UNIVERSITY OF FLORENCE

DEPARTMENT OF COMPUTER SCIENCE

DCML Project

Play-Style Classifier

A Machine Learning Approach, Anomaly Detection

Student:

Filippo Di Martino

Chapter 1

Introduction

The Play-Style classifier predict which user is playing the video game based on their mouse and keyboard actions. By analyzing the patterns of mouse movements and keyboard inputs, the program employs machine learning techniques to distinguish between different play styles and therefore different people. The program keeps count of how many times a specific key has been pressed in a predetermined unit of time. The keys taken into account are part of both the keyboard and the mouse of the player. Based on the data collected and analyzed, a prediction is made as to whether the owner ([0] base case) or another person ([1] anomaly case) is playing the game.

Chapter 2

Project Structure

The project is structured into two basic folders, Monitoring and ML. On GitHub there are also two other folders, one dedicated to the images saved during the tests (Images) and the other to do injection tests even if they were not later used in the final test (Injection). There are also two files (readme.md and requirements.txt) for the documentation needed to describe how to use the code.

```
/DCML_Project
├── README.md
├── main.py
├── requirements.txt
├── /Monitoring
│   ├── CompleteMonitor.py
│   ├── KeyboardMonitor.py
│   ├── MouseMonitor.py
│   └── StatsMonitor.py
├── /ML
│   ├── /Data (contains the dataset.csv)
│   ├── /TrainedModels (models trained with all the features)
│   ├── /TrainedModelsNoCPU_RAM (models trained without CPU and RAM features)
│   ├── SupervisedModelComparison.py
│   ├── UnsupervisedModelComparison.py
│   ├── prepareDataset.py (create the dataset from multiple matches)
│   └── visualizeDecisionTree.py (to see how the dataset is split)
└── Images (contains all images)
```

Chapter 3

Monitoring

The first part of the project focuses on system monitoring. There are four files.py in the monitoring folder. Three of these are responsible for monitoring different parts of the system: one dedicated to the keyboard, one to the mouse, and one to computer statistics. The monitor related to the Keyboard keeps track of how many times the keys were pressed in the last time unit; the keys taken into account are those used during the game. The monitor related to Mouse in addition to keeping track of the clicks in the last time unit, it also keeps track of the position of the mouse in the last instant. The statistics-related monitor monitors CPU and RAM usage in the last instant. The last remaining file "CompleteMonitor.py" is a class responsible for coordinating the three monitors described above as they run concurrently on 3 different threads. the results obtained are then collected and merged into a single variable to be used for saving and/or analysis.

3.1 Stats Monitor

The StatsMonitor class has the task of monitoring system resources, focusing mainly on "CPU" and "MEMORY." To perform this function, it makes use of the *psutil* library. This choice was driven by the demonstration provided during the course on the use of this library. Also included in the file is the *Timer* library, used to schedule the reading of system resource values on a separate thread. The decision to use the *Timer* library was dictated by its intuitiveness of use and its ability to automatically handle the creation and management of the thread, scheduling over time the next function call to be executed.

3.2 Keyboard Monitor

The KeyboardMonitor class has the task of monitoring key press. A total of 10 keys are considered, representing the main keys during game use. The monitor uses the same *Timer* library to schedule activities and the *pynput* library to detect the keys pressed. It reads the keys at regular intervals and counts how many times and which

keys were pressed in the last interval. As a result, an array is obtained in which for each key there is an integer indicating how many times it was pressed during the last interval.

3.3 Mouse Monitor

The MouseMonitor class is responsible for monitoring the position of the mouse and the number of clicks made. Similar to keyboard management, this class makes use of the *Timer* and *pynput* libraries. A peculiarity of this class is that it shall not only count the number of clicks (as it does in the KeyboardMonitor class), but also take an instantaneous reading of the mouse position (similar to StatsMonitor).

3.4 Complete Monitor

The CompleteMonitor class is responsible for managing the outputs of the three monitors previously described. It acquires data from the queue of each of the three monitors, combines them into a single variable and writes to a CSV file (monitored_values.csv) a row with all the monitored values at each time interval. It also handles communication with the prediction model through a function called get_realTime_data(), which returns the last monitored values in a dataframe format.

Chapter 4

Machine Learning

The second part of the project focuses on machine learning. Within this folder are the trained models, both with all the features and without the computer statistics (CPU and MEMORY). In addition, there are three crucial files: two dedicated to the selection and comparison of the models considered, and another containing the dataset used.

4.1 Dataset

The dataset used consists of 4 games played, two of which were played by me and two by my girlfriend. The games I played are labeled as "normal" with value "0", while the other two games are labeled as "anomalies" with value "1". I choose this method because i had no chance to simulate the play-style of another person by injecting the pressing of the keys or the movement of the mouse. One way to avoid this could be to label the anomaly some rows that a human can not possibly perform, but that was not the intention of the project.

The total monitored values are 17, but for training purposes 16 were used in one case and 14 in the other. In the first case the timestamp value was dropped, in the second were dropped cpu, memory and timestamp.

1	TIMESTAMP	CPU	MEMORY	MOUSE_X	MOUSE_Y	LEFT_CLICKS	RIGHT_CLICKS	q	w	e	r	t	d	f	tab	space	ctrl	anomaly
2	1.7044144549664904e+18	13.8	60.5	-380	-174	0	0	0	0	0	0	0	0	0	0	0	0	0
3	1.7044144559844685e+18	12.2	60.6	-160	454	0	0	0	0	0	0	0	0	0	0	0	0	0
4	1.7044144575134177e+18	22.8	60.6	1113	585	0	2	0	0	0	0	0	0	0	0	0	0	0
5	1.7044144580244214e+18	14.6	60.6	1169	558	0	1	0	0	0	0	0	0	0	0	0	0	0
6	1.7044144595488077e+18	14.5	60.4	1169	558	0	0	0	0	0	0	0	0	0	0	0	0	0
7	1.7044144600627858e+18	28.1	60.4	1169	558	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4.1: small shot of the dataset

The complete dataset was created by alternating the matches in order, putting one after the other. During the training phase of the model, the dataset is shuffled.

4.2 Supervised Comparison

Six different models were included in the file *ML/SupervisedModelComparison.py* during the comparison phase of the considered models. The selection of models was straightforward since they are all models that have been studied during the course of study. Specifically, the models considered are:

RandomForest, *KNeighbors*, *GaussianNB*, *LinearDiscriminantAnalysis*, *LogisticRegression*, and *MultiLayerPerceptron*.

Evaluation of the models was based on accuracy, recall, and F1 score. Initially, the models were trained using the predefined parameters to determine which model obtained a better initial score.

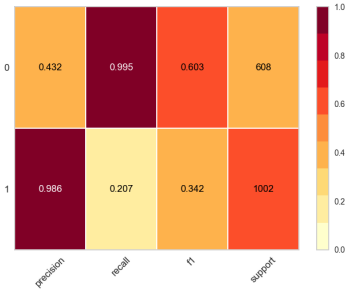

Figure 4.2: Gaussian NB

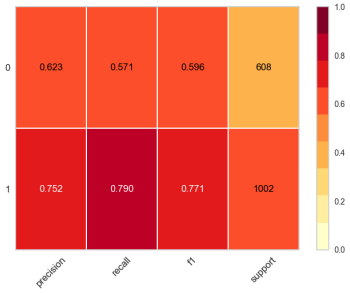

Figure 4.3: KNeighbors

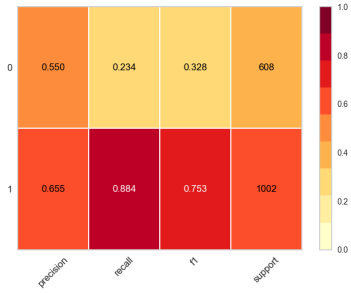

Figure 4.4: LDA

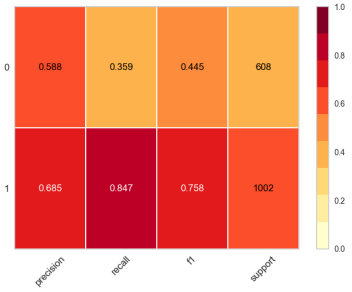

Figure 4.5: Logistic Regression

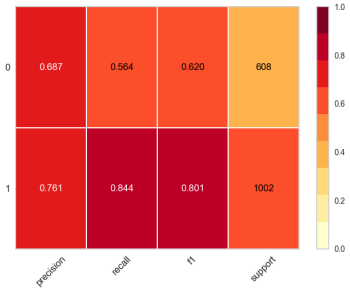

Figure 4.6: MultiLayerPerceptron

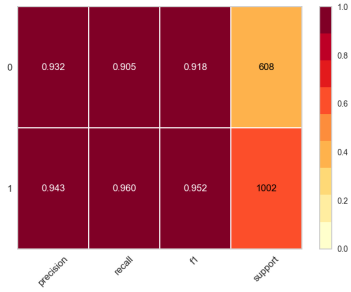

Figure 4.7: RandomForest

As we can observe from the images, the best performing model in all fields is the *Random Forest* model, as it scored above 90 percent in all metrics evaluated. The column *support* represents the number of samples, corresponding to the number of rows in the dataset, marked as anomalous or normal, respectively.
Once the results were examined, a search was performed on which parameters were best suited for the model. Therefore, I decided to create a decision tree to identify on which features the splits were made. The first result obtained is as follows:

After noticing the overfitting problem and working to reduce the size of the tree, I modified the parameters such as min_samples_split and min_impurity_decrease to get less overfitted and more generalized decision trees. During this phase, I found

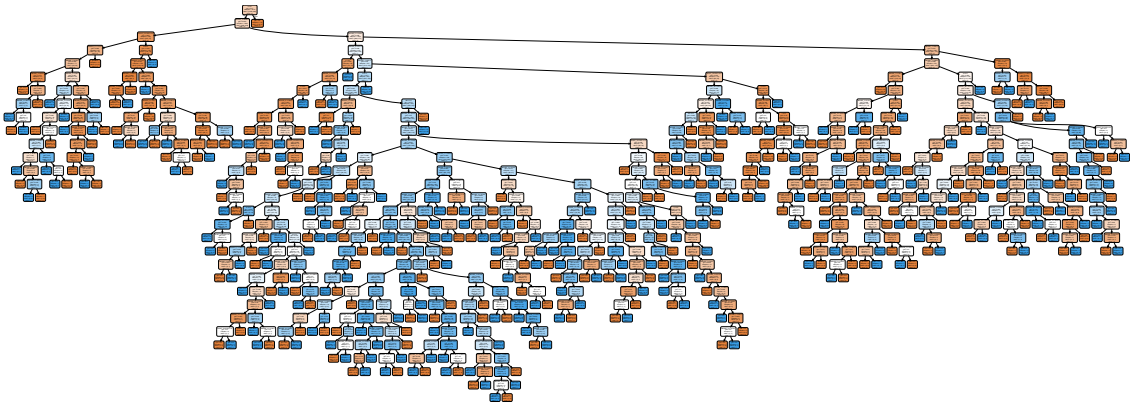

Figure 4.8: Decision Tree without parameters setted

that the tree made splits mainly based on CPU and RAM values. However, the choice to classify a row in the dataset as anomaly or normal should not depend on these values, since one cannot know whether, at that precise moment, the system is performing additional operations such as updates or installations.

Consequently, I decided to remove the CPU and RAM features and repeated the tests, printing the decision trees again. The metrics obtained are as follows:

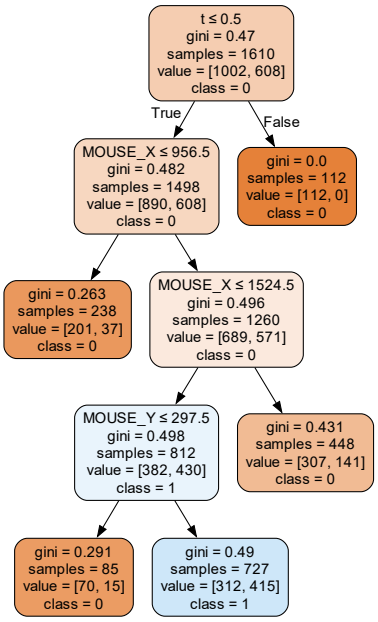

Figure 4.9: decision tree with parameters

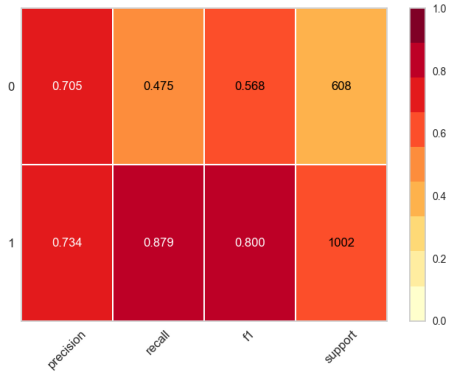

Figure 4.10: Random forest with parameter without CPU and MEMORY

After noticing a decrease in performance and a general reduction in the decisional tree size caused by the values put into the parameters, I chose to limit the final tree depth to 8, allowing the model to decide which splits to make more accurately. This

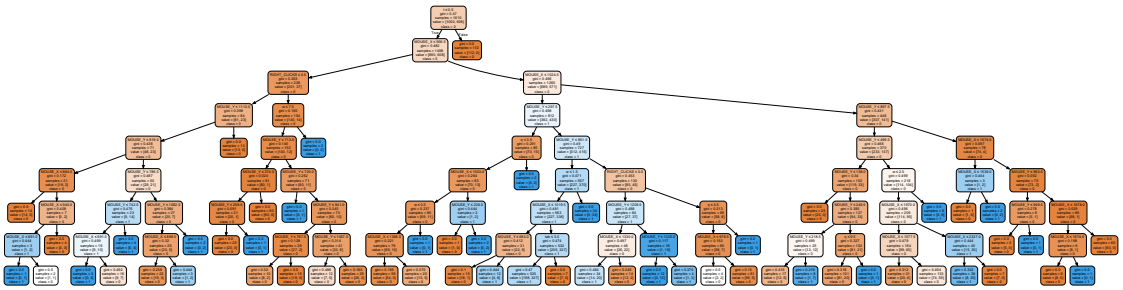

Figure 4.11: Decision tree max depth = 8

led to good results while maintaining a good degree of generality. The final tree has dimensions similar to what is shown in the image 4.11.

4.2.1 Mouse distance

I also tried transforming the dataset, focusing mainly on the information retrieved from the mouse. One of the modifications I tested was to calculate the distance between points recorded by the mouse for each time interval. This modification resulted in a performance increase of a few percentage points (from .69 to .74) compared to keeping only the coordinates. However, I did not investigate this transformation further because calculating the actual distance required complex work. The distance traveled by the mouse depends not only on the initial and final position, but also on other factors such as the speed and acceleration of the movement. Implementing this calculation would have required changing the time interval, creating a new dataset and re-training the models, making the project too complex.

4.3 UnSupervised Comparison

Two different models were considered in the *ML/UnsupervisedModelComparison* file: *ABOD* and *HBOS*. Similar to the supervised models, I tried training these models initially without changing the parameters, thus keeping the default settings. However, the metrics obtained were poor and not very encouraging.

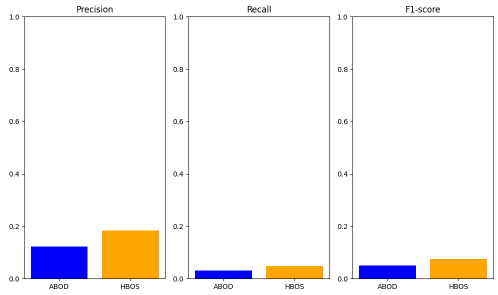

Figure 4.12: blue: ABOD, orange: HBOS

```
ABOD:
Precision: 0.12179487179487179
Recall: 0.03125
F1-score: 0.049738219895287955

HBOS:
Precision: 0.18354430379746836
Recall: 0.047697368421052634
F1-score: 0.07571801566579635
```

Figure 4.13: precision, recall and f1 score of ABOD and HBOS

I decided to not investigate further on the unsupervised models because of the initial metrics.

Chapter 5

RealTime Analisys

The integration between the monitor and the prediction model is handled through two files: main.py and CompleteMonitor.py. The CompleteMonitor class is responsible for acquiring data from the three monitors, transforming it into a format that can be analyzed by the model, and managing the data set. In the main.py file, the monitor with an interval of 0.5 seconds is instantiated (monitoring performed 2 times per second). Depending on the value of the variable monitor_only, the program decides whether to perform only monitoring or also prediction of the results. Prediction is done by loading the pre-trained model into a variable and using that model recursively at each time interval to predict the value received from the monitor. If more than 5 consecutive outlier values are recorded, the program starts to beep until the value returns to normal. This choice was made because in similar game situations or moments of stalemate, players' behaviors may be very similar, thus being misleading to base the classification as an anomaly on a single line. Accordingly, the "alarm" is triggered if anomalies are recognized for more than 2.5 consecutive seconds. In console, the program output will be [0] or [1] each time the model predicts the outcome, triggering the sound if you have more than 5 times the prediction [1] in a row

At the end of the program (e.g., on keyboard interruption or pressing "stop" from the IDE), the monitored data will be saved within a CSV file.

Chapter 6

Results and Discussion

The way the program works is clearly highlighted in the video available on Google Drive. The video shows the game, the person who is playing it, and the real-time results of the model's prediction (displayed in the lower right corner). During my girlfriend's gameplay, the model repeatedly emits warning signals for a prolonged period as it recognizes her play style as an anomaly. In contrast, during my gameplay, the model only occasionally predicts some lines as anomalous, but not enough to trigger the alarm. During the test, the Random Forest classifier was used with a maxDepth parameter set to 8.
It is important to mention the difference in accuracy obtained by including CPU and MEMORY usage information in the dataset.

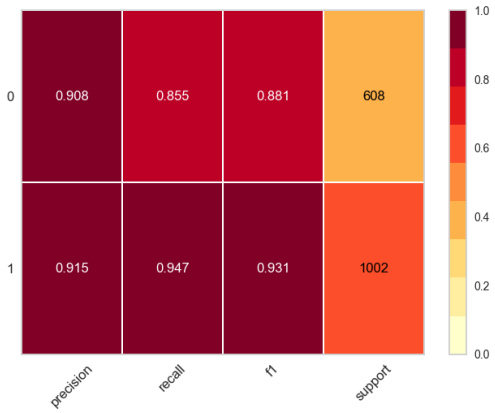

Figure 6.1: all features

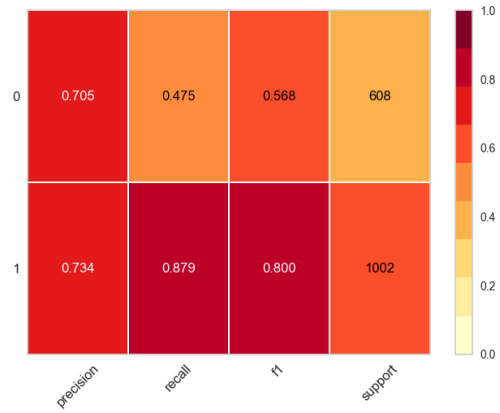

Figure 6.2: no CPU and MEMORY

Although the values obtained with all features are objectively better and given our complete knowledge of the dataset domain, it is critical to consider which of these values are actually relevant. Including CPU and memory usage information may increase evaluation metrics, but what it actually does is that it biases the program to run only in a specific situation and on a specific system. PC statistics vary by model and component performance and also by system state. To ensure the usability of the program on other systems, it is necessary to remove this data.

For example, during a test with the model that considered all features, the results were very different from what was expected. This is due to the fact that to record the results (the video), it is necessary to use a program to capture the screen image, which results in increased CPU and memory usage. So even though the evaluation metrics are higher, during this test the "alarm" sound went on only once after numerous attempts.