_k \mid c_j, D)$ is the probability that the word in position i is w_k. This probability value can be approximated by making the assumption that the word probability for one text position is independent from the words that occur in other positions $(P(a_i = w_k \mid c_j, D) = P(a_l = w_k \mid c_j, D), \forall i, l)$.

When training the Naive Bayes algorithm using the documents in the dataset D, the probability values $P(c_j \mid D)$ and $P(w_k \mid c_j, D)$ can be estimated in this way:

- $P(c_j \mid D) \rightarrow \hat{P}(c_j) = \frac{t_j}{|D|}, \forall$ class label t_j
  where t_j is the number of documents in class c_j

- $P(w_k \mid c_j, D) \rightarrow \hat{P}(w_k \mid c_j)$: it depends on the adopted representation method for the BoW:

  - Multi-variate Bernoulli Naive Bayes distribution: the feature f_k in the feature vector is 1 if w_k appears in the dataset, otherwise its value is 0. In this case we have:

    $$\hat{P}(w_k \mid c_j) = \frac{t_{i,j} + 1}{t_j + 2}$$

    where:
    $t_{i,j}$ is the number of documents in D of class c_j containing the word w_i;
    t_j is the number of the documents in D of class c_j;
    the number 1 and 2 are parameters for Laplace smoothing.

  - Multinomial Naive Bayes distribution: the feature f_k in the feature vector is equal to the number of times w_k appears in the dataset. In this case we have:

    $$\hat{P}(w_k \mid c_j) = \frac{TF_{i,j} + \alpha}{TF_j + \alpha|V|}$$

    where:
    $TF_{i,j}$ is the total number of times word w_i occurs in the doc d_j;
    TF_j is the total number words in the document d_j;
    α is the smoothing parameters (set to 1 for Laplace smoothing).

In conclusion, in order to classify a new document d_i using the Naive Bayes classifier we need to follow these steps:

1. Choose a method for representing the features of each document in the dataset (Bernoulli or Multinomial distribution).

2. Train the naive Bayes algorithm in order to learn $\hat{P}(c_j)$ and $\hat{P}(w_i \mid c_j)$.

3. Eliminate all the words of d_i that are not included in the dataset.

4. Estimate the best class value to assign to d_i using the following formula:

   $$v_{NB} = \arg\max_{v_j \in V} \hat{P}(c_j) \prod_{i=1}^{length(d)} \hat{P}(w_i \mid c_j)$$

4 Project implementation

The following section provides a detailed description about the implementation of the project. The project consists in the following files:

- `naive_bayes_test.py`: it contains a specific implementation of the Naive Bayes for text classification as described in Section 2. It has been built in a modular way using the `NaiveBayesText` Python class and contains different modules that allows the user to split the dataset into training and test sets, train the Naive Bayes algorithm with the training set and evaluate the performances computing the confusion matrix and other evaluation parameters such as the accuracy, precision, recall and f1-score.
- `util.py`: it is an utility Python script that allows the user to build and format the dataset from the files contained in the DREBIN project so that it can be managed by the `NaiveBayesText` class.
- `malware_detector.py`: it is a Python script that uses the DREBIN dataset to perform the malware detection by classifying the samples in the test set in malware or benign application. For the evaluation, the script makes a comparison between the Bernoulli and Multinomial Naive Bayes by plotting, for each method, the confusion matrix and the other evaluation parameters (accuracy, precision, recall and f1-score).
- `malware_family_classifier.py`: it is a Python script that selects randomly a subset of malwares from the DREBIN dataset and uses the Naive Bayes classifier to predict the malware family. The evaluation is quite similar to the malware detection, in fact the script makes a comparison between the Bernoulli and Multinomial Naive Bayes by plotting, for each method, the confusion matrix and the accuracy.

A more detailed description about the implementation of each modules of the project is provided in the following subsections.

4.1 `naive_bayes_test.py`

It contains the `NaiveBayesText` Python class that implements the Bernoulli and Multinomial Naive Bayes algorithms for text classification. It consists of different methods described in the following.

4.1.1 `NaiveBayesText` constructor

It is used to create an object of the `NaiveBayesText` class and it deals with the initialization of the class attributes such as

- `vocabulary`: a Python list used to store all the features read from the application files of the DREBIN dataset.
- `Pc`: it represents the probability $\hat{P}(c_j)$ that will be computed during the training phase by the `fit` method.
- `Pw_c`: it represents the probability $\hat{P}(w_i \mid c_j)$ that will be computed during the training phase by the `fit` method.

Its input parameters `perc` and `random_seed` will be used by the `split_dataset` to split dataset. In particular, `perc` represents the percentage of the samples in the dataset that will be included in the training set.

4.1.2 `split_dataset` method

This method receives as parameter the dataset built with the `build_dataset` function and randomly split it into training and test sets. This method returns `training_set`, `training_class`, `test_set`, `test_class` and *classes*.

`training_set` and `test_set` represent respectively the training and test set. They are list of dictionaries, where each dictionary represent an application and has the following format: $\{f_i, n_i\}$; where f_i is the i-th feature and n_i is the number of times the feature appears in the application file.

`training_class` and `test_class` are lists of class label related to the documents in the `training_set` and `test_set` respectively.

Finally, `classes` is a dictionary that describes for each class label (key of the dictionary) the number of documents included in it.

4.1.3 `fit` method

The `fit` method uses the input parameters `training_set` and `training_class` in order to train the Naive Bayes algorithm and compute `Pc` and `Pw_c`. Through the *mode* parameter the user can choose to use the Bernoulli or Multinomial variant of the Naive Bayes algorithm.

4.1.4 `classify` method

It is used to classify the samples in `test_set`, in fact it returns the list of predicted (the most probable) class labels for the samples in `test_set`.

4.1.5 `evaluate` method

This methods uses its input parameters `test_set` and `predicted_labels` to evaluate the classification performances by computing the accuracy of the model. In addition, it also uses the `confusion_matrix` method of the Scikit-learn library to compute the confusion matrix.

4.1.6 `classification_metrics` method

This method evaluates the performances of the Naive Bayes classifier using different classification metrics. First of all, given the predicted class labels (`predicted_labels`) and the true class labels (`test_class`), the method computes the number of True Positive (TP), True Negative (TN), False Positive (FP) and False Negative (FN). Then these values are used to compute the following classification metrics:

- `accuracy`: it is defined as: $\frac{TP+TN}{TP+TN+FP+FN}$ and in our context represents the number of applications that are classified correctly among all the applications.
- `precision`: it is defined as $\frac{TP}{TP+FP}$ and represents the ability of the classifier to avoid false positives that, in our case, is the number real malware (TP) among those that have been classified as malware (TP + FP).
- `recall`: it is defined as $\frac{TP}{TP+FN}$. Moreover, it describes the ability of the classifier to avoid false negatives and in our case represents the number of correctly detected malware (TP) among those that are in the dataset (TP + FN).
- `f1-score`: it is defined as $2 \cdot \frac{precision \cdot recall}{precision + recall}$ and can be interpreted as a weighted average of precision and recall.

4.2 `util.py`

The `util` Python module contains the `build_dataset` function which receives as input the path of the 'features_vector' directory of the DREBIN dataset, the path of the 'sha256_family.csv' CSV file, the number of samples to extract (`max_samples`) and the file parsing mode (`mode`) in order to parse the file in 'features_vector' directory and build the dataset. If `mode` is set to `"bin_class"`, then benign applications are labeled as *"safe"* instead the label *"malware"* is assigned to malware applications; so `mode = "bin_class"` should be used in case of malware detection (binary classification). On the other hand, to perform malware family classification, the user should use `mode = "skip_safe"` so that only the malware applications are inserted in the dataset. The `dataset`, which is returned by the `build_dataset` function, is represented as a set (dictionary) of dictionaries with the following structure:
{<sha-app$_i$> : {"target": <class-label>, "features": {$f_{i,k}$: $n_{i,k}$}}}.

4.3 `malware_detector.py`

The `malware_detector` module uses both the Bernoulli and Multinomial Naive Bayes classifier implemented in the `NaiveBayesText` class to perform malware detection:

1. Import the needed packages:

```
import naive_bayes_text as nb
from util import build_dataset
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report
import seaborn as sns; sns.set()
```

2. Initialize the parameters for the `build_dataset` function and the global variables:

```
dataset_path = "../drebin/feature_vectors/"
family_labels_path = "../drebin/sha256_family.csv"
feature_list = ['permission', 'call', 'api_call', 'intent']
eta = 1
types = ["bernoulli", "multinomial"]
mode = "bin_class"
rseed = 20
num_samples = 10000

DEBUG = False
num_iter = 4
```

3. Create a new instance of the `NaiveBayesText` class:

```
nb_classifier = nb.NaiveBayesText(debug_mode=False, perc=2/3, rand_seed=rseed)
```

4. Build the dataset:

```
dataset = build_dataset(dataset_path, family_labels_path, feature_list,
                        num_samples, eta, rseed, mode, True, False)
```

5. Initialize the `avg_perf` variable to collect the average values of performance metrics (accuracy, precision, recall and f1-score) for both the Naive Bayes types (Bernoulli and Multinomial):

```
avg_perf = {t: {"accuracy": 0, "precision": 0, "recall": 0, "f1-score": 0} for t in types}
```

6. Evaluate performances of the two types of Naive Bayes classifier `num_iter` times:

```
 for n in range(num_iter):
     print("TEST #{}".format(n+1))
         for t in types:
```

7. Randomly split the dataset into training set and test set:

```
train_set, train_class, test_set, \
test_class, classes = nb_classifier.split_dataset(dataset, random_split=True)
```

8. Train the classifier:

```
nb_classifier.fit(train_set, train_class, classes, t)
```

9. Classify samples (applications) in the test set:

```
estimated_class = nb_classifier.classify(test_set, classes)
```

10. Compute the accuracy and the confusion matrix:

```
accuracy, conf_matrix = nb_classifier.evaluate(estimated_class, test_class, classes)
```

11. Print the confusion matrix using `matplotlib` and `seaborn` Python libraries:

```
target_names = list(classes.keys())
plt.figure()
if t == "bernoulli":
    plt.title("Confusion matrix TEST #{} - Bernoulli Naive Bayes".format(n+1))
else:
    plt.title("Confusion matrix TEST #{} - Multinomial Naive Bayes".format(n+1))
    sns.heatmap(conf_matrix.T, square=True, annot=True, fmt='d', cbar=False,
                xticklabels=target_names, yticklabels=target_names)
    plt.xlabel('true label')
    plt.ylabel('predicted label')
    plt.draw()
    plt.tight_layout()
```

12. Compute the performance metrics and print them:

```
metrics = nb_classifier.classification_metrics(estimated_class, test_class, classes, class_map)
print("Performance evaluation:")
print("TP: {tp}, TN: {tn}, FP: {fp}, FN: {fn}".format(
      tp=metrics["TP"], tn=metrics["TN"], fp=metrics["FP"], fn=metrics["FN"]))

print("Precision: {:.2f}".format(metrics["precision"]))
print("Recall: {:.2f}".format(metrics["recall"]))
print("Accuracy: {:.2f}".format(metrics["accuracy"]))
print("F1-score: {:.2f}".format(metrics["f1-score"]))
```

13. Print the classification report using the `classification_report` method of Scikit-learn library:

```
print("Classification report using sklearn:")
print(classification_report(test_class, estimated_class, target_names=target_names))
```

14. Print the average value of evaluation parameters (`classification_report`):

```
    print("Average performances over {} tests:".format(num_iter))
    for t in types:
        if t == "bernoulli":
            print("> Multi-variate Bernoulli Naive Bayes")
        else:
            print("> Multinomial Naive Bayes")

        for metric in sorted(avg_perf[t]):
            avg_perf[t][metric] /= num_iter
            print("Average {m}: {value:.2f}".format(m=metric, value=avg_perf[t][metric]))
```

4.4 `malware_family_classifier.py`

The `malware_family_classifier` module uses both the Multi-variate and Multinomial Naive Bayes classifier implemented in the `NaiveBayesText` class to perform malware family classification:

1. Import the needed packages:

```
import naive_bayes_text as nb
from util import build_dataset
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report
import seaborn as sns; sns.set()
```

2. Initialize the parameters for the `build_dataset` function and the global variables:

```
dataset_path = "../drebin/feature_vectors/"
family_labels_path = "../drebin/sha256_family.csv"
```

```
feature_list = ['permission', 'call', 'api_call', 'intent']
eta = 1
types = ["bernoulli", "multinomial"]
mode = "skip_safe"
rseed = 10
num_samples = 4000

DEBUG = False
num_classes = 6
num_iter = 4
```

3. Create a new instance of the `NaiveBayesText` class:

```
nb_classifier = nb.NaiveBayesText(debug_mode=False, perc=2/3, rand_seed=rseed)
```

4. Build the dataset:

```
dataset = build_dataset(dataset_path, family_labels_path, feature_list,
                        num_samples, eta, rseed, mode, True, False)
```

5. Select from the dataset only the malwares that belong to the the most numerous `num_classes` classes:

```
labels = {c: 0 for c in set([dataset[app]["target"] for app in dataset])}
    for l in labels:
        labels[l] = len([app for app in dataset if dataset[app]["target"] == l])

    if num_classes > len(labels):
        num_classes = len(labels)

    sel_classes = [c for c, num in sorted(labels.items(), key=lambda x: x[1], reverse=True)[:num_classes]]

    parsed_dataset = {}
    for app in dataset:
        if dataset[app]["target"] in sel_classes:
            parsed_dataset[app] = dataset[app]
```

6. Initialize the `avg_accuracy` variable to collect the average accuracy for both the Naive Bayes types (Bernoulli and Multinomial):

```
avg_accuracy = {t: 0 for t in types} types}
```

7. Evaluate performances of the two types of Naive Bayes classifier: `num_iter` times:

```
for n in range(num_iter):
    print("TEST #{}".format(n+1))
        for t in types:
```

8. Randomly split the dataset into training set and test set:

```
train_set, train_class, test_set, \
test_class, classes = nb_classifier.split_dataset(parsed_dataset, random_split=True)
```

9. Train the classifier:

```
nb_classifier.fit(train_set, train_class, classes, t)
```

10. Classify malware in the test set:

```
estimated_class = nb_classifier.classify(test_set, classes)
```

11. Compute the accuracy and the confusion matrix:

```
accuracy, conf_matrix = nb_classifier.evaluate(estimated_class, test_class, classes)
```

12. Print the confusion matrix using the `matplotlib` and `seaborn` Python libraries:

```
target_names = list(classes.keys())
plt.figure()
if t == "bernoulli":
    plt.title("Confusion matrix TEST #{} - Bernoulli Naive Bayes".format(n+1))
else:
    plt.title("Confusion matrix TEST #{} - Multinomial Naive Bayes".format(n+1))
    sns.heatmap(conf_matrix.T, square=True, annot=True, fmt='d', cbar=False,
                xticklabels=target_names, yticklabels=target_names)
    plt.xlabel('true label')
    plt.ylabel('predicted label')
    plt.draw()
    plt.tight_layout()
```

13. Print the classification report using the `classification_report` method of Scikit-learn library:

```
print("Classification report using sklearn:")
print(classification_report(test_class, estimated_class, target_names=target_names))
```

14. Print the average accuracy:

```
print("Average accuracy over {} tests:".format(num_iter))
print("Bernoulli Naive Bayes -> accuracy = {:.2f}".format(avg_accuracy["bernoulli"]))
print("Multinomial Naive Bayes -> accuracy = {:.2f}".format(avg_accuracy["multinomial"]))
```

5 Evaluation and obtained results

This section describes the evaluation method and the obtained results by applying the Naive Bayes classifier implemented through the `NaiveBayesText` Python library to the DREBIN dataset.

5.1 Malware detection

Malware detection consists in a binary classification problem where, given a set of application files (samples), the Naive Bayes classifier should distinguish the malware from the benign applications. Malware detection is performed through the `malware_detector.py` script. It builds a dataset consisting of 10000 samples (files) chosen randomly from all the application files in the DREBIN project. As described in Section 2, the applications are characterized by different features, however in our evaluation we considered only *permission*, *call*, *api_call* and *intent*, since, as described in Table 4 of [1] they are the most common features with respect to all the malware families. The performances evaluation is repeated 4 times for each type of the Naive Bayes algorithm (Multi-variate Bernoulli and Multinomial) as described in Section 4. During each iteration the dataset is randomly split into training test (66.7 %) and test set (33.3 %). At the end, the scripts prints out the average results of the performance metrics so that the user can make a comparison between the Bernoulli and Multinomial variants of the Naive Bayes classifier. The following figure shows an example of running the script (for brevity, only the first iteration is shown):

```
**** Malware Detection ****
TEST #1
Number of docs in the training set: 6667
Number of docs in the test set: 3333

Using the Naive Bayes classifier with Multi-variate Bernoulli distribution
Classes:
{'safe': 6139, 'malware': 528}

Accuracy: 0.60
Confusion matrix:
[[ 269    6]
 [1330 1728]]
```

```
Performance evaluation:
TP: 269, TN: 1728, FP: 1330, FN: 6
Precision: 0.17
Recall: 0.98
Accuracy: 0.60
F1-score: 0.29

Classification report using sklearn:
             precision    recall  f1-score   support

       safe       0.17      0.98      0.29       275
    malware       1.00      0.57      0.72      3058

avg / total       0.93      0.60      0.69      3333

------------------------------------------------------------

Using the Naive Bayes classifier with Multinomial distribution
Classes:
{'safe': 6139, 'malware': 528}

Accuracy: 0.95
Confusion matrix:
[[ 209   66]
 [  89 2969]]

Performance evaluation:
TP: 209, TN: 2969, FP: 89, FN: 66
Precision: 0.70
Recall: 0.76
Accuracy: 0.95
F1-score: 0.73

Classification report using sklearn:
             precision    recall  f1-score   support

       safe       0.70      0.76      0.73       275
    malware       0.98      0.97      0.97      3058

avg / total       0.96      0.95      0.95      3333

------------------------------------------------------------
```

The following figure shows the average values of the performance metrics obtained by running the `malware_detector.py` script:

```
Average performances over 4 tests:
> Multi-variate Bernoulli Naive Bayes
Average accuracy: 0.59
Average f1-score: 0.27
Average precision: 0.16
Average recall: 0.98
------------------------------------------------------------
> Multinomial Naive Bayes
Average accuracy: 0.95
Average f1-score: 0.72
Average precision: 0.69
Average recall: 0.75
------------------------------------------------------------
```

As we can see, the Multinomial variant outperforms the Bernoulli variant since, except for the recall, it guarantees higher results. In particular, considering the Bernoulli variant, the average precision is a bit low meaning that the Naive Bayes classifier assigns many times the label *malware* to benign applications. On the other hand, in our context, it is better to have an high recall because in some cases having a benign application classified as malware can be worse than having a malware application classified as benign since in the first case the application can be further analyzed and it could be possible to discover the mistake of the classifier, but in the second case the application, being considered benign, could be installed causing damage to the user.

5.2 Malware family classification

Malware detection consists in a multi-class classification problem where, given a set of application files (samples), the Naive Bayes classifier should predict the class of each sample. Malware detection is performed through the `malware_family_classifier.py` script: it builds a dataset consisting of 2490 malware (benign applications are not considered) chosen randomly from all the malware that belong to the 6 most numerous families. As for the malware detection evaluation, in the family classification we considered only a sub-set among all the features (*permission*, *call*, *api_call* and *intent*). In addition, like the malware detection, the performances evaluation has been repeated 4 times for each type of the Naive Bayes algorithm (Multivariate Bernoulli and Multinomial) and during each iteration the dataset is randomly split into training test (66.7 %) and test set (33.3 %). At the end, the scripts prints out the average accuracy and makes a comparison between the Bernoulli and Multinomial variants of the Naive Bayes classifier. The following figure shows an example of running the script (for brevity, only the first iteration is shown):

```
*** Malware Family Classification ***
Number of docs in the parsed dataset: 2490
TEST #1
Number of docs in the training set: 1660
Number of docs in the test set: 830
```

```
Classes:
{'BaseBridge': 163, 'DroidKungFu': 300, 'Plankton': 293, 'GinMaster': 170, 'Opfake': 293, 'FakeInstaller': 441}

Using the Naive Bayes classifier with Multi-variate Bernoulli distribution

Accuracy: 0.939
Confusion matrix:
[[ 71   3   1   2   0   1]
 [  0 167   0   2   0   2]
 [  1   1 205   0  15   0]
 [  0   2   0  67   0   0]
 [  1   1  13   0 112   6]
 [  0   0   0   0   0 157]]

Classification report using sklearn:
               precision    recall  f1-score   support

   BaseBridge       0.97      0.91      0.94        78
  DroidKungFu       0.96      0.98      0.97       171
     Plankton       0.94      0.92      0.93       222
    GinMaster       0.94      0.97      0.96        69
       Opfake       0.88      0.84      0.86       133
FakeInstaller       0.95      1.00      0.97       157

  avg / total       0.94      0.94      0.94       830

------------------------------------------------------------

Using the Naive Bayes classifier with Multinomial distribution

Accuracy: 0.948
Confusion matrix:
[[ 71   1   1   5   0   0]
 [  0 167   0   3   0   1]
 [  0   0 206   0  16   0]
 [  0   0   0  69   0   0]
 [  1   0  15   0 117   0]
 [  0   0   0   0   0 157]]

Classification report using sklearn:
               precision    recall  f1-score   support

   BaseBridge       0.99      0.91      0.95        78
  DroidKungFu       0.99      0.98      0.99       171
     Plankton       0.93      0.93      0.93       222
    GinMaster       0.90      1.00      0.95        69
       Opfake       0.88      0.88      0.88       133
FakeInstaller       0.99      1.00      1.00       157

  avg / total       0.95      0.95      0.95       830

------------------------------------------------------------
```

The following figure shows the average results of the accuracy obtained by running the `malware_family_classifier.py` script.

```
Average accuracy over 4 tests:
Bernoulli Naive Bayes -> accuracy = 0.92
Multinomial Naive Bayes -> accuracy = 0.95
```

The obtained results are quite good and prove that both the Multinomial and the Bernoulli versions are quite effective in malware family classification.

6 Conclusions

This report describes the obtained results of malware detection and malware family classification on the DREBIN dataset by using the `NaiveBayesText` Python class which is an implementation of the Naive Bayes machine learning algorithm for text classification. In our case, malware detection and family classification can be regarded as text classification problems because the DREBIN dataset consists of 123,453 text files and each file is related to an application and contains the application features extracted through static analysis: requested hardware components, permissions, application components (activities, service receivers, providers, services), intents, APIs call and network addresses. For the evaluation the Naive Bayes classifier has been chosen since it is a machine learning algorithm that provides a very good trade-off between performances and complexity. In fact, since it is very simple and fast, it performs very well on very high-dimensional datasets and for this reason it is often used is in text classification. The performances evaluation is repeated 4 times for each type of the Naive Bayes algorithm (Multi-variate Bernoulli and Multinomial) considering a subset of the samples in the DREBIN dataset in order to make a comparison between the Multi-variate Bernoulli and Multinomial versions by computing the average performance metrics (accuracy, precision, recall and f1-score for malware detection and only accuracy for malware family classification). Experimental results demonstrate that in general the Naive Bayes classifier implemented through our method provides good performances but the Multinomial variant outperforms the Bernoulli variant (avg accuracy: 0.95 vs 0.59).

References

[1] Daniel Arp et al. "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket." In: *Ndss*. 2014.

[2] Chenglin Li et al. “Android Malware Detection based on Factorization Machine”. In: *CoRR* (2018).

[3] Andreas Müller and Sarah Guido. *Introduction to Machine Learning with Python*. O’Reilly Media, 2016.