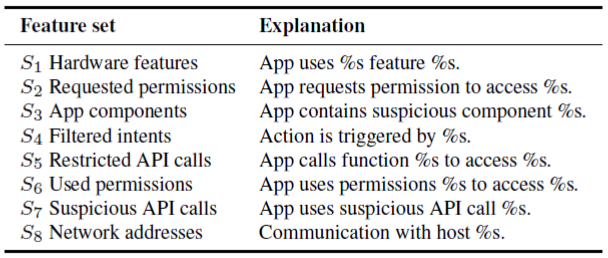
Malware detection

Homework 1 - Machine Learning

Mariani Matteo
Matricola ID: 1815188

Contents



Feature set	Explanation
S_1 Hardware features	App uses %s feature %s.
S_2 Requested permissions	App requests permission to access %s.
S_3 App components	App contains suspicious component %s.
S_4 Filtered intents	Action is triggered by %s.
S_5 Restricted API calls	App calls function %s to access %s.
S_6 Used permissions	App uses permissions %s to access %s.
S_7 Suspicious API calls	App uses suspicious API call %s.
S_8 Network addresses	Communication with host %s.

Figure 1: Feature set of an Android application

Introduction

Android is one of the most popular platforms for smartphones today. With several hundred thousands of applications in different markets, it provides a wealth of functionality to its users. Unfortunately, smartphones running Android are increasingly targeted by attackers and infected with malicious software. In contrast to other platforms, Android allows for installing applications from unverified sources, such as third-party markets, which makes bundling and distributing applications with malware easy for attackers. [1]
Machine Learning permits to build malware analysis techinques that (1) not need human support and (2) are resilient to obfuscation techniques[1]. In general machine learning permits to create malware analysis techniques based on the semantic of an application and not the code appeareance.

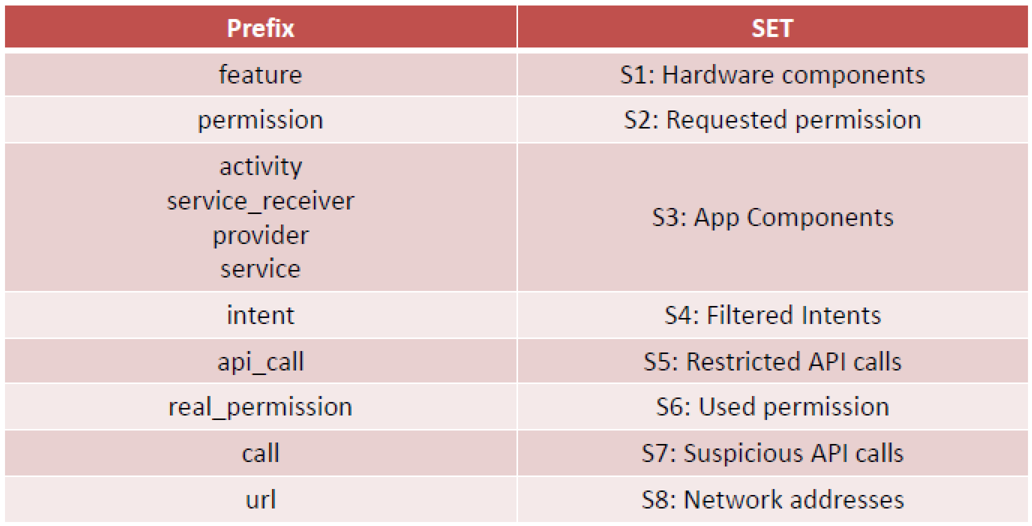
Malware Detection

The Android platform provides several informations about the applications' behaviour through the *feature sets*. Roughly speaking, they describe not only the permissions that a specific application has or the hardware component that it will use (feature sets extracted from the *AndroidManifest.xml*), but also all the calls to different APIs and the network addresses used from the application during its execution (feature sets extracted from disassembled code). There are several sets and a list with a brief explanation is given in the figure 1.
In this paper, the malware analysis is based on the Drebin dataset; a huge dataset that allow to focus entirely on the design choices of the machine learning method - instead of gathering the core informations from the applications, that Drebin already provides in a precise format.
Our *goal* is simple: given an apk, classify it as malware or not (malware detection).

Structure of Drebin dataset

Drebin dataset consists on 123.453 benign applications and 5.560 malware applications. All the applications are stored in text files whose name is the SHA1 hash of the apk. Each application contains the list of its features. Each feature is composed by a *prefix* and a *value*, the prefix represent the set the feature belong to (figure 2). The SHA1 hashes related to the malware apk are stored in a *.csv* file. [2]

[1]The obfuscation technique consists in hiding some application's sensitive informations that could be easily analyzed and reveal the malicious intent of the application.



Prefix	SET
feature	S1: Hardware components
permission	S2: Requested permission
activity service_receiver provider service	S3: App Components
intent	S4: Filtered Intents
api_call	S5: Restricted API calls
real_permission	S6: Used permission
call	S7: Suspicious API calls
url	S8: Network addresses

Figure 2: Prefixes of the feature set.

Naive Bayes for Malware Detection

There are several machine learning approaches that can be used for this task, each one differ not only on the type of preprocessing required on the dataset and on the way of representing each feature of the apk but also on the construction of the learning set and classification model.
In the following analysis, Bayesan approach has been used.
Naive Bayes classifiers are a family of probabilistic classifiers[2] based on applying Bayes' theorem with strong (naive) indipendence assumptions between the feature. Naive Bayes classifiers can be trained very efficiently in a *supervised learning* setting, like the Drebin dataset, in which we know which apk is labeled as malware or not.
In the Malware Detection problem, the input of the Naive Bayes is the set of applications (characterized with sequences of feature::info) and the learn function is:

$$f : apps \longrightarrow C, C = \{malware, not\ malware\}$$

So, the dataset D is characterized by a set of couple in which the first element app_i is the application - that is uniquely identified by its SHA1 hash - and the second element c_i is the label (class) that the application is certain to belong to.

$$D = \{< app_i, c_i >\}$$

It's assumed a *Multinomial Naive Bayes Distribution*, so - roughly speaking - the frequencies with which certain feature::info occurs in each classes are considered. In pseudocode, the algorithm is something similar to this:

[2]A probabilistic classifier is a classifier that is able to predict, given an observation of an input, a probability distribution over a set of classes, rather than only outputting the most likely class that the observation should belong to.

Algorithm 1: Naive Bayes Malware Detection with multinomial distribution (MU)

```
1  Learn Naive Bayes Malware MU ($D, C$);
   Input : Dataset $D$ and the set of all possible classes $C$
2  $F \leftarrow$ all distinct feature::info in $D$
3  for each target value $c_j \in C$ do
4      $apps_j \leftarrow$ subset of $D$ for which the target value is $c_j$
5      $t_j \leftarrow |apps_j|$ : total number of applications in $c_j$
6      $P(c_j) \leftarrow \frac{t_j}{|D|}$
7      $TF_j \leftarrow$ total number of feature::info in $apps_j$ (counting duplicates)
8      for each feature::info $f_i$ in $F$ do
9          $TF_i, j \leftarrow$ total number of times feature::info $f_i$ occurs in $apps_j$
10         $P(f_i|c_j) \leftarrow \frac{TF_i, j+1}{TF_j+|F|}$
11     end
12 end
```

Implementation

In this section will be explained step by step the implementation of the Naive Bayes analyzed before, according to the resolution of the Malware Detection problem. The programming language used is Python. The program is composed by two modules: *main.py* and *utils.py*. During the explanation, portions of the code will be analyzed to fully understand the role of each component.

main.py

```python
from src.utils import *

""" loading known malware csv file """
malware_csv_path = "/home/matteo/Scrivania/drebin/sha256_family.csv"
known_malware_dict = getKnownMalwareDict(malware_csv_path)
app_directory_path = "/home/matteo/Scrivania/drebin/feature_vectors_10000"

k_fold=5
(precision,recall,falsePosRate,accuracy,fMeasure) = applyNaiveBayes(app_directory_path,
                                                                    known_malware_dict,
                                                                    k_fold)

print "precision: " + str(precision*100) + " %"
print "recall: " + str(recall*100) + " %"
print "false positive rate: " + str(falsePosRate*100) +" %"
print "accuracy: " + str(accuracy*100) + " %"
print "fMeasure: " + str(fMeasure)
```

The *main.py* is pretty straight-forward:

- the *.csv* file is loaded and a dictionary with an integer key and values composed by the SHA1 hash name of the applications is created;
- the parameter `k_fold` is initialized for the *k-fold cross validation method*[3];

[3]In a prediction problem, a model is usually given a dataset of known data on which training is run (training dataset), and a dataset of unknown data (or first seen data) against which the model is tested (testing dataset). The goal of cross validation is to define a dataset to "test" the model in the training phase (i.e., the validation dataset), in order to limit problems like overfitting, give an insight on how the model will generalize to an independent dataset.

- the Naive Bayes is applied to the dataset, obtaining some useful evaluation metrics.

It's easy to understand that the core method is exactly `applyNaiveBayes(path,known_malware,k_fold)` that take as input the path of the dataset, the set of applications that are known to be malware in the same dataset and the parameter for the k-fold cross valitadion.

applyNaiveBayes method in utils.py

```
def applyNaiveBayes(app_directory_path,
                    known_malware_dict,
                    k_fold):
    """ Among all the features available (10),
     only a subset is taken to avoid huge computational time """
    searched_features = getTargetFeatures()
    print "considered features: " + str(searched_features)

    """ get a shuffled list of all the sha_256 app names s.t.
     in every instance of NaiveBayes, the pool is randomized """
    shuffled_app_names = getShuffledListApps(app_directory_path)
    tot_apps = len(shuffled_app_names)
    print "number of applications considered: " + str(tot_apps)

    print "number of k-fold: " + str(k_fold)
    step = tot_apps//k_fold

    """ list containing tuples of 4 elements corresponding to
     TP, TN, FP and FN for each step k"""
    evaluation_metrics_data = []

    """ k—cross validation """
    iter=1
    for i in range(0,step*k_fold,step):
        truePos = trueNeg = falsePos = falseNeg = 0
        """ evaluation set for the step k containing the sha_256 digest of the apps """
        validSet_app_names = shuffled_app_names[i:i+step]
        """ training set for the step k containing the sha_256 digest of the apps """
        trainSet_app_names = shuffled_app_names[0:i] + shuffled_app_names[i+step:tot_apps]
        """ start training on the trainingSet """
        (malware_dict,occMalwFeatures) = doTraining(app_directory_path,
                                                    trainSet_app_names,
                                                    known_malware_dict,
                                                    searched_features)
        """ start classification on the validSet"""
        for app_sha256_name in validSet_app_names:
            isMalware = 0
            if app_sha256_name in known_malware_dict.viewvalues():
                isMalware = 1
            path_app_sha256_name = app_directory_path + "/" + app_sha256_name
            valid_current_app = open(path_app_sha256_name,'r')
            valid_current_app = valid_current_app.readlines()
            """ isMalware is the correct class for the valid current_app.
               If the classification missclassify the app, a FP or FN occurs """
            classification = classify(valid_current_app,
                                      searched_features,
                                      malware_dict,
                                      occMalwFeatures)
               if  classification!= isMalware:
                if isMalware:
                    falseNeg+=1
                else:
                    falsePos+=1
             else:
                if isMalware:
                    truePos+=1
                else:
                    trueNeg+=1

        """ some useful infos for the report """
        print "iteration " + str(iter)
        print "--> # malware in the training set: " + str(occMalwFeatures[0]) +
                "/" + str(len(trainSet_app_names))
        print "--> # not malware in the training set: " + str(occMalwFeatures[1]) +
                "/" + str(len(trainSet_app_names))
        print "--> # True positive: " + str(truePos)
        print "--> # True negative: " + str(trueNeg)
        print "--> # False positive: " + str(falsePos)
        print "--> # False negative: " + str(falseNeg)
        iter+=1

        evaluation_metrics_data.append((truePos,trueNeg,falsePos,falseNeg))

    return getEvalutationMetrics(evaluation_metrics_data, k_fold)
```

First of all, to avoid some extra computational time required from the classification, only a subset of all the available features is considered. The choice is basically deterministic but in the *Results* section is explained how some features can be more useful than others during the learning phase.
An other trick is used to have a good insight on how the model is good to resolve the malware detection problem: each time the program runs, the entire dataset - or, more precisely, the list containing all the SHA1 hashes of the dataset - is randomly shuffled. In this way, you have the guarantee that the results of the algorithm is not bonded on some same properties considered in each run.
After all these preliminary steps, the creation of the training set and the testing set starts. The training set is used to train the model on understanding which features/properties could affect the probability for which a specific apk could be a malware or not. The testing set - or valid set - is filled with applications that need to be evaluate after the training phase, using the model. Each set is created from the entire dataset: one part will be considered as testing set and the remaining part as training set. Generally, the training set is larger than the testing one because the model needs a lot of data to be trained well.
Using a k-fold cross validation, $\frac{|D|}{k\_fold}$ is considered as testing set - where D is the entire dataset - and $|D| - \frac{|D|}{k\_fold}$ is considered as training set.

In each `step` :

- the training and testing set changes in a way that after all the $step_i$ for $i = 1...k\_fold$ each application was part of the testing test at most one time;
- the number of True Positive (TP), True Negative (TN), False Positive (FP) and False Negative (FN) are collected in a list (`evaluation_metrics_data`);
- the training phase starts on the training set, collecting the occurrences of all feature::info for each application and distinguish them from malware or not malware occurrences. Those data are stored in `malware_dict` dictionary;

- the classification phase starts on the testing set, comparing for each application its true class and the class that the model thinks that that particular apk belongs to. In base of the results, the values TP, TN, FP and FN are updated.

Let's consider a deeper analysis of the training phase and the classification phase.
The **learning phase**, is accomplished via the `doTraining(path,training_set,known_malware,searched_features)` method that take as input the path of the dataset, the training set containing the SHA1 digests of the applications, the set of applications that are known to be malware and the list of features to consider.

doTraining method in utils.py

```python
def doTraining(app_directory_path,shuffled_app_names,known_malware_dict,searched_features):
    malware_dict = {}
    numFeatureInfoMalw = 0
    numFeatureInfoNotMalw = 0
    numMalwareApps = 0
    numNotMalwareApps = 0
    numUniqueFeatureInfo = 0

    """ for each app manifest in shuffled list of app names """
    for app_sha256_name in shuffled_app_names:
        isMalware = 0
        path_app_sha256_name = app_directory_path + "/" + app_sha256_name
        current_app = open(path_app_sha256_name,'r')
        current_app = current_app.readlines()
        """ if the current app is a known malware, then the flag isMalware is set to true"""
        if app_sha256_name in known_malware_dict.viewvalues():
            isMalware=1
            numMalwareApps+=1
        else:
            numNotMalwareApps+=1
        """for each line in the current app """
        for feature_line in current_app:
            feature_line = feature_line.replace("::", " ", 1).split()
            '''condition used to skip an iteration of the loop if 1 of 2 conditions are met:
                - some \n in the app manifest appears (empty lines)
                - some features not requested appear during the scanning
            '''
            if len(feature_line)==0 or not(feature_line[0] in searched_features):
                continue
            """ Case 1: the feature encountered has already been stored as
               a key in the dictionary"""
            if feature_line[0] in malware_dict:
                feature_info_malware_dict = malware_dict[feature_line[0]]
                """ Case 1.1: the current feature::info is not already been stored
                    in the nested dictionary """
                if not feature_line[1] in feature_info_malware_dict:
                    numUniqueFeatureInfo+=1
                    if isMalware:
                        numFeatureInfoMalw+=1
                        feature_info_malware_dict[feature_line[1]] = (1,0)
                    else:
                        numFeatureInfoNotMalw+=1
                        feature_info_malware_dict[feature_line[1]] = (0,1)
                else:
                    """ Case 1.2: the current feature::info is already present in
                         the dictionary, only update required"""
                    """ if the current app is considered as malware from the csv,
                         then the corrispondent occurrence is incremented """
                    if isMalware:
                        numFeatureInfoMalw+=1
                        (occMalw,occNotMalw) = feature_info_malware_dict[feature_line[1]]
                        feature_info_malware_dict[feature_line[1]] = (occMalw+1,occNotMalw)
                    else:
                        numFeatureInfoNotMalw+=1
                        (occMalw,occNotMalw) = feature_info_malware_dict[feature_line[1]]
                        feature_info_malware_dict[feature_line[1]] = (occMalw,occNotMalw+1)

            else:
                """ Case 2: the feature encountered is not in the dictionary, it's new """
                new_feature_info_malware_dict = {}
                numUniqueFeatureInfo+=1
                if isMalware:
                    numFeatureInfoMalw+=1
                    new_feature_info_malware_dict[feature_line[1]] = (1,0)
                else:
                    numFeatureInfoNotMalw+=1
                    new_feature_info_malware_dict[feature_line[1]] = (0,1)
                malware_dict[feature_line[0]] =  new_feature_info_malware_dict

    occMalwFeature = [numMalwareApps,
                      numNotMalwareApps,
                      numFeatureInfoMalw,
                      numFeatureInfoNotMalw,
                      numUniqueFeatureInfo]
    return (malware_dict,occMalwFeature)
```

The main output is a `malware_dict`, a dictionary that has as key the *feature_type* (i.e. permission, api_call, url...) and as value a second dictionary `feature_info_malware_dict`. The nested dictionary has as key the *info* associated to the feature_type and as value a couple of integer `(occMalw,occNotMalw)` representing the occurences of the corrispondent feature_type::info in malware ad not malware cases.
A second part of the output is represented by the list `occMalwFeature` that incapsulates some useful data for the calculation of the conditional probabilities associated to each feature::info - per class - like (1) the number of malware and not malware applications found in the current training set, (2) the number of feature::info associated to malware or not malware applications of the same set (counting duplicates) and (3) the number of unique feature::info found in the entire current training set, without distinction between malware or not malware membership.

After collecting these data, the **classification phase** can finally begin.
The method `classify(current_app,searched_features,malware_dict,occMalwFeatures)` take as input one of the applications of the testing set that need to be classified according to the trained model, the list of features considered and the two training phase's outputs (`malware_dict` and `occMalwFeature`). It returns 1 if the current application is considered a malware (`probMalw > probNotMalw`), 0 otherwise. The code below explains in detail how the `probMalw` and `probNotMalw` are obtained.

classify method in utils.py

```python
def classify(current_app,
             searched_features,
             malware_dict,
             occMalwFeatures):
    isMalware = 0
    """ following parameters are explicity renamed to semplify
     their recognition in the code """
    priorProbMalw = occMalwFeatures[0]/float(occMalwFeatures[0]+occMalwFeatures[1])
    priorProbNotMalw = occMalwFeatures[1]/float(occMalwFeatures[0]+occMalwFeatures[1])
    totUniqueInfoFeatures = occMalwFeatures[4]
    numFeatureInfoMalware = occMalwFeatures[2]
    numFeatureInfoNotMalware = occMalwFeatures[3]


    """ i=1,2 because there are only two classes available """
    for i in range(2):
        if i==0:
            probMalw = priorProbMalw
        else:
            probNotMalw = priorProbNotMalw
        for feature_line in current_app:
            feature_line = feature_line.replace("::", " ", 1).split()
            '''condition used to skip an iteration of the loop if 1 of 2 conditions are met:
                - some \n in the app manifest appears (empty lines)
                - some features not requested appeared during the scanning
            '''
            if len(feature_line)==0 or not(feature_line[0] in searched_features):
                continue
            """ if exsist the couple feature::info in the dictionary """
            if feature_line[1] in malware_dict[feature_line[0]]:
                feature_dict = malware_dict[feature_line[0]]
                (occMalw,occNotMalw) = feature_dict[feature_line[1]]
                if i==0:
                    tot=numFeatureInfoMalware+totUniqueInfoFeatures
                    prob_featureXclass = ((occMalw+1)/float(tot))
                    probMalw*= prob_featureXclass
                else:
                    tot=numFeatureInfoNotMalware+totUniqueInfoFeatures
                    prob_featureXclass = ((occNotMalw+1)/float(tot))
                    probNotMalw*= prob_featureXclass
    if probMalw > probNotMalw:
        isMalware=1
    return isMalware
```

The result is then compared to the true class of the current apk - in the `applyNaiveBayes` method:

- if the classification of some apk is equal to its true class (`classification == isMalware`, where `isMalware` is the true class of the apk) then the application is correctly classified by the model. TP or TN are increased by 1 - depending if the apk is a malware or not;
- otherwise, if the classification of some apk is not equal to its true class (`classification != isMalware`) then the application is wrongly classified by the model. FN or FP are increased by 1 - depending on the kind of missclassification.

particular part of the applyNaiveBayes method in utils.py

```
            classification = classify(valid_current_app,
                                      searched_features,
                                      malware_dict,
                                      occMalwFeatures)

               if  classification!= isMalware:
                if isMalware:
                    falseNeg+=1
                else:
                    falsePos+=1
             else:
                if isMalware:
                    truePos+=1
                else:
                    trueNeg+=1

        """ some useful infos for the report """
        print "iteration " + str(iter)
        print "--> # malware in the training set: " + str(occMalwFeatures[0]) +
                 "/" + str(len(trainSet_app_names))
        print "--> # not malware in the training set: " + str(occMalwFeatures[1]) +
                 "/" + str(len(trainSet_app_names))
        print "--> # True positive: " + str(truePos)
        print "--> # True negative: " + str(trueNeg)
        print "--> # False positive: " + str(falsePos)
        print "--> # False negative: " + str(falseNeg)
        iter+=1

        evaluation_metrics_data.append((truePos,trueNeg,falsePos,falseNeg))

    return getEvalutationMetrics(evaluation_metrics_data, k_fold)
```

Remember that in each $step_i$ the number of TP, TN, FP and FN are collected in the list `evaluation_metrics_data` as tuples and after the last step ($step_{\text{k\_fold}}$) all the data required to evaluate the classification are finally collected and normalized w.r.t. the value `k_fold` of the k-fold cross validation. This is accomplished in the method `getEvaluationMetrics(evaluation_metrics_data,k_fold)` that produces as output a tuple of evaluation metrics that describe how good the model was in the classification.

getEvaluationMetrics method in utils.py

```
def getEvalutationMetrics(evaluation_metrics_data,k_fold):
    normTP = normTN = normFP = normFN = 0
    for (tp,tn,fp,fn) in evaluation_metrics_data:
        normTP+=tp
        normTN+=tn
        normFP+=fp
        normFN+=fn
    normTP = normTP/float(k_fold)
    normTN = normTN/float(k_fold)
    normFP = normFP/float(k_fold)
    normFN = normFN/float(k_fold)
    precision = getPrecision(normTP,normFP)
    recall = getRecall(normTP, normFN)
    falsePosRate = getFalsePosRate(normTN, normFP)
    accuracy = getAccuracy(normTP,normTN,normFP,normFN)
    fMeasure = getFMeasure(precision, recall)
    return (precision, recall, falsePosRate, accuracy, fMeasure)
```

The metrics are the following:

- *precision*, that represents the fraction of relevant instances among the retrieved instances (how many files are real malware among those I considered as malware?). The formula is $\frac{TP}{(TP+FP)}$;
- *recall*, that represents the fraction of relevant instances that have been retrieved over the total amount of relevant instances (how many malware did I spot among those in the test set?). The formula is $\frac{TP}{(TP+FN)}$;
- *false positive rate*, that represents the percentage of incorrect results that are, in fact, negative (how many files did I wrongly consider as malware among all the benign files?). The formula is $\frac{FP}{(FP+TN)}$;
- *accuracy*, that represents the percentage of files that have been classified correctly by the model. The formula is $\frac{TP+TN}{(TP+FP+FN+TN)}$;
- *f-measure*, that can be interpreted as a weighted average of precision and recall, for which the best value is 1 and the worst is 0. The formula is $\frac{2\cdot(precision\cdot recall)}{(precision+recall)}$.

In the following section, the Naive Bayes will be applied several times with different inputs and tweaking some parameters in order to explain how the classifier responds in different instances of the same problem.

Results

An example of a specific report produced by a particular combination of the parameters

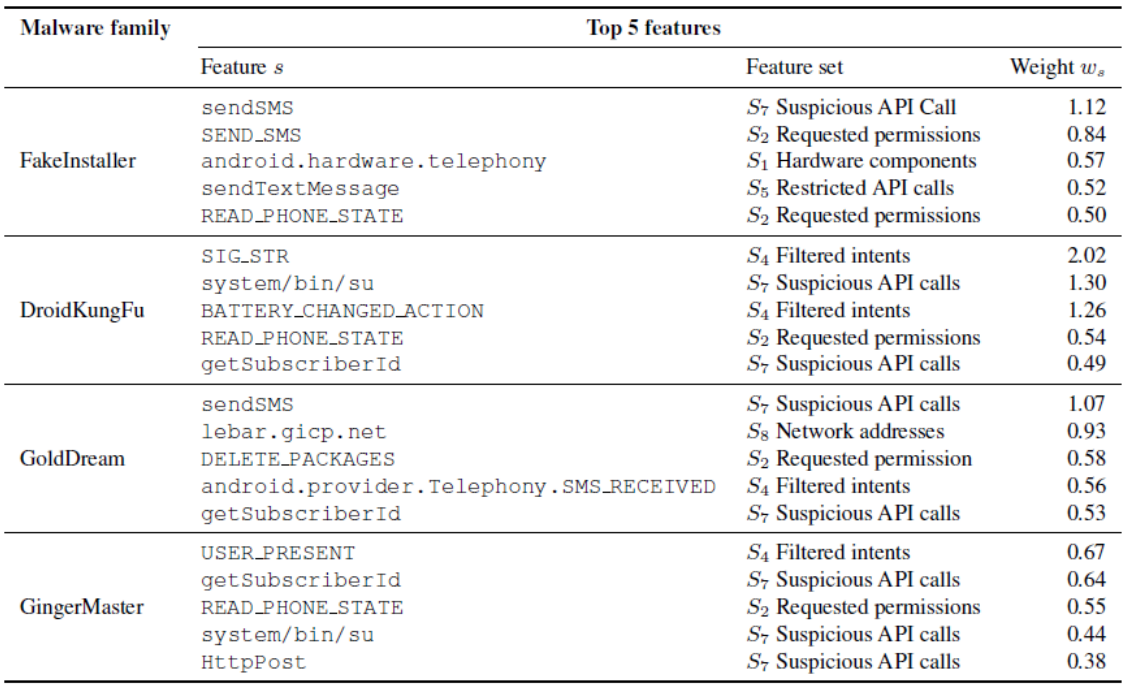
```
start! [2017_11_15_15_03_02]
considered features: ['url', 'call', 'intent', 'permission', 'feature']
number of applications considered: 1000
number of k-fold: 2
iteration 1
--> # malware in the training set: 26/500
--> # not malware in the training set: 474/500
--> # True positive: 7
--> # True negative: 473
--> # False positive: 3
--> # False negative: 17
iteration 2
--> # malware in the training set: 24/500
--> # not malware in the training set: 476/500
--> # True positive: 3
--> # True negative: 474
--> # False positive: 0
--> # False negative: 23
classification done! [2017_11_15_15_03_03]

precision: 76.9230769231 %
recall: 20.0 %
false positive rate: 0.315789473684 %
accuracy: 95.7 %
fMeasure: 0.31746031746
```

Assuming the Drebin dataset, the results are provided modifing three parameters:

- k_fold value of the k-fold cross validation;

- the size of the dataset in input, taking a random subset of the entire Drebin dataset D;
- the features considered, in different numbers and categories.

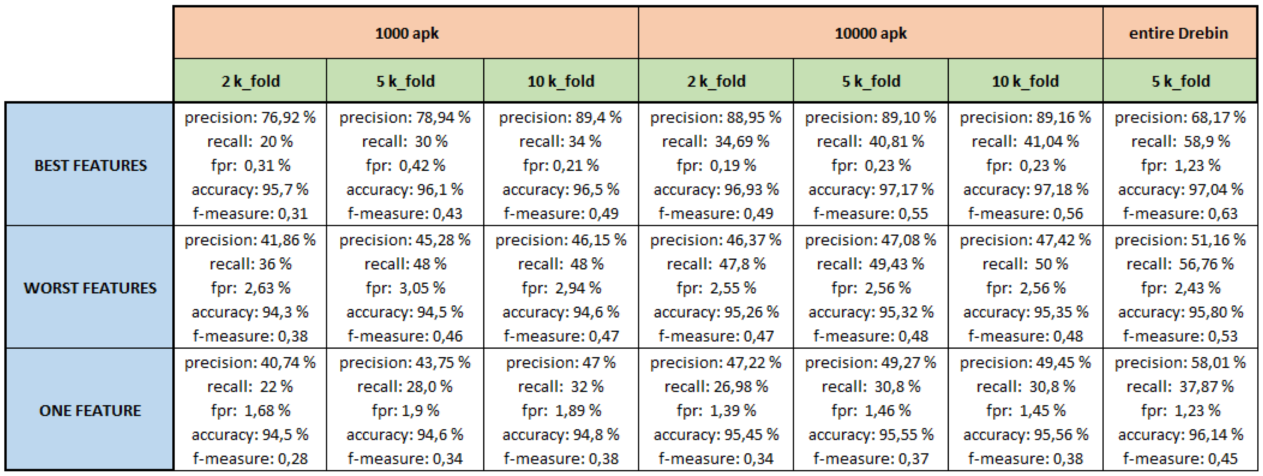
The output of each combination of the parameters is stored in .txt file that records all the useful informations for the analysis - as can be seen in the example. Here, it's considered a subset of D composed by 1000 applications; taking care of only five features and using two iterations.
How many features need to be considered? This is not a simple question to answer. As we'll see in the following results, considering too many features *may* lead the model to predict wrong on more data. This happens because some features could be misleading for the model itself, granting a "foggy" and imprecise vision of the true nature of the dataset. For this reason, sometimes we'll achieve a better performance considering only a subset of the features.
Which features have to be considered? Even in this case, the answer is not straight-forward, but the [DEEDAM][1] paper can give a good insight of what we should choose to obtain a better model. The figure below represents which feature set have the higher weight in those malware belonging in the top four Drebin's malware families. Using this information, we can determine a specific subset of features that can help the model to train better on the dataset.



Malware family	**Top 5 features**		
	Feature s	Feature set	Weight w_s
FakeInstaller	`sendSMS`	S_7 Suspicious API Call	1.12
	`SEND_SMS`	S_2 Requested permissions	0.84
	`android.hardware.telephony`	S_1 Hardware components	0.57
	`sendTextMessage`	S_5 Restricted API calls	0.52
	`READ_PHONE_STATE`	S_2 Requested permissions	0.50
DroidKungFu	`SIG_STR`	S_4 Filtered intents	2.02
	`system/bin/su`	S_7 Suspicious API calls	1.30
	`BATTERY_CHANGED_ACTION`	S_4 Filtered intents	1.26
	`READ_PHONE_STATE`	S_2 Requested permissions	0.54
	`getSubscriberId`	S_7 Suspicious API calls	0.49
GoldDream	`sendSMS`	S_7 Suspicious API calls	1.07
	`lebar.gicp.net`	S_8 Network addresses	0.93
	`DELETE_PACKAGES`	S_2 Requested permission	0.58
	`android.provider.Telephony.SMS_RECEIVED`	S_4 Filtered intents	0.56
	`getSubscriberId`	S_7 Suspicious API calls	0.53
GingerMaster	`USER_PRESENT`	S_4 Filtered intents	0.67
	`getSubscriberId`	S_7 Suspicious API calls	0.64
	`READ_PHONE_STATE`	S_2 Requested permissions	0.55
	`system/bin/su`	S_7 Suspicious API calls	0.44
	`HttpPost`	S_7 Suspicious API calls	0.38

Considering the entire Drebin dataset each time a test is required, can be too demanding because of its size. For this reason, a subset - of a proper size - can be chosen with ease, but it should mantain the same distribution of the Drebin dataset - picking it at random is a solution.
Explained these assumptions, the following table of results can be easily understood (figure 3). The Naive Bayes for Malware Detection was tested on two subsets of the Drebin - constituted by 1.000 and 10.000 apk, respectively - and the entire Drebin. The parameter of the k_fold changed from 2 up to 5 and 10. Three subsets of the features were considered:

- *"Best Features"*, represents the subset of features composed by: *url*, *call*, *intent*, *permission* and *feature*. These features are the ones with the higher weight, according to the previous table of the [DEEDAM][1];



	1000 apk			10000 apk			entire Drebin
	2 k_fold	5 k_fold	10 k_fold	2 k_fold	5 k_fold	10 k_fold	5 k_fold
BEST FEATURES	precision: 76,92 % recall: 20 % fpr: 0,31 % accuracy: 95,7 % f-measure: 0,31	precision: 78,94 % recall: 30 % fpr: 0,42 % accuracy: 96,1 % f-measure: 0,43	precision: 89,4 % recall: 34 % fpr: 0,21 % accuracy: 96,5 % f-measure: 0,49	precision: 88,95 % recall: 34,69 % fpr: 0,19 % accuracy: 96,93 % f-measure: 0,49	precision: 89,10 % recall: 40,81 % fpr: 0,23 % accuracy: 97,17 % f-measure: 0,55	precision: 89,16 % recall: 41,04 % fpr: 0,23 % accuracy: 97,18 % f-measure: 0,56	precision: 68,17 % recall: 58,9 % fpr: 1,23 % accuracy: 97,04 % f-measure: 0,63
WORST FEATURES	precision: 41,86 % recall: 36 % fpr: 2,63 % accuracy: 94,3 % f-measure: 0,38	precision: 45,28 % recall: 48 % fpr: 3,05 % accuracy: 94,5 % f-measure: 0,46	precision: 46,15 % recall: 48 % fpr: 2,94 % accuracy: 94,6 % f-measure: 0,47	precision: 46,37 % recall: 47,8 % fpr: 2,55 % accuracy: 95,26 % f-measure: 0,47	precision: 47,08 % recall: 49,43 % fpr: 2,56 % accuracy: 95,32 % f-measure: 0,48	precision: 47,42 % recall: 50 % fpr: 2,56 % accuracy: 95,35 % f-measure: 0,48	precision: 51,16 % recall: 56,76 % fpr: 2,43 % accuracy: 95,80 % f-measure: 0,53
ONE FEATURE	precision: 40,74 % recall: 22 % fpr: 1,68 % accuracy: 94,5 % f-measure: 0,28	precision: 43,75 % recall: 28,0 % fpr: 1,9 % accuracy: 94,6 % f-measure: 0,34	precision: 47 % recall: 32 % fpr: 1,89 % accuracy: 94,8 % f-measure: 0,38	precision: 47,22 % recall: 26,98 % fpr: 1,39 % accuracy: 95,45 % f-measure: 0,34	precision: 49,27 % recall: 30,8 % fpr: 1,46 % accuracy: 95,55 % f-measure: 0,37	precision: 49,45 % recall: 30,8 % fpr: 1,45 % accuracy: 95,56 % f-measure: 0,38	precision: 58,01 % recall: 37,87 % fpr: 1,23 % accuracy: 96,14 % f-measure: 0,45

Figure 3: Results of the Naive Bayes classification with particular parameters on different Drebin's subset

- *"Worst Features"*, on the contrary, represents the subset of features that have the lowest weight and are: *api_call* and *real_permission*;
- *"One Feature"*, represents only one feature - *api_call*, for these evalution tests. This case is considered to highlight the scenario in which - sometimes - more features could be misleading for the model.

Analyzing the table of the result, there are some conclusion that can be made:

- higher k_fold parameter allow to make more precise estimation of the classifier's performance;
- let's consider the recall and the precision. Even if both high recall and high precision is something desirable, in the real world this is difficult - impossible - to achieve. Considering the task of malware detection, if we should pick only one of the two to maximize, the candidate should be the recall. The choice is pretty obvious: we can make some errors saying that some "safe" apk is a malware (low precision) but it's a *real* problem if some apk could be considered safe even if it's a real malware, in reality (low recall). But it's really so simple? If the Naive Bayes classifies *all* the apk as malware, then the recall would be 100%. This would not be a good classificator: balance is important. For this reason, the F-measure can give a good insight on how the classifier is *really* good. The F-measure has a higher value when (1) the "best features" are considered and (2) the dataset in input is large enough;
- the accuracy is not a good measure to evaluate the true performance of this classification because the Drebin dataset is terrible unbalanced (123.453 benign and 5.560 malware applications). However, we can still get some useful information: the accuracy is lower in the cases in which the "worst features" are considered but it's slightly higher in the "one feature" cases, as expected (beacuse of the misleading features considered).

References

[1] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck *DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket.*

[2] *The Drebin Dataset.* 2012. [https://www.sec.cs.tu-bs.de/ danarp/drebin/]