**UNIVERSITY OF PISA**

Data Mining and Machine Learning

*Year 2020/2021*

# MONEY GUARD

*An application to recognize credit card frauds*

*AHMED SALAH TAWFIK IBRAHIM*

*FRANCESCO DEL TURCO*

## INDEX

# 1. Introduction

Money guard is an application that allows the user to check whether a transaction made with his/her card is fraudulent or not. Inserting the information about the transaction, the user will be able to see if it has been a normal transaction or if something unusual happened.

Considering the nature of our project, our aim is not only to find the model with highest accuracy: in such a context, it is important to use a model that has a low number of False Positives, that in our case are identified as fraudulent transactions that are erroneously classified as non-fraudulent ones. This situation is the most dangerous situation and must be avoided.

At the same time, it is also important to have a low number of False Negatives: in our application, False Negatives are non-fraudulent transactions that are erroneously classified as fraudulent ones. In real-world applications for credit card fraud detection, each transaction (or a statistical group of transactions) classified as fraudulent is manually inspected by humans: if we generate a high number of False Negatives, we would not create a problem to the user directly, but the controller may miss real frauds.

The purpose of the project is to implement a classification model able to recognize fraudulent transactions in order for the user to have a tool to control the validity of the transaction done with his/her credit card. At the same time, a simple data stream model is implemented to simulate how a stream of transactions would be handled.

The implementation of the application can be found in the following git repository:

https://github.com/ahmed531998/MoneyGuard

# 2. Dataset

The dataset used to train the model is a simulated credit card transaction dataset that contains information about several credit card transactions that happened between 1st January 2019 and 31st December 2020.
The original dataset had as attributes for each instance:

- id of the transaction;
- transaction date and time;
- credit card number;
- merchant's name, category, longitude and latitude;
- transaction amount;
- credit card holder's first and last name, gender, street address, city, city's population, state, job, date of birth, latitude and longitude;
- UNIX timestamp corresponding to the transaction time;
- is_fraud, corresponding to the class attribute.

After an initial study, we noticed how some attributes of the dataset were redundant or not useful for our study: we therefore decided to remove them using a script in order to have a dataset with lower dimensionality and without redundancy The "id" attribute has been removed since it was not interesting for our study, representing a fictitious id given to the transaction in the original dataset. Moreover, some attributes have been modified in order for them to be processed in an easier way: in particular, we converted the "date of birth" attribute to an "age" attribute which represents the age of the user at the time of the transaction. In both cases, these changes have been done to make the classification process easier.
In this dataset, the class attribute is called "is_fraud" and tells if a transaction is fraudulent or not. We therefore have just two values for the class attribute, 0 for non-fraudulent (positive class) transactions and 1 for fraudulent ones (negative class).
The dataset does not present any missing or null value, but it is strongly unbalanced, since there were many more non-fraudulent transactions than fraudulent ones. It was therefore necessary to perform a resampling of the dataset during the pre-processing phase, in order to have a balanced dataset to be processed by the classification algorithms.

# 3. Classification

To carry on the KDD process, we have used the Python library "python-weka-wrapper3", corresponding to the Java Weka API and running on a Java Virtual Machine. To load the dataset and work with it, we have used the Weka ArffLoader converter.

## 3.1 Pre-processing

After an initial manual attribute selection where we removed the "id" column and added an "age" column, we performed a pre-processing phase in order to obtain a dataset where we can use a classifier on. The pre-processing phase was divided into four parts: attribute discretization, attribute removal, conversion of string attributes to nominal of  and resampling.
During the discretization phase, we studied the distribution of the continuous attributes with respect to each class using a kernel density estimation. Out of all the continuous attributes, only the amount showed a different distribution of values for each class which encouraged us to perform a supervised discretization on it. Therefore, we used the supervised Weka discretization filter which is based on the Fayyad and Irani.
During the second phase, we used the Remove unsupervised Weka filter to remove the redundant and useless attributes aforementioned, that in this case were "transaction date and time" (redundant, same as UNIX time), "date of birth" (redundant, same as "age") and  "transaction number" (irrelevant).
During the third phase, we converted the string attributes into nominal attributes for them to be correctly handled by the classifiers. In particular, using the StringToNominal unsupervised Weka filter, we converted the following attributes: credit card number,  merchant's name, credit card holder's first and last name, street, city, state and zip code of the credit card holder.
During the resampling, we had to balance the dataset since it was strongly imbalanced in favour of the non-fraudulent transactions. In particular, before pre-processing, the dataset had a total of 1,604,202 instances, divided into 1,596,051 non-fraudulent transactions and 8,151 fraudulent transactions. For rebalancing, we decided to apply both undersampling and oversampling (for this last one, we used

SMOTE). We decided to use both rebalancing techniques because simple undersampling would have caused an important loss of information from the majority class, since we would have to move from more than 1.5M instances to 8,151; at the same time, we decided to use SMOTE as oversampling algorithm, instead of simple oversampling, because the oversampling required was considerable, so just duplicating the same 8,151 instances of the minority class would not have added any value (the percentage of SMOTE instances to create has been set to 14,622%, while the percentage for undersampling was set to 85.8%). For undersampling, we applied the Resample Weka filter.

At the end of the pre-processing phase, we obtained a fully balanced dataset, with a total of 2,400,000 instances divided in two classes of 1,200,000 instances each, with each instance having 18 attributes, both nominal and numeric.

## 3.3 Dataset splitting and classification

We decided to test different classification algorithms against our dataset, in order to see which one was the best in terms of computational time and final results. Since we had only a single dataset, we had to split it into a training and a test set: our approach was to test each classifier both splitting the dataset using a 10-folds cross validation and a percentage split (70% of the initial dataset has been used to generate the training set, while the other 30% has been used as test set).

For our application, we decided to use different algorithms and see which one had the best results. In particular, we used J48 (with and without pruning), Naive Bayes, Bayesian Belief Networks, Random Forest and KNN. We also decided to test it using the Hoeffding Tree, because it could be interesting to understand the accuracy of this model in many situations. In fact, a bank may be interested in processing a lot of transactions per hour happening with their cards, so an algorithm like the Hoeffding Tree may be of higher interest for such an application.

Since we had too many attributes to consider, it was of great essence to perform attribute selection on our training set. We therefore decided to use Weka's Attribute selected classifier with all the aforementioned classification models, except the Hoeffding Tree model, since in the case of Hoeffding Tree we are dealing with streams of data, which means we cannot just fix a subset of attributes to be used in our model because the importance of each attribute changes in the case of streaming. Therefore, we used attribute selection only with the rest of the models. In fact, we tried two different attribute selection methods: CFS Subset evaluation with Best First search and Information Gain with the ranker, where the latter was configured to select the best 4 attributes. The CFS Subset attribute selection selected the attributes “age” and “amount” whereas the InfoGain selected the attributes amount, merchant, UNIX time and latitude with InfoGain values (0.732, 0.51, 0.456, 0.443) respectively. In any case, the CFS Subset selection gave generally good results; however, the InfoGain outperformed it and that is why we decided to use it and show its results in this documentation. We refer the curious reader to check the results of CFS subset selection in the github repository.

We will now present the results of each model we have studied, showing for each model what we have obtained with the 10-folds cross-validation; after the comparison of each model, we will also show the results of the percentage split method on the best performing classification model in terms of results and computational time. We refer the curious reader to check the detailed results (cross validation and train-test split) of each model on our github repository.

## J48 (with pruning)

| | | |
|---|---|---|
| **Correctly classified instances:** | 2,393,791 | (99.7414%) |
| **Incorrectly classified instances:** | 6207 | (0.2586%) |
| **Kappa statistic:** | 0.9948 | |
| **Mean absolute error:** | 0.0038 | |
| **Root mean squared error:** | 0.0482 | |
| **Relative absolute error:** | 0.7575% | |
| **Root relative squared error:** | 9.6378% | |
| **Total Number of Instances:** | 2,399,998 | |

**Detailed accuracy by class:**

| | TP Rate | FP Rate | Precision | Recall | F-Measure | MCC | ROC Area | PRC Area | Class |
|---|---|---|---|---|---|---|---|---|---|
| - | 0.997 | 0.003 | 0.997 | 0.997 | 0.997 | 0.995 | 0.999 | 0.999 | 0 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| - | 0.997 | 0.003 | 0.997 | 0.997 | 0.997 | 0.995 | 0.999 | 0.999 | 1 |
| Weighted Avg. | 0.997 | 0.003 | 0.997 | 0.997 | 0.997 | 0.995 | 0.999 | 0.999 | |

**Confusion matrix:**

| Positive (0) | Negative (1) | Classified as |
|---|---|---|
| 1,196,852 | 3,147 | **Positive (0)** |
| 3,060 | 1,196,939 | **Negative (1)** |

**Computational time:**

The pruned J48 has completed its execution using the 10-folds cross-validation method in 38 minutes (considering both the training phase and the testing phase); the split percentage method needed 3 minutes to complete the training phase and less than 1 second to complete the test phase, achieving an accuracy of 99.6843% in classifying 719,999 instances (30% of the dataset). Despite the high accuracy, J48 should not be used in a similar context, since an increase in the dimension of the dataset would significantly slow down the execution time. Indeed the tree has 15,801 leaf nodes which is a huge number.

## J48 (without pruning)

| | | |
|---|---|---|
| **Correctly classified instances:** | 2,395,161 | (99.7985%) |
| **Incorrectly classified instances:** | 4,837 | (0.2015%) |
| **Kappa statistic:** | 0.996 | |
| **Mean absolute error:** | 0.0027 | |
| **Root mean squared error:** | 0.0426 | |
| **Relative absolute error:** | 0.5438% | |
| **Root relative squared error:** | 8.5125% | |
| **Total Number of Instances:** | 2,399,998 | |

**Detailed accuracy by class:**

| | TP Rate | FP Rate | Precision | Recall | F-Measure | MCC | ROC Area | PRC Area | Class |
|---|---|---|---|---|---|---|---|---|---|
| - | 0.998 | 0.002 | 0.998 | 0.998 | 0.998 | 0.996 | 0.999 | 0.999 | 0 |
| - | 0.998 | 0.002 | 0.998 | 0.998 | 0.998 | 0.996 | 0.999 | 0.999 | 1 |
| Weighted Avg. | 0.998 | 0.002 | 0.998 | 0.998 | 0.998 | 0.996 | 0.999 | 0.999 | |

**Confusion matrix:**

| Positive (0) | Negative (1) | Classified as |
|---|---|---|
| 1,197,188 | 2,811 | **Positive (0)** |
| 2,026 | 1,197,973 | **Negative (1)** |

**Computational time:**

The unpruned J48 has completed its execution in 33 minutes using the 10-folds cross-validation method (considering both the training phase and the testing phase), slightly faster than the pruned version; the split percentage method needed 2 minutes to complete the training phase and less than 1 second to complete the test phase (similar results to the pruned version), achieving an accuracy of 99,6843% in classifying 719,999 instances. We therefore can see that the unpruned version of the J48 is very similar to its pruned counterpart, especially on the accuracy. It is also important to notice that the number of False Positives in the unpruned version is slightly less than the pruned version. The tree sizes are also similar as the unpruned version has 20,351 leaf nodes. Therefore, since both tree models give almost the same result, there is no advantage in selecting one over the other.

## Naive Bayes

**Correctly classified instances:** 2,299,098 (95.7958%)

**Incorrectly classified instances:** 100,900 (4.2042%)

| | |
|---|---|
| **Kappa statistic:** | 0.9159 |
| **Mean absolute error:** | 0.0609 |
| **Root mean squared error:** | 0.1749 |
| **Relative absolute error:** | 12.1881% |
| **Root relative squared error:** | 34.9856% |
| **Total Number of Instances:** | 2,399,998 |

**Detailed accuracy by class:**

| | TP Rate | FP Rate | Precision | Recall | F-Measure | MCC | ROC Area | PRC Area | Class |
|---|---|---|---|---|---|---|---|---|---|
| - | 0.963 | 0.047 | 0.953 | 0.963 | 0.958 | 0.916 | 0.993 | 0.994 | 0 |
| - | 0.953 | 0.037 | 0.963 | 0.953 | 0.958 | 0.916 | 0.993 | 0.993 | 1 |
| Weighted Avg. | 0.958 | 0.042 | 0.958 | 0.958 | 0.958 | 0.916 | 0.993 | 0.993 | |

**Confusion matrix:**

| Positive (0) | Negative (1) | Classified as |
|---|---|---|
| 1,155,725 | 44,274 | **Positive (0)** |
| 56,626 | 1,143,373 | **Negative (1)** |

**Computational time:**

Naive Bayes has completed its execution using the 10-folds cross-validation method in 23 minutes  (considering both the training phase and the testing phase); the split percentage method just needed 2 minutes to complete the training phase and exactly 1 second to complete the test phase, achieving an accuracy of 95.8429% in classifying 719,999 instances. Despite Naive Bayes a bit faster than the decision trees (both pruned and unpruned), it achieved a bit lower accuracy as it had a higher number of false positives and false negatives which led to a drop in the overall accuracy. This should be avoided because, despite being less problematic with respect to a false positive as the number of false negatives is more, a false negative is still a transaction that a human, in real-life applications, would have to check manually.

## Bayesian Belief Networks (Split Percentage)

| | | |
|---|---|---|
| **Correctly classified instances:** | 708,790 | (98.4432%) |
| **Incorrectly classified instances:** | 11,209 | (1.5568%) |
| **Kappa statistic:** | 0.9689 | |
| **Mean absolute error:** | 0.0223 | |
| **Root mean squared error:** | 0.1074 | |
| **Relative absolute error:** | 4.4664% | |
| **Root relative squared error:** | 21.4889% | |
| **Total Number of Instances:** | 719,999 | |

**Detailed accuracy by class:**

| | TP Rate | FP Rate | Precision | Recall | F-Measure | MCC | ROC Area | PRC Area | Class |
|---|---|---|---|---|---|---|---|---|---|
| - | 0.988 | 0.019 | 0.981 | 0.988 | 0.984 | 0.969 | 0.999 | 0.999 | 0 |
| - | 0.981 | 0.012 | 0.988 | 0.981 | 0.984 | 0.969 | 0.999 | 0.999 | 1 |
| Weighted Avg. | 0.984 | 0.016 | 0.984 | 0.984 | 0.984 | 0.969 | 0.999 | 0.999 | |

**Confusion matrix:**

| Positive | Negative | Classified as |
|---|---|---|
| 355,877 | 4,331 | **Positive** |
| 6,878 | 352,913 | **Negative** |

**Computational time:**

Bayesian Belief Network has completed its execution for the split percentage in 4 minutes for training phase and less than 1 second for the test phase, achieving an accuracy of 98.4432% in classifying 719,999 instances. It is a bit slower than the Naive Bayes method but it gives a higher accuracy. The number of false negatives is more than the number of false positives and the model takes more time in comparison to other models to be built. Furthermore, it is computationally heavy to perform the 10-fold cross validation analysis and that is why we reported the split percentage only because we could not run cross validation on our machines. Thus, just like Naive Bayes, it is not a good candidate especially since we have better accuracies with much less computation cost.

## Random Forest

| | | |
|---|---|---|
| **Correctly classified instances:** | 2,396,440 | (99.8517%) |
| **Incorrectly classified instances:** | 3,558 | (0.1483%) |
| **Kappa statistic:** | 0.997 | |
| **Mean absolute error:** | 0.0027 | |
| **Root mean squared error:** | 0.0357 | |
| **Relative absolute error:** | 0.5381% | |
| **Root relative squared error:** | 7.1439% | |
| **Total Number of Instances:** | 2,399,998 | |

**Detailed accuracy by class:**

| | TP Rate | FP Rate | Precision | Recall | F-Measure | MCC | ROC Area | PRC Area | Class |
|---|---|---|---|---|---|---|---|---|---|
| - | 0.998 | 0.001 | 0.999 | 0.998 | 0.999 | 0.997 | 1.000 | 0.999 | 0 |
| - | 0.999 | 0.002 | 0.998 | 0.999 | 0.999 | 0.997 | 1.000 | 1.000 | 1 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Weighted Avg. | 0.999 | 0.001 | 0.999 | 0.999 | 0.999 | 0.997 | 1.000 | 1.000 | |

**Confusion matrix:**

| **Positive (0)** | **Negative (1)** | **Classified as** |
|---|---|---|
| 1,197,903 | 2,096 | **Positive (0)** |
| 1,462 | 1,198,537 | **Negative (1)** |

**Computational time:**

Random Forest algorithm has completed its execution using the 10-folds cross-validation method in 43 minutes (considering both the training phase and the testing phase); the split percentage method needed 3 minutes to complete the training phase and 5 seconds to complete the test phase, achieving an accuracy of 99.8111% in classifying 719,999 instances (30% of the dataset). Despite being slightly slower than J48, it achieves a better accuracy, improving both the False Positives and Negatives rate. Also, the way the forest was constructed is preferable as each tree in the forest contains only 70% of the training set and so it can generalize more than the J48. Despite being way slower than Naive Bayes and Bayesian Belief, it is a very good candidate for our application thanks to its high level of accuracy.

## Hoeffding Tree

| | | |
|---|---|---|
| **Correctly classified instances:** | 2,255,035 | (93.9599%) |
| **Incorrectly classified instances:** | 144,963 | (6.0401%) |
| **Kappa statistic:** | 0.8792 | |
| **Mean absolute error:** | 0.0667 | |
| **Root mean squared error:** | 0.22 | |
| **Relative absolute error:** | 13.3339% | |
| **Root relative squared error:** | 44.0024% | |

**Total Number of Instances:** 2,399,998

**Detailed accuracy by class:**

| | TP Rate | FP Rate | Precision | Recall | F-Measure | MCC | ROC Area | PRC Area | Class |
|---|---|---|---|---|---|---|---|---|---|
| - | 0.938 | 0.059 | 0.941 | 0.938 | 0.940 | 0.879 | 0.987 | 0.987 | 0 |
| - | 0.941 | 0.062 | 0.938 | 0.941 | 0.940 | 0.879 | 0.987 | 0.986 | 1 |
| Weighted Avg. | 0.940 | 0.060 | 0.940 | 0.940 | 0.940 | 0.879 | 0.987 | 0.987 | |

**Confusion matrix:**

| Positive (0) | Negative (1) | Classified as |
|---|---|---|
| 1,126,005,148 | 73,994 | **Positive (0)** |
| 70,969 | 1,129,030 | **Negative (1)** |

**Computational time:**

The Hoeffding Tree algorithm has completed its execution using the 10-folds cross-validation method in 11 minutes (considering both the training phase and the testing phase); the split percentage method needed 1 minute to complete the training phase and less than 1 second to complete the test phase, achieving an accuracy of 94.0353% in classifying 719,999 instances (30% of the dataset). In this case, we obtained better results than Naive Bayes in terms of computational time with equivalent results in terms of accuracy: it is therefore important to underline that the Hoeffding Tree algorithm is meant to be used on data streams and requires a very high amount of data to perform properly. It is also important to remember that this algorithm does not handle concept drift, so if we want to use it in an application for credit card fraud detection, we have to make sure that our data stream is not affected by concept drift (or we have to use the improved CVFDT algorithm).

## K-Nearest-Neighbours (split percentage)

In the following section we present the results of the KNN model using the percentage method to split the dataset into training and test set: due to the high amount of time required by KNN, it is not the right model to use for this project, but it is interesting to see the results provided by it on this kind of dataset. The algorithm has been run using K = 1.

| | | |
|---|---|---|
| **Correctly classified instances:** | 718,626 | (99.8093 %) |
| **Incorrectly classified instances:** | 1,373 | (0.1907 %) |
| **Kappa statistic:** | 0.9962 | |
| **Mean absolute error:** | 0.0019 | |
| **Root mean squared error:** | 0.0437 | |
| **Relative absolute error:** | 0.3815 % | |
| **Root relative squared error:** | 8.7337 % | |
| **Total Number of Instances:** | 719,999 | |

**Detailed accuracy by class:**

| | TP Rate | FP Rate | Precision | Recall | F-Measure | MCC | ROC Area | PRC Area | Class |
|---|---|---|---|---|---|---|---|---|---|
| - | 0.997 | 0.001 | 0.999 | 0.997 | 0.998 | 0.996 | 0.998 | 0.998 | 0 |
| - | 0.999 | 0.003 | 0.997 | 0.999 | 0.998 | 0.996 | 0.998 | 0.997 | 1 |
| Weighted Avg. | 0.998 | 0.002 | 0.998 | 0.998 | 0.998 | 0.996 | 0.998 | 0.997 | |

**Confusion matrix:**

| Positive (0) | Negative (1) | Classified as |
|---|---|---|
| 359,279 | 929 | **Positive (0)** |
| 444 | 359347 | **Negative (1)** |

**Computational time:**

The K-Nearest-Neighbours algorithm has completed its execution using the split percentage method in 9 hours and 58 minutes total for both training (which is nothing because KNN is a lazy learner) and test phase. The computational time, as expected, is way higher than any other classifier: this is due to the nature of KNN itself, that for each instance tries to calculate the distance with its nearest neighbour and gives to that instance the same label of the latter. Since our dataset has a very high number of instances, it is obvious that the time to calculate the distance between one instance of the test set and each instance of the training set is incredibly high compared to the other classifiers. It is important to note that the algorithm obtained the highest accuracy of any other classifier studied, in particular it obtained the best number of both False Positives and False Negatives, but since it is important to give a fast response to frauds to block them as soon as possible, this algorithm does not fit the purpose of the application and therefore it has to be discarded from our selection.

## Other models

We decided to run the same models as before also removing more attributes from the dataset, which could have been considered as redundant, to check if the results in terms of speed would have increased without losing accuracy. More specifically, we removed the merchant, the first and last name, the credit card number and all the non-numeric address information. In general, we encountered better execution times, in particular for J48, which decreased the execution time by more than a half (both pruned and unpruned), but losing performances in terms of overall accuracy (94% against the 99% of the version without attribute removal). Naive Bayes is the algorithm that encountered the smallest improvement in time, actually even gaining some accuracy, so overall for Naive Bayes it would be better to choose a reduced dataset; very small changes have been encountered also in the Hoeffding Tree algorithm, the random forest and the Bayesian Belief Networks. Besides J48 and Naive Bayes, no algorithm really improved its overall performance with the reduction of the dimensionality of our dataset, so we decided not to remove the attributes that may have been considered useless since there was no significant gain from doing that.

## 3.3 Comparison of classification models

To decide which classifier was best for our application, we need not only to consider the overall accuracy, but it is important to consider the False Positives rate: as already mentioned above in the documentation, a False Positive represents a fraudulent transaction that has been classified as non-fraudulent. In such a situation, we will not be able to recognize a fraud and we will not be able to warn the users, resulting in a loss of money on their side. We also have to consider the overall accuracy as a relevant metric in our application domain, because in any case also False Negatives would have to be handled, in a real world application, by a human controller: each fraudulent transaction is in fact controlled and either confirmed or not by a human, so False Negatives (non-fraudulent transaction classified as fraudulent) are still to be avoided, even if they do not cause problems to the final user.

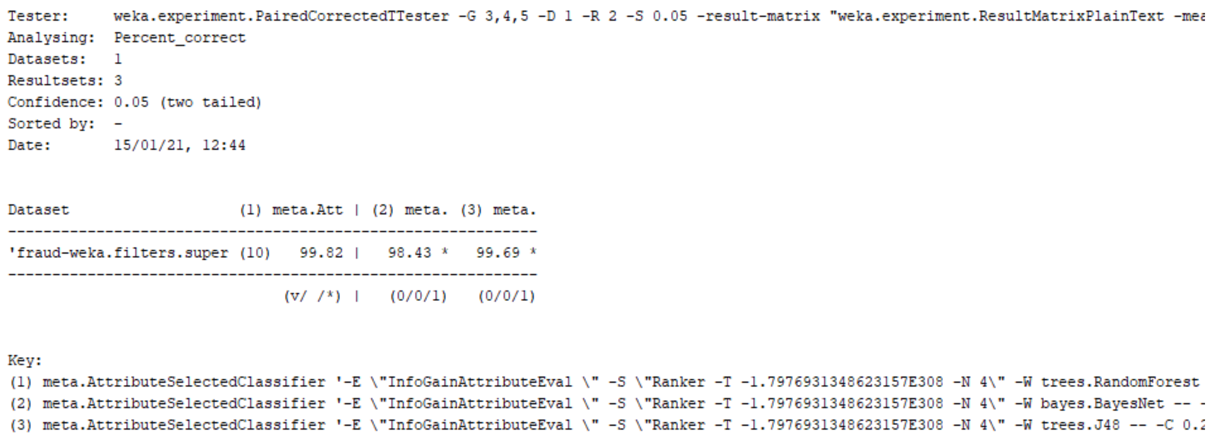
The table below show the results obtained by the different classifiers using the split percentage method (execution time is expressed in minutes):

| Classifier | Accuracy | FP Rate | Precision | Recall | F-Measure | ROC Area | Ex. time |
|---|---|---|---|---|---|---|---|
| **J48** | 99.7414 | 0.003 | 0.997 | 0.997 | 0.997 | 0.999 | 3 |
| **J48 (unpruned)** | 99.7985 | 0.002 | 0.998 | 0.998 | 0.998 | 0.999 | 2 |
| **Naive Bayes** | 95.7958 | 0.042 | 0.958 | 0.958 | 0.958 | 0.993 | 2 |
| **BBN** | 98.4432 | 0.016 | 0.984 | 0.984 | 0.984 | 0.999 | 4 |
| **Random Forest** | 99.8517 | 0.001 | 0.998 | 0.999 | 0.999 | 1.000 | 3 |
| **Hoeffding Tree** | 93.9599 | 0.060 | 0.940 | 0.940 | 0.940 | 0.987 | 1 |
| **1-NN** | 99.8093 | 0.002 | 0.998 | 0.998 | 0.998 | 0.998 | 598 |

Considering just the accuracy and the FP rate, we could be driven to choose 1-NN for our application, but it appears obvious reading the execution time that it cannot be adopted for our purpose. It is therefore interesting to see that the AUC for 1-NN is also lower than (even though the difference is not significant) the AUC for almost all other classifiers, so the goodness of 1-NN in this context can be considered lower than other classifiers also thanks to a statistical measure such as the AUC.

Between the other classifiers, the best one seems to be Random Forest, which has not only the best accuracy and the best FP rate, but also shows the best AUC with respect to all the other classifiers.

To decide the best classifier to pick for static classification, we performed a paired t-test on Weka to understand if there was a statistically significant difference between Random Forest, Bayesian Belief Networks and J48. In the following, we took the Random Forest as base case, with classifier (1) being the Random Forest model, classifier (2) the Bayesian Belief Networks model and classifier (3) the J48 model, run 10 times in the experimenter with the split percentage method (70% of the dataset going in the training set).



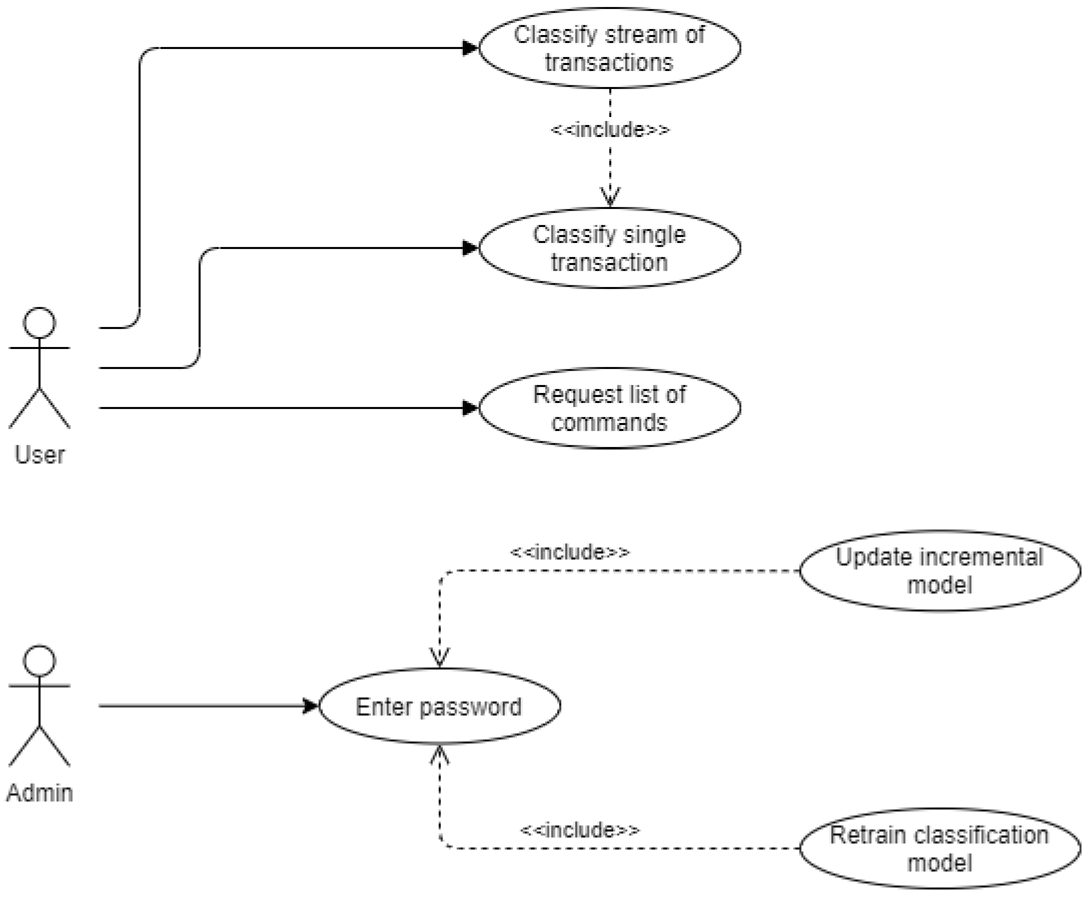
```
Tester:     weka.experiment.PairedCorrectedTTester -G 3,4,5 -D 1 -R 2 -S 0.05 -result-matrix "weka.experiment.ResultMatrixPlainText -mea
Analysing:  Percent_correct
Datasets:   1
Resultsets: 3
Confidence: 0.05 (two tailed)
Sorted by:  -
Date:       15/01/21, 12:44


Dataset                   (1) meta.Att | (2) meta. (3) meta.
-------------------------------------------------------------
'fraud-weka.filters.super (10)   99.82 |   98.43 *   99.69 *
-------------------------------------------------------------
                               (v/ /*) |   (0/0/1)   (0/0/1)


Key:
(1) meta.AttributeSelectedClassifier '-E \"InfoGainAttributeEval \" -S \"Ranker -T -1.7976931348623157E308 -N 4\" -W trees.RandomForest
(2) meta.AttributeSelectedClassifier '-E \"InfoGainAttributeEval \" -S \"Ranker -T -1.7976931348623157E308 -N 4\" -W bayes.BayesNet -- -
(3) meta.AttributeSelectedClassifier '-E \"InfoGainAttributeEval \" -S \"Ranker -T -1.7976931348623157E308 -N 4\" -W trees.J48 -- -C 0.2
```

As we can see, in terms of accuracy and with a confidence level of 0.05, Random Forest results better than both J48 and Bayesian Belief Networks. From the paired t-test, the Random Forest had statistically better results also in terms of AUC and F-Measure, while in terms of FP rate there was not any statistically significant difference, even if the FP rate of Random Forest was the lowest between the tree classifiers. We therefore decided, supported also by the paired t-test, to use the Random Forest model for our application.

# 4. Application

In the following section, important portions of code of the final application are reported to explain the functionalities we have built.

## 4.1 Use case diagram

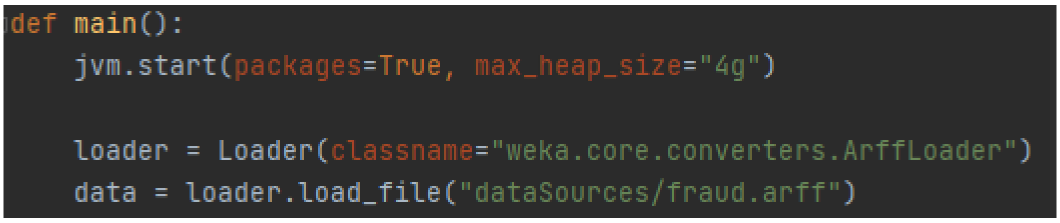


From the use case diagram, we can see that we have 4 main functionalities: retrain the classification model; classify a new instance; classify a stream of instances (simulated by a file) as they arrive; using a stream of instances (simulated by a file) update the classifier incrementally. It is also possible to use the "help" command to ask for the list of commands and the explanation on how to use the commands.

## 4.2 Model retrain

An administrator may want to retrain the model from time to time to update the classifier. In order to do that, the administrator should be able to call the command “retrain” from the user interface and, by inserting a password that ideally just he/she knows, start the retrain process, calling the main() function of the “generateModels.py” module.

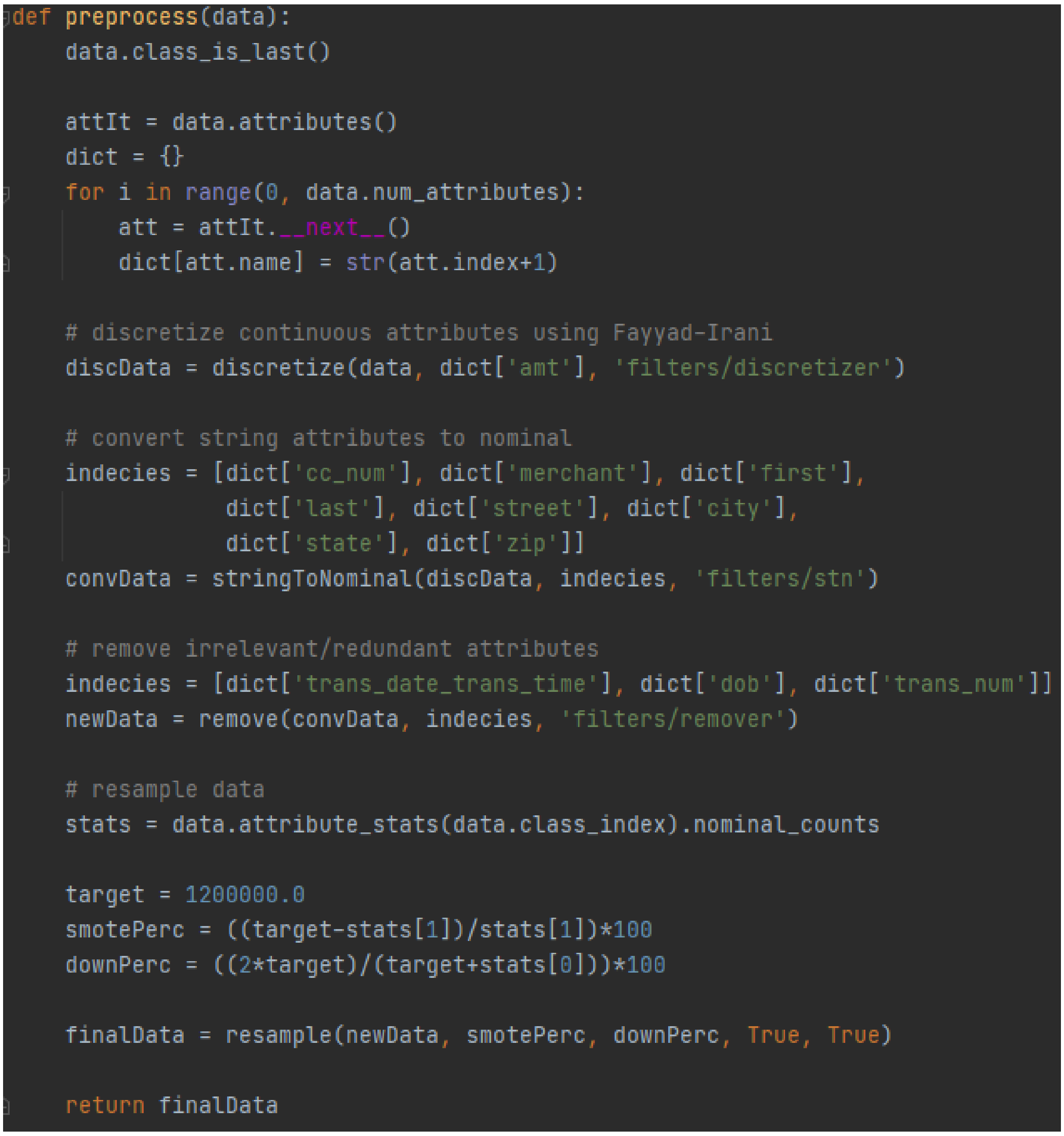
In the main() function, the very first thing we have to do is to start the Java Virtual Machine that will support the *python-weka-wrapper3* library we are currently using. Before doing anything else, we ask for the password, since this command should be run just by an administrator when the system is offline. After receiving the right password, we can declare a loader to load the updated ARFF file with all the transactions done in the past: this file is updated each time a transaction is classified, by appending the transaction with the classification result at the end of the file.



```
def main():
    jvm.start(packages=True, max_heap_size="4g")

    loader = Loader(classname="weka.core.converters.ArffLoader")
    data = loader.load_file("dataSources/fraud.arff")
```

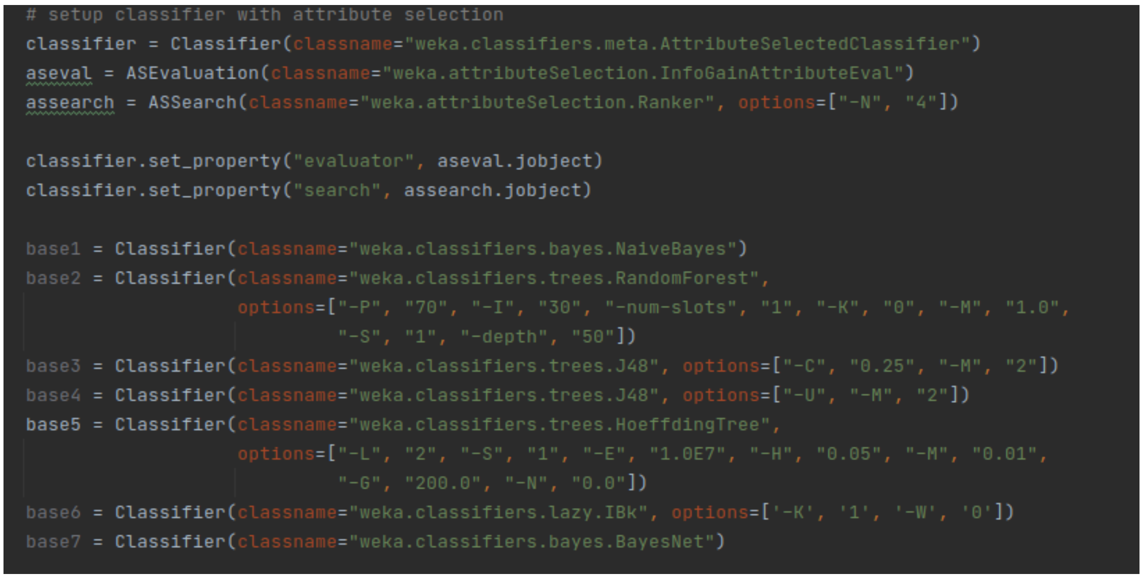
Once the data have been loaded, we perform a pre-processing phase on them: during this phase, beside setting the class as the last attribute, we perform, in order, a discretization step, conversion of string attributes into nominal attributes, we remove irrelevant and redundant attributes and resample the data using both undersampling and oversampling (with SMOTE).

For each filter (with exception for the undersampler and SMOTE), we gave as input a file in which to save the filter, to be used to pre-process the instances we will have to classify without having to build the filters back to scratch: in this way, the pre-processing phase of the instances to be classified will be exactly the same we performed during the retrain process. A detailed description of each step of the pre-processing phase is described in section 3.1.



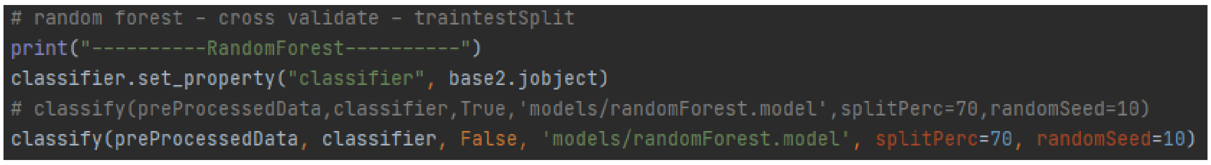
```python
def preprocess(data):
    data.class_is_last()

    attIt = data.attributes()
    dict = {}
    for i in range(0, data.num_attributes):
        att = attIt.__next__()
        dict[att.name] = str(att.index+1)

    # discretize continuous attributes using Fayyad-Irani
    discData = discretize(data, dict['amt'], 'filters/discretizer')

    # convert string attributes to nominal
    indecies = [dict['cc_num'], dict['merchant'], dict['first'],
                dict['last'], dict['street'], dict['city'],
                dict['state'], dict['zip']]
    convData = stringToNominal(discData, indecies, 'filters/stn')

    # remove irrelevant/redundant attributes
    indecies = [dict['trans_date_trans_time'], dict['dob'], dict['trans_num']]
    newData = remove(convData, indecies, 'filters/remover')

    # resample data
    stats = data.attribute_stats(data.class_index).nominal_counts

    target = 1200000.0
    smotePerc = ((target-stats[1])/stats[1])*100
    downPerc = ((2*target)/(target+stats[0]))*100

    finalData = resample(newData, smotePerc, downPerc, True, True)

    return finalData
```

After pre-processing and after printing the class attribute to show the difference in the number of instances, we set up the classifier using Weka's meta AttributeSelectedClassifier: for attribute selection, we use InfoGain combined with Ranker. We then pass to instantiate the actual classifier, that will be the Random Forest, setting the maximal depth of the tree equal to 50 (to avoid overfitting) and the number of trees in the forest equal to 30. In the following picture, we show the aforementioned process, also displaying the setting for all the other classifiers used during the study.



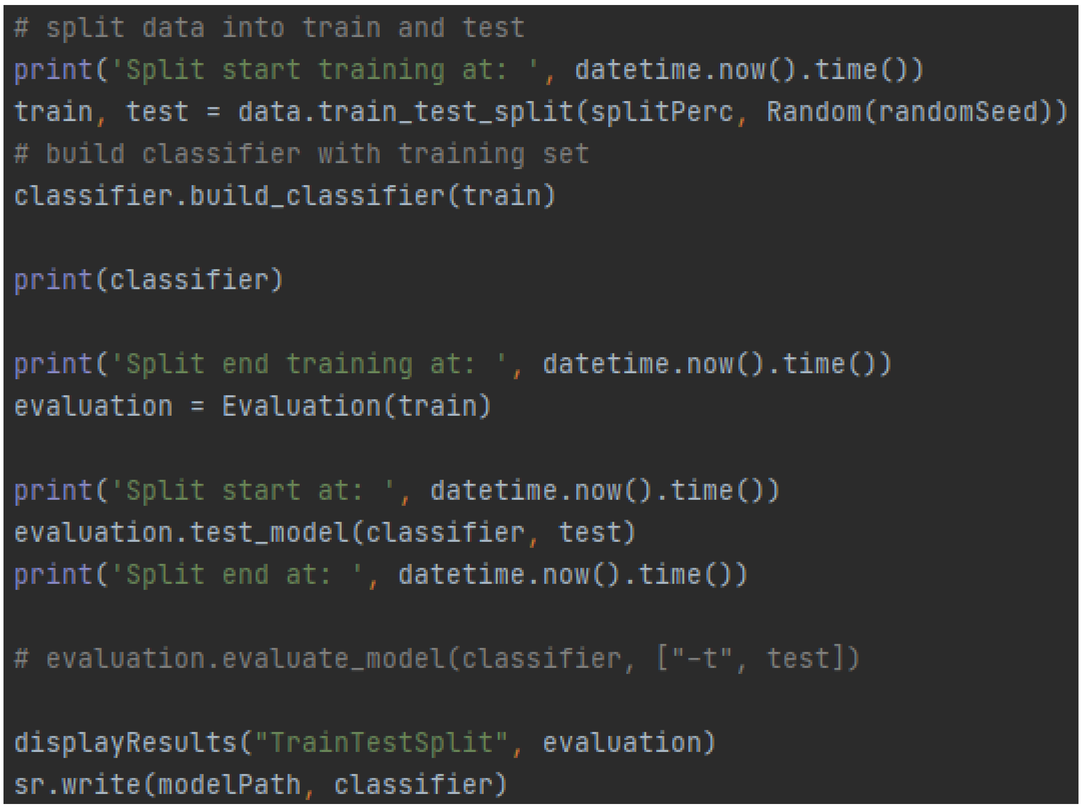
```
# setup classifier with attribute selection
classifier = Classifier(classname="weka.classifiers.meta.AttributeSelectedClassifier")
aseval = ASEvaluation(classname="weka.attributeSelection.InfoGainAttributeEval")
assearch = ASSearch(classname="weka.attributeSelection.Ranker", options=["-N", "4"])

classifier.set_property("evaluator", aseval.jobject)
classifier.set_property("search", assearch.jobject)

base1 = Classifier(classname="weka.classifiers.bayes.NaiveBayes")
base2 = Classifier(classname="weka.classifiers.trees.RandomForest",
                   options=["-P", "70", "-I", "30", "-num-slots", "1", "-K", "0", "-M", "1.0",
                            "-S", "1", "-depth", "50"])
base3 = Classifier(classname="weka.classifiers.trees.J48", options=["-C", "0.25", "-M", "2"])
base4 = Classifier(classname="weka.classifiers.trees.J48", options=["-U", "-M", "2"])
base5 = Classifier(classname="weka.classifiers.trees.HoeffdingTree",
                   options=["-L", "2", "-S", "1", "-E", "1.0E7", "-H", "0.05", "-M", "0.01",
                            "-G", "200.0", "-N", "0.0"])
base6 = Classifier(classname="weka.classifiers.lazy.IBk", options=['-K', '1', '-W', '0'])
base7 = Classifier(classname="weka.classifiers.bayes.BayesNet")
```

Finally, we perform the actual retrain, using the split percentage method with 70% of instances going to the training test: we save the model in the file “randomForest.model”, to be used during classification without having to perform the training again. In the following picture, we can also see the call (commented) to the classify function to create the model using cross-validation.



```
# random forest - cross validate - traintestSplit
print("----------RandomForest----------")
classifier.set_property("classifier", base2.jobject)
# classify(preProcessedData,classifier,True,'models/randomForest.model',splitPerc=70,randomSeed=10)
classify(preProcessedData, classifier, False, 'models/randomForest.model', splitPerc=70, randomSeed=10)
```

In the classify function, we take the split percentage and the random seed passed in the call and create the train and test sets using the train_test_split() function from the API on our dataset; then, we build the classifier, passed again as an argument of the function, using the newly created training set. Using an Evaluation instance, we then evaluate the results obtained from the classification of the test set; finally, we print those final results to show the result of the classification process.



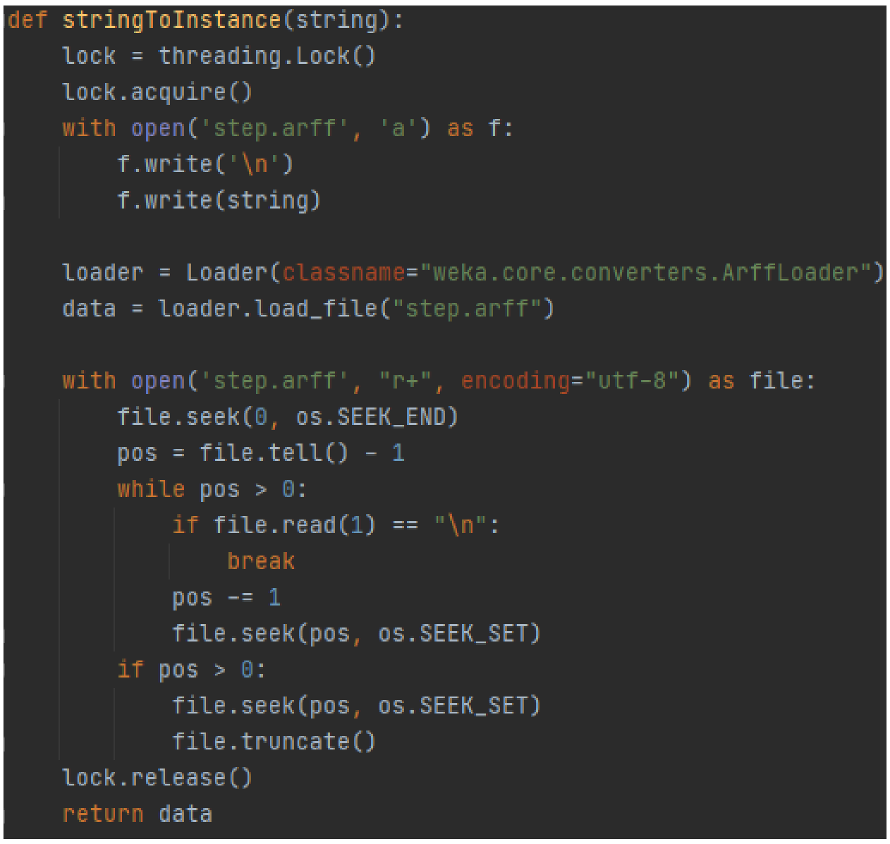
```python
# split data into train and test
print('Split start training at: ', datetime.now().time())
train, test = data.train_test_split(splitPerc, Random(randomSeed))
# build classifier with training set
classifier.build_classifier(train)

print(classifier)

print('Split end training at: ', datetime.now().time())
evaluation = Evaluation(train)

print('Split start at: ', datetime.now().time())
evaluation.test_model(classifier, test)
print('Split end at: ', datetime.now().time())

# evaluation.evaluate_model(classifier, ["-t", test])

displayResults("TrainTestSplit", evaluation)
sr.write(modelPath, classifier)
```

We also print the times where each part of the classification process starts and ends, in order to show how long it took to train and test the retrained model.

## 4.3 Classification of a single instance

A user may want to check if a transaction he/she plans to do may be fraudulent or not; similarly, a user may want to check if a transaction he/she did in the past is fraudulent, because there is no similar tool from his/her bank or in the payment process. The idea is that the user, either before or after the transaction, should be able to receive a detailed description of the transaction with all the information about it: ideally, this description would always have the same format. For simplicity, we assumed that the user is able to retrieve from its bank the description of a transaction, either already completed or not, in a string format corresponding to the format we require. We could also ask to the user to manually input all the information about the transaction, but from the side of the user experience it would be very problematic (the user would have to insert latitude and longitude of the merchant, that are not easily available), so it would make more sense, on the bank side, to generate this string once some of the information about the transaction is inserted by the user, retrieving the missing ones with a quick search.

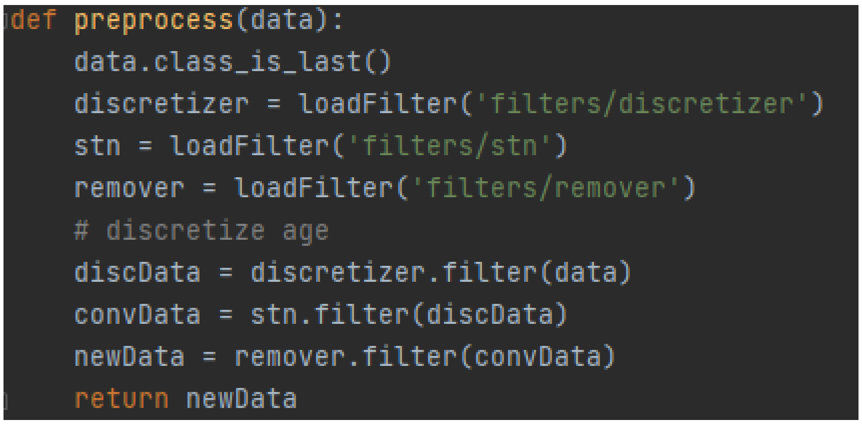
Provided that the user has the string corresponding to the transaction, he/she will have to call the command “classify” from the user interface: the application will call the classifyOne() function from the “useModel.py” module, which will initially ask for the transaction in string format (it is possible to simply copy and paste the transaction in the terminal). The first thing performed by the classifyOne() function is to turn the string into an instance of a dataset: unfortunately, the library is not complete, so there is not a simple way to turn a string or a list to an instance of a dataset, so what we did is to use a support ARFF file where we create a 1-instance dataset to be processed, as we can see in the following picture:



```python
def stringToInstance(string):
    lock = threading.Lock()
    lock.acquire()
    with open('step.arff', 'a') as f:
        f.write('\n')
        f.write(string)

    loader = Loader(classname="weka.core.converters.ArffLoader")
    data = loader.load_file("step.arff")

    with open('step.arff', "r+", encoding="utf-8") as file:
        file.seek(0, os.SEEK_END)
        pos = file.tell() - 1
        while pos > 0:
            if file.read(1) == "\n":
                break
            pos -= 1
            file.seek(pos, os.SEEK_SET)
        if pos > 0:
            file.seek(pos, os.SEEK_SET)
            file.truncate()
    lock.release()
    return data
```

In this function, the first thing we do is to acquire a lock so that we can write and read from the support ARFF file without the risk of spurious writes and delete on it. After the lock is acquired, we write the string to the support ARFF file step.arff (the string will already be in the format we need), then we load the dataset using an ArffLoader instance. Once loaded the dataset, we no longer need the Arff file, so we can delete the instance we wrote in it and release the lock, so that other transactions can be processed.

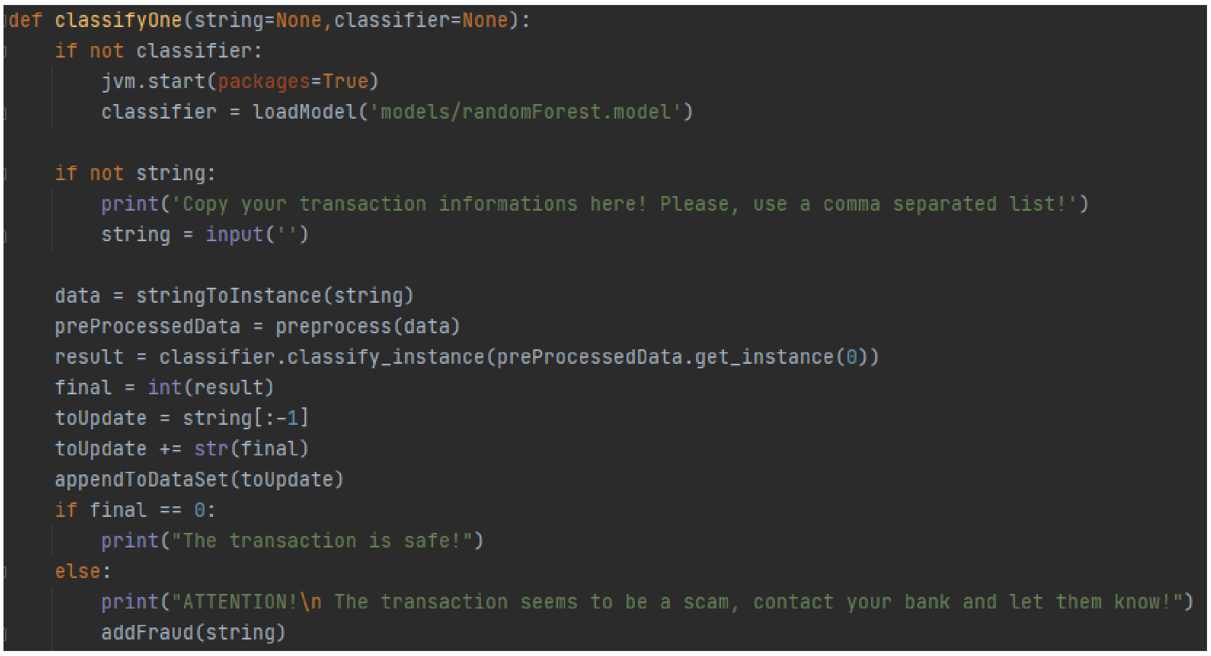
Since the string will give all the information as the ones in the original dataset, we will have to perform a pre-processing phase: here, we use the filters created during the retrain phase in order to have the same exact classification for all the data.



```
def preprocess(data):
    data.class_is_last()
    discretizer = loadFilter('filters/discretizer')
    stn = loadFilter('filters/stn')
    remover = loadFilter('filters/remover')
    # discretize age
    discData = discretizer.filter(data)
    convData = stn.filter(discData)
    newData = remover.filter(convData)
    return newData
```

At this point, we can perform the actual classification of the instance by calling the classify_instance function of the classifier, that is our pre-trained Random Forest: we then get the result of the classification and perform a print on the terminal to either reassuring the user that the transaction is safe or warn him/her that the transaction may be a fraud.

Finally, we save the new transaction in the general ARFF file containing all the transactions and used to retrain the classification model, appending to the end of the string the result of the classification. In the following picture, we can see the classifyOne() function in its whole: it is important to underline that in the end, taking the original string, we are not picking the last character. This character will be a placeholder character that has to be inserted by the bank to simulate the presence of the class: if not present and we do not set the class attribute for the dataset, the classifier will not work. After obtaining the result from our tool, we change the placeholder value with our prediction and append the complete string to the fraud.arff file. If a transaction is found to be fraudulent, we also write the corresponding string into a text file, in order for a human controller to check it asynchronously.



```python
def classifyOne(string=None,classifier=None):
    if not classifier:
        jvm.start(packages=True)
        classifier = loadModel('models/randomForest.model')

    if not string:
        print('Copy your transaction informations here! Please, use a comma separated list!')
        string = input('')

    data = stringToInstance(string)
    preProcessedData = preprocess(data)
    result = classifier.classify_instance(preProcessedData.get_instance(0))
    final = int(result)
    toUpdate = string[:-1]
    toUpdate += str(final)
    appendToDataSet(toUpdate)
    if final == 0:
        print("The transaction is safe!")
    else:
        print("ATTENTION!\n The transaction seems to be a scam, contact your bank and let them know!")
        addFraud(string)
```

## 4.4 Classification of a stream of instances

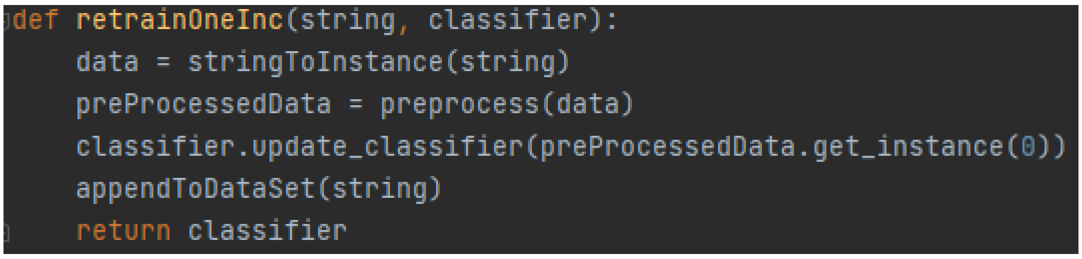
In real world applications, the new transactions are passed to a machine learning model built for credit card fraud detection in order to understand if the transaction is fraudulent or not. Generally, during the creation of the transaction, all the information useful to the model is retrieved by the bank software and merged together in the format required to the classifier, in order for it to process them as they come. Since we do not have a real stream of transactions, we simulate with a text file containing some transactions, that are fetched one by one and processed by the classifier as it would do with a real stream of data.

From the main user interface, we can use the command “stream0” to start the stream processing: the command will call the function stream() from the “useModel.py” module, passing the stream.txt file and the parameter 0 as option, meaning that in this case we only want to classify the instances without updating the classifier incrementally. Passing 1 as parameter for the option argument, we can run the same function asking to also retrain the model incrementally.

In the stream() function, after starting the JVM, the first thing that we do is to load the model for the classification: since here we are (ideally) working with a stream of data, the model we are using is the Hoeffding Tree model. After loading the model, we just read the transactions one by one from the file and perform the classifyOne()

function if the option is 0, already used to classify a single instance, passing as arguments the string corresponding to the transaction and the Hoeffding tree model to be used for classification. This process simulates the handling of a stream of data, where each transaction has to be processed by its own in the fastest time possible.

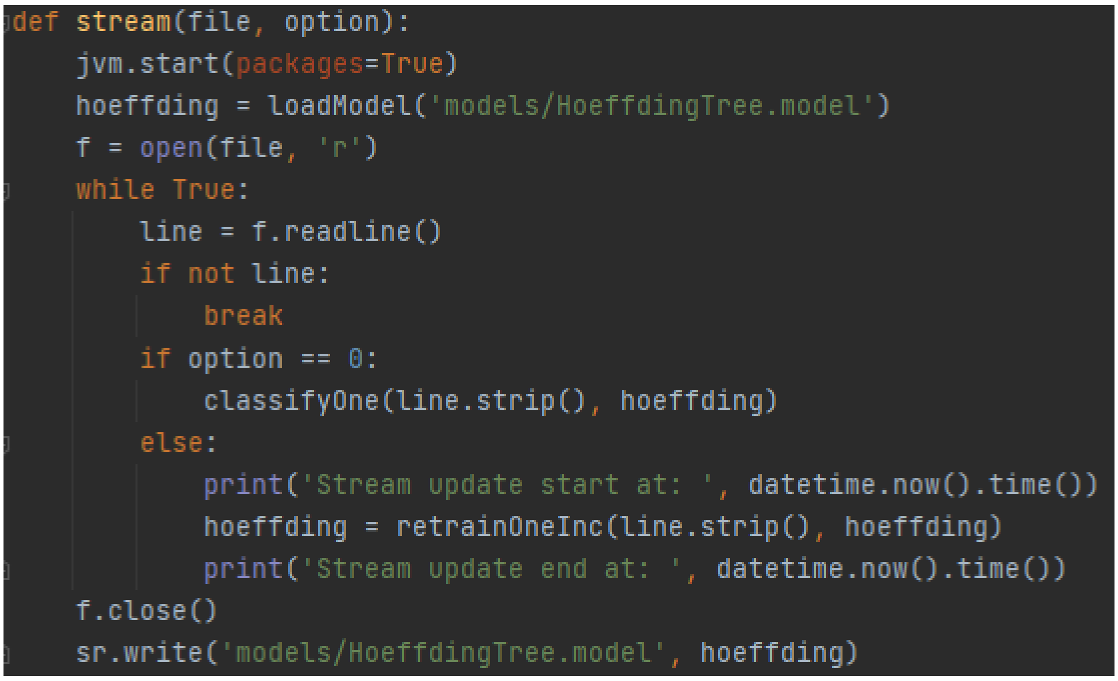
Since Hoeffding Tree is an incremental classifier, we can also decide to update the classifier with the incoming instances from the datastream considering each instance already correctly classified (so human control has already been performed), using the command "stream1" from the main user interface. If invoked, this command run again the stream() function, but instead of calling the classifyOne() function, the retrainOneInc() function is called, defined as follows:



```
def retrainOneInc(string, classifier):
    data = stringToInstance(string)
    preProcessedData = preprocess(data)
    classifier.update_classifier(preProcessedData.get_instance(0))
    appendToDataSet(string)
    return classifier
```

This function is very similar to classifyOne(), but instead of using the classify_instance() function of the classifier, we use the update_classifier() to update the Hoeffding Tree model: in this way, it is not required to perform a retrain for the Hoeffding Tree model (even if it is suggested to better handle concept drift over time, as stated in "Credit Card Fraud Detection: a Realistic Modeling and a Novel Learning Strategy" from A. Dal Pozzolo, G. Boracchi et al.). In both cases, the execution time of the overall process is very low, so we can handle a high amount of incoming transactions thanks to this model.

In the following picture it is possible to see the implementation of the stream() function, in which we want to highlight the stream update timing with the two print() functions in the else statement.



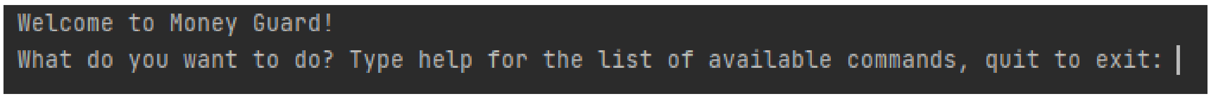
```
def stream(file, option):
    jvm.start(packages=True)
    hoeffding = loadModel('models/HoeffdingTree.model')
    f = open(file, 'r')
    while True:
        line = f.readline()
        if not line:
            break
        if option == 0:
            classifyOne(line.strip(), hoeffding)
        else:
            print('Stream update start at: ', datetime.now().time())
            hoeffding = retrainOneInc(line.strip(), hoeffding)
            print('Stream update end at: ', datetime.now().time())
    f.close()
    sr.write('models/HoeffdingTree.model', hoeffding)
```

Once we have classified all instances and exited the while loop, we save the new Hoeffding Tree model to be reused when required.

It is worth mentioning that our application is a very good candidate to be integrated into banking applications in order to notify users about the possibility that a certain transaction done through their accounts are fraudulent.

# 5. User manual

In the following section we want to show, using some screenshots from the user interface, how the application works in its main functionalities. First of all, we can start the application running the “main.py” module, from which we will be able to see a prompt waiting for a command to be input from the user, as in the following picture.

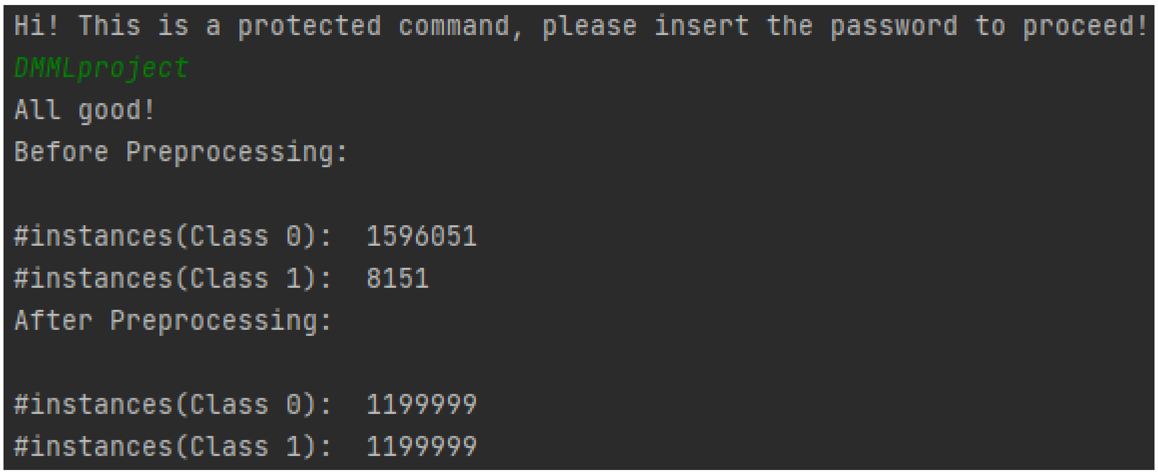


```
Welcome to Money Guard!
What do you want to do? Type help for the list of available commands, quit to exit: |
```

Typing help, a list of commands will be displayed with a small description for all of them.

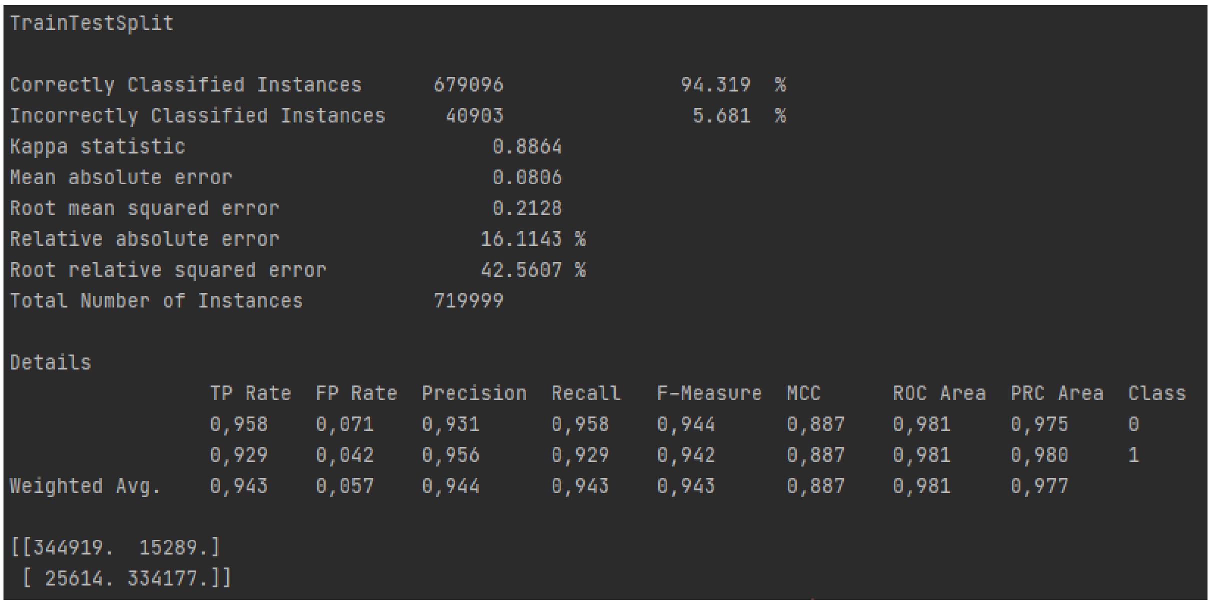
## “Retrain” command

Using the command “retrain”, once the JVM is started we will be asked for a password in input: once the password is correctly entered, we will be able the distribution of instances in the classes before and after the preprocessing phase, that highlights the rebalancement performed.



```
Hi! This is a protected command, please insert the password to proceed!
DMMLproject
All good!
Before Preprocessing:

#instances(Class 0):  1596051
#instances(Class 1):  8151
After Preprocessing:

#instances(Class 0):  1199999
#instances(Class 1):  1199999
```

At this point, the model is retrained using the percentage split method (as always, 70% of the dataset is used for the training set) and the results of the classification performed on the test set are returned, along with the attribute selected during the attribute selection (for Random Forest, they are the category of the merchant and the amount of money involved in the transaction). The results of the classification are shown in the following format:

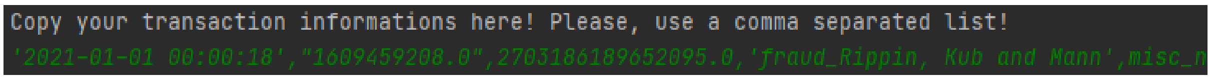


```
TrainTestSplit

Correctly Classified Instances      679096              94.319  %
Incorrectly Classified Instances     40903               5.681  %
Kappa statistic                          0.8864
Mean absolute error                      0.0806
Root mean squared error                  0.2128
Relative absolute error                 16.1143 %
Root relative squared error             42.5607 %
Total Number of Instances           719999

Details
                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 0,958    0,071    0,931      0,958    0,944      0,887    0,981     0,975     0
                 0,929    0,042    0,956      0,929    0,942      0,887    0,981     0,980     1
Weighted Avg.    0,943    0,057    0,944      0,943    0,943      0,887    0,981     0,977

[[344919.  15289.]
 [ 25614. 334177.]]
```

Along with all of these information, the times for the training and test phases are shown.

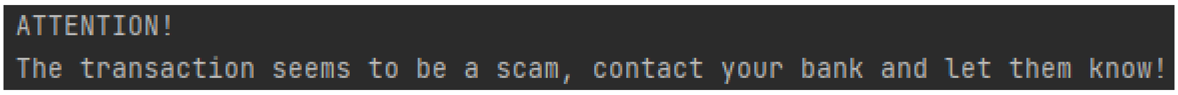
## "Classify" command

From the terminal, we can invoke the "classify" command to classify a single instance: after calling it (the JVM may require some seconds to start), it is required to insert the transaction information. These can be copied and pasted from a txt file directly into the terminal.



```
Copy your transaction informations here! Please, use a comma separated list!
'2021-01-01 00:00:18',"1609459208.0",2703186189652095.0,'fraud_Rippin, Kub and Mann',misc_n
```

After a few seconds, the classification should be finished and the result will be shown as a line of text, telling the user that the transaction is either safe or fraudulent.



```
ATTENTION!
The transaction seems to be a scam, contact your bank and let them know!
```

After the execution of this command, we can check that, as the last line of our fraud.arff file where all the transactions are stored, we will have the new transaction saved.

## “Stream0” and “Stream1” commands

The execution of the command “stream0” is similar to the execution of the command “classify”, with the display of the final classification of all the instances. The transactions detected as fraud are also added to the fraudTrans.txt file to be examined by a human controller. For the “stream1” command, we can see the time spent to classify each instance using the Hoeffding Tree.