Università della Svizzera italiana

Security ML **2023**

Student: Filippo Casari

Report for Assignment 2

1. Malware Detection

1.1. Understanding the Data

First, I would say that a lot of features are important to detect a malware in an Android application since certain permissions or using some APIs could trigger these features. It seems to me that the dataset is well structured.
Due to the fact that malicious apps frequently make unusual or suspicious API calls, the API call signatures are crucial characteristics for identifying Android malware. Similar to this, an application's requests for authorization can be a key sign of its potential for malicious behavior.
I am not an expert in Android applications, but ServiceConnector seems a very interesting feature that a malware can exploit. A ServiceConnection object is created when a component binds to a service using the bindService method. This object gets callbacks from the Android system when the connection to the service is made, when it is severed, or when an error occurs. Then, you can interact with the service by calling methods on its interface using the ServiceConnection object.
As I mentioned, since I do not have expertise in this field I cannot suggest other features. As I will explain in the following sections, my models work very well; no extra features are necessary to me indeed. Due to its nature, malicious apps may use this service to interact with a hidden or malicious service.

1.2. Evaluating Models

In this scenario, I would say that it is more impactful a false-negative since if we got a false-positive this should not be a problem for our system. In fact, in the worst case scanario, a benign application is killed by the detector. In contrast, if our detector is not able to detect properly a malicious app, we will allow the malicious program to run anyway without control of its behavior.
For this task since the class weights are not balanced, I used *balanced accuracy* as evaluation metric. Class 0 (benign): about 63.04 %, Class 1 (malware): about 36.96%.

1.3. Training Your Models

ELBO method

Before report the models used, I want to explain what I did before the classification.
Usually, before training the model I implemented a clustering algorithm to see whether the task was difficult for the classification. In particular, I first use ELBO method combined with KMeans

to see (without using true labels/y_true) which value of "k" (hyperparameter of Kmeans) optimizes the clustering between points of the dataset.
It turns out that k=2 is the best k meaning that the task is not tough for binary classification.

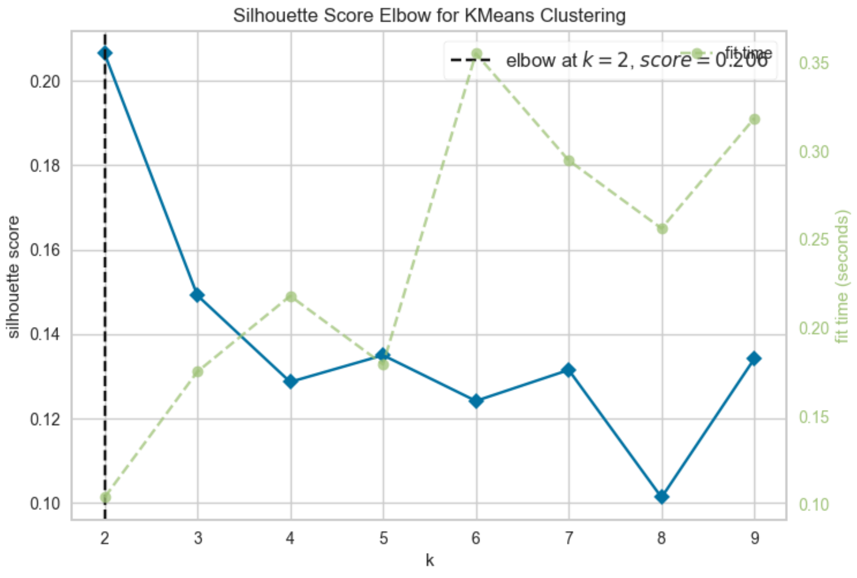


Figure 1: ELBO method

As one can see, both the maximum score and fit time are when k=2.

PCA

Moreover, I decided to use PCA to reduce the dimensionality of the dataset and apply on it the KMeans in order to plot and visualize the data. Furthermore, this allowed me to see if ELBO method was actually correct.

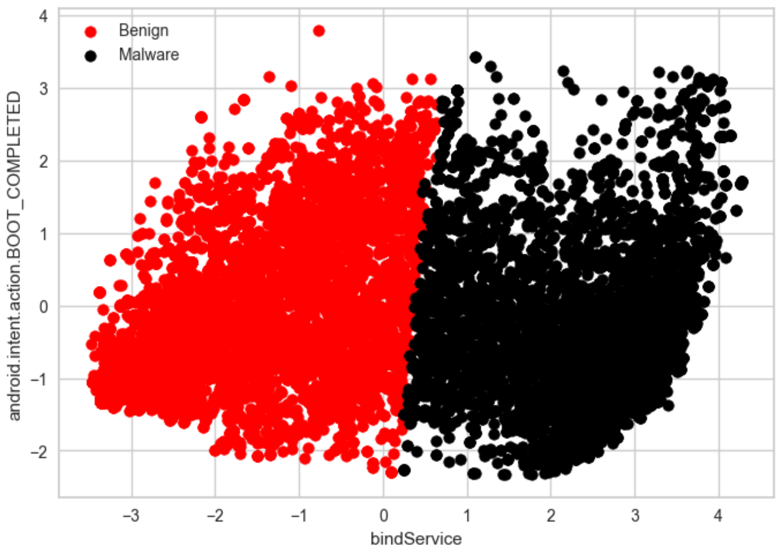


Figure 2: PCA

In fact, the data are almost linearly separable.

Preprocessing

I noticed that some rows on the 92nd column of the dataset were filled with **?**. Obviously, this could be a problem during the training. Consequetly, I replaced these values with zeros since I

presumed that the values themselves were unknown and the number of these occurencies was not crucial for the task.
Moreover, I converted Bs and Ss with 0s and 1s respectively to get a label number representation.
Finally, I report the correlation matrix of the dataset to see which features are more correlated with labels.

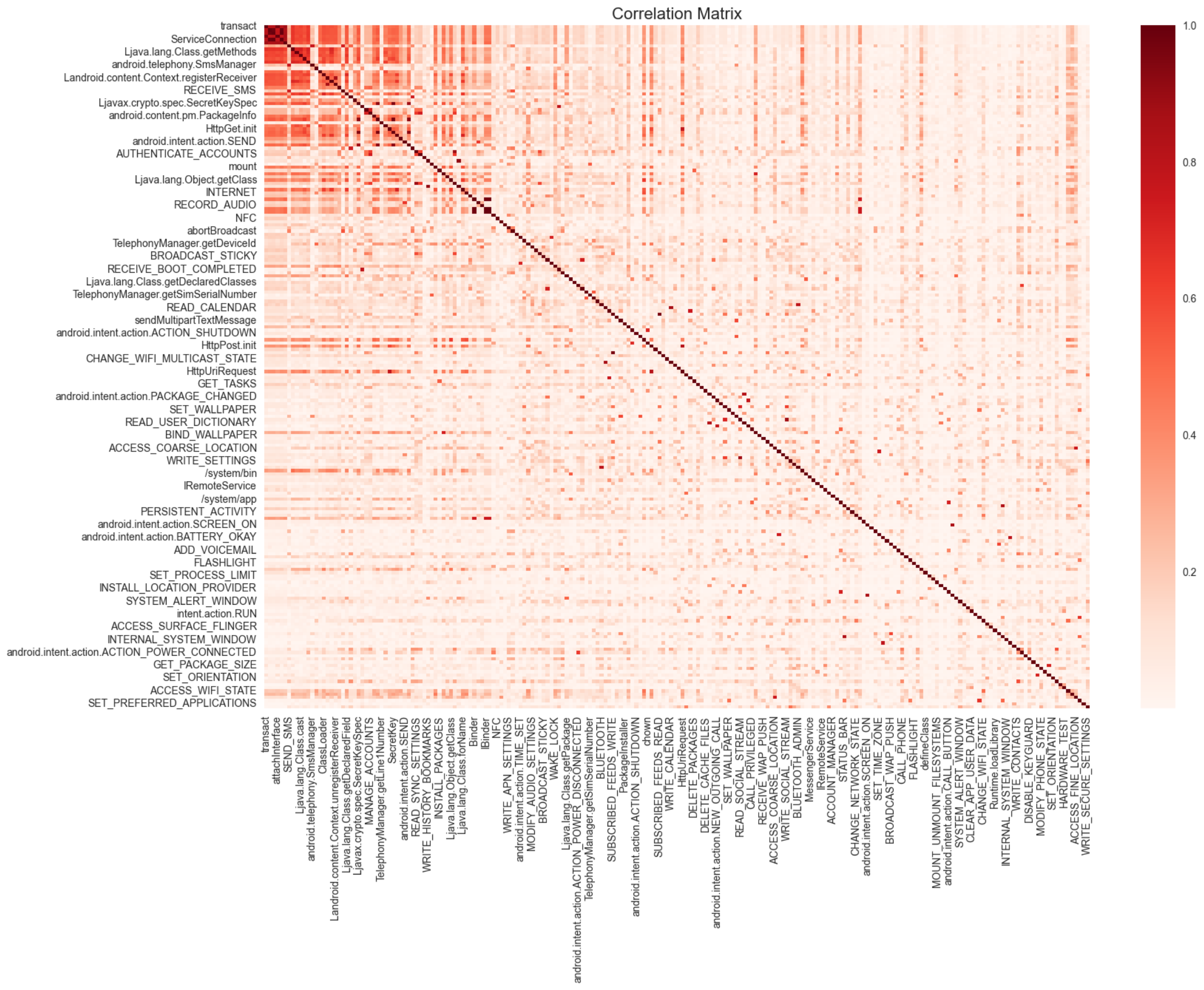


Figure 3: Correlation matrix

It turns out that the most correlated features are: 'bindService', 'attachInterface', 'ServiceConnection', 'SecretKey', 'IBinder', 'android.os.IBinder', 'SUBSCRIBED_FEEDS_READ'

Data Splitting

I decided to split the dataset in 3 parts:

- Training set: 60 %
- Validation set: 20 %
- Test set: 20 %

ML models

I have chosen 3 models:

- Neural Networks: usually works very well for almost every task

- Gaussian Naive Bayes: it is a simple model that can be used for binary classification
- Decision Tree: simple and easy-readable model

In order to find the best hyperparameters (tuning) I wrote a function called *GridSearchFun* which uses GridSearch for searching over specified parameter values for an estimator. In addition, this function returns the best estimator for each model. I draw below the NN loss function (both training and validation) (fig. 4).

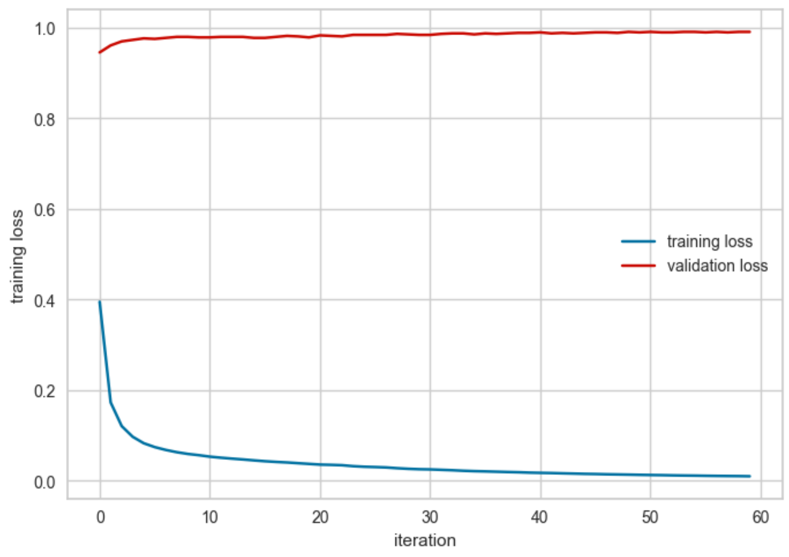


Figure 4: NN loss function

Since for Decision Tree and Gaussian NB there is not a training loss I used *learning_curve* that determines cross-validated training and test scores for different training set sizes. In this way it has been easier to plot and visualize how the models perform during training.
The blue line represents the mean of the computed balanced accuracy. Moreover, ShuffleSplit has been used for validation scores through the mentioned sklearn learning curve. Below the final results (fig. 5 and fig. 6).

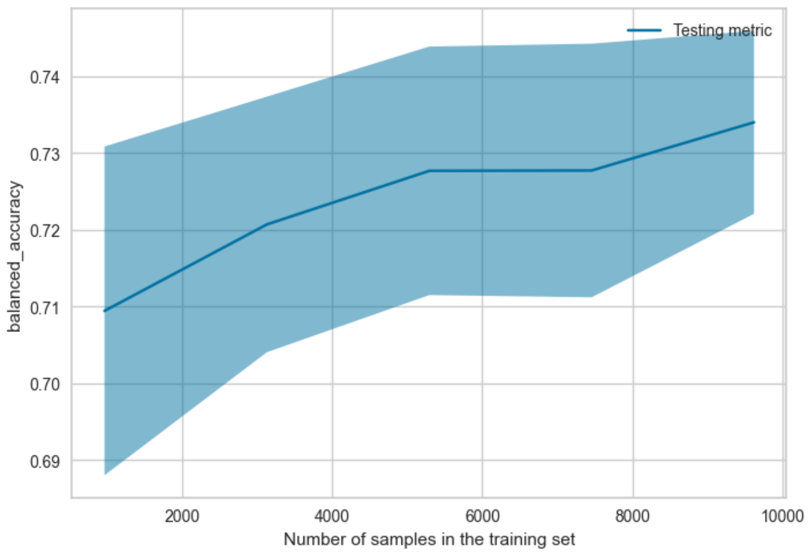
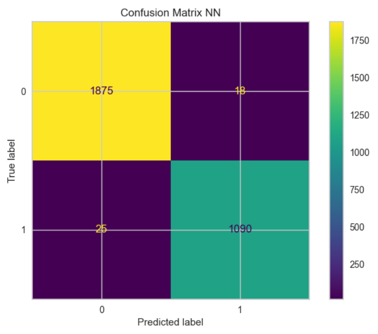
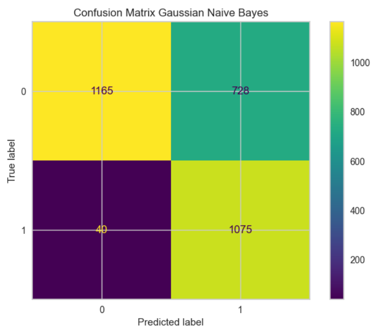


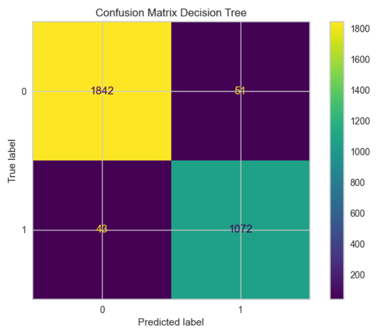
Figure 5: Learning Curve Gaussian NB



(a) Neural Networks



(b) Gaussian NB



(c) Decision Tree

Figure 7: Confusion Matrix

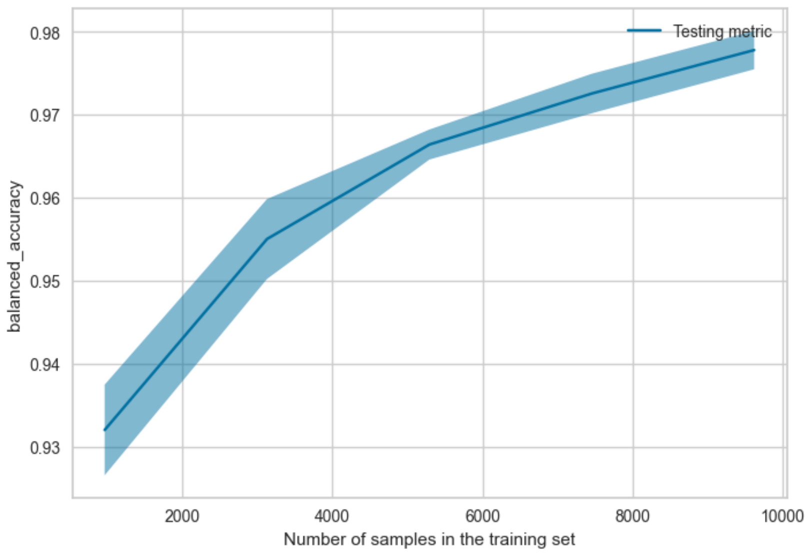


Figure 6: Learning Curve Decision Tree

As one can see, the best model is the Decision Tree with a balanced accuracy during training of almost 96%. However, the NN seems to be the best candidate during the testing. In fact, looking at the test phase the NN performs very well (tab. 1).

Model	**Balanced Accuracy**	**Accuracy**
NN	0.9851	0.9857
Decision Tree	0.9680	0.9697
Gaussian NB	0.7897	0.7446

Table 1: Test results

Furthermore I plotted the confusion matrix to confirm visually the results I obtained.

1.4. Quality of the Dataset

I believe that the dataset is valuable since both NN and Decision Tree can be considered good models for this task (False Negatives). Moreover with a more complex model the results could be much better, maybe reaching almost 100 percent of f1 score and balanced accuracy.

1.5. Bypassing your Model

I picked up the very first sample from the test set. For each model I change every time the probability to swap 0s to 1s each feature of the sample. It turns out that the best model that recognizes the sample as malicious is the decision tree. However, the probability to catch it is very low (about 40% Accuracy). NN performs worse (about 20 %).
I changed 0s to 1s because I wanted to keep the hostile features as they were without compromising the malicious signatures or permissions. In the end I successfully bypassed my models.

1.6. Improving your Model

2. Hardware Trojan Detection