Author: Marco Agatensi
Student ID: 7172615
email: marco.agatensi@edu.unifi.it

PROJECT REPORT

Objective of the Project

The final goal of this project is to develop an Anomaly Detector to use at run time to detect ongoing anomalies in my laptop.
In short, this project can be divided in three main parts:

1. System Monitoring for data collection (choose the parameter to monitor, choose the injections to perform, build the monitor, build the injector, gather data when the system is in a normal state and when it is under injection);
2. Training of a binary classifier (load the dataset and preprocess data, choose machine learning algorithms to train, train them and save the one with better performance);
3. Integration of the monitor with the classifier to develop the final anomaly detector

1 – System Monitoring

A monitor is a process that reads some values of the system under monitoring. First of all, to build a monitor, it is then necessary to define what resources have to be monitored: I chose to monitor the CPU and the Virtual Memory using the python library *psutil*. In detail the monitor reads data about:

- CPU → For each logical core (8 cores) are monitored the times (percentage), frequencies and percentages of usage. For the entire CPU is monitored the frequency, while the percentage of usage is obtained by averaging the utilization percentage of each individual logical core to avoid inconsistencies between the utilization percentages of individual cores and the one of the entire CPU (which would be obtained through measurements at different instants of time). For each physical core (4 cores) the temperature is monitored.
- Virtual Memory → all information that can be read about the Virtual Memory using *psutil.virtual_memory()* function are monitored (memory available, free, used, inactive, buffers, cached, shared, ecc).

To make the code as reusable as possible, I wrote the code so that it was organized into classes and each of them has some parameters that allow for some customization during the creation of objects.
The monitor is represented by the *SystemMonitor* class (file SystemMonitor.py). The constructor of this class has some parameters that allow the user to build different types of monitors. The monitor also saves the current state of the system using the enum *SystemState* (*SystemState.NORMAL* or *SystemState.UNDER_INJECTION*) that is defined in the file SystemState.py.
To gather system data, the class *SystemMonitor* has the method *monitor()* that returns a dictionary which represents a data point (set of monitored parameters at a certain instant of time and in additions the name of the current injection, that has value “None” if there is no ongoing injection). This class also provide the method *start_injection(str)*, that has to be called before starting an injection to set the current state of the system to *SystemState.UNDER_INJECTION* and to save the type of the current injection. When an ongoing injection stops, the other method *end_injection()* has to be called to set the state of the system to *SystemState.NORMAL* and to set the current injection to “None”.
The injection of anomalies is done using the *InjectionManager* class which makes use of the *LoadInjector* class to load the injections. The user, to inject anomalies into the system has to create an instance of the *InjectionManager* class passing as arguments to the constructor at least the path to the json file that contains the list of injectors and the estimated duration of each injection. The *InjectionManager* class has the method *read_injectors(bool)* that reads the json file and save the injectors into a list. If the object of this class has been created passing the parameter *inj_number* with a value greater than the number of injectors in the json file, then additional injectors, randomly chosen from those already present in the list, are added to the list until that value is reached. Each injector in the list is an instance of the *LoadInjector* class. The user, after calling the *read_injector()* method, can start an injection by calling the method *start_injection()* that take the first element of the list of injectors and call on it the method *inject()* of the class *LoadInjector* that starts the real injection. When the user want to stop the current injection, it has to call the method *stop_injection()* of the *InjectionManager* class that, using the *force_close()* method of the class *LoadInjector*, stops the current injection. The *LoadInjector* class, is an abstract class with two implementation: *CPUStressInjection*, that performs the injection on some single logical core or on the entire CPU, and *MemoryStressInjection* that performs the injection on the Virtual Memory. To overload the entire CPU or its logical core are used respectively the functions *load_all_core()* and *load_single_core()* of the python library *cpu_load_generator*. The load on the target core is dynamically determined based on the current load of the other cores. For this

purpose, there is an instance of the *SystemMonitor* class that monitors only the CPU current usage. The Virtual Memory, during a VM memory injection, is filled up until a certain threshold, i.e. 90%, of the virtual memory usage is reached. When this threshold is reached, the VM is discharged by a random amount to avoid a system crash. For this purpose, there is an instance of the *SystemMonitor* class that monitors only the VM current usage.
In the file main.py there is the main method in which the monitor and the injection manager are created. Then, there is a loop in which phases of normal behavior are alternated with injection phases, until all the injectors have been injected. Data points that are monitored are written in a csv file using the function *write_dict_to_csv()* in the file utilities.py. For each normal phase 120 data points are written to the csv, and for each injection 80 data points are written. The number of injections performed is 60 and then at the end of monitoring, that lasts around 2 hours and half, the final csv dataset contains 12.000 data points for 116 features. After each monitoring step, only a sleep of 0.1 second is done as the monitoring system, to collect more accurate cpu data, already takes 0.6 seconds. The output csv file of the dataset is saved in the folder output_folder.

2 – Machine Learning part on Google Colab

2.1 – Dataset manipulation

After building the dataset, I saved it on my Google Drive account. After that I load it in a Google Colab Notebook using *pandas*.
In the Colab Notebook, before more or less each module there is an explanation of what the module does.
Since there are 116 features in the dataset, I have analyzed each feature to understand which of them remove from the dataset to have a meaningful final dataset to use to train machine learning algorithms. The statistical value used for this analysis is the standard deviation. First of all I have removed from the dataset information about date and time to avoid the algorithms to learn from them. As you can see in the Notebook, I have created two different dataset configurations: the first configuration contains all the feature of the initial dataset but the ones whit a standard deviation equal to zero; in the second configuration, unlike the first, all features with low, but still positive standard deviation were also removed. I did this to see weather or not the features with low standard deviation were useful for training algorithms.
After this first phase, I have prepared the training and test data. Since features have very different ranges of values between them, we can apply the feature scaling to reduce these differences. We can use the min-max technique or the standard scaler. I prefer to use the standard scaler since the min-max restricts values between a specific range and then it is more sensitive to outliers. The standard scaler is fitted only on the training set and the transformation is done one both training and test sets. This is done to avoid that the algorithm is influenced during training by the test set, distorting the model testing phase resulting in a model with higher performance metrics.
Since I will use also a Recurrent Neural Network, I have used a different mechanism to shuffle data and to split the dataset between training, validation and test sets. In the first phase of the project, I have monitored the system by alternating the normal state to the abnormal state (where an injection is performed). Then, the dataset is organized as a chain of sequences where a sequence consists of data monitored while the system is in normal state followed by data monitored with the system under one specific attack (for a total of 60 sequences).
For RNNs, time sequences are of primary importance and then the order of data points inside the same sequence have to be kept unchanged. Then, I splitted the dataset into a list of sequences; subsequently I shuffled the list of sequences (to make them independent of each other), and finally I did the split between training, validation and test sequences. After that I applied the standard scaler on data (as before, the fit of the scaler is done only on the training set and the transformation is applied on training, validation and test sets). To prepare training, validation and test sets for the RNN I have written two support functions, namely *extract_sequences* and *split_train_val_test_scale*, both available in the Google Colab Notebook.

2.2 – Training Phase

Since the goal of this project was to develop an anomaly detector that only detects known errors/attack, I have only used supervised machine learning algorithms, as in general they are better than unsupervised ones in detecting these type of errors/attacks.
In particular I have trained:
1) Random Forest: a bagging algorithm based on a forest of Decision Trees;
2) eXtreme Gradient Boosting: a boosting algorithm whose base-learners are Decision Trees;
3) Stacking of -> Random Forest + eXtreme Gradient Boosting + ADABoost and as meta-learner Support Vector Machine Classifier (SVC);
4) Voting of -> Random Forest + eXtreme Gradient Boosting + ADABoost;

5) Recurrent Neural Network -> Gated Recurrent Unit (GRU).
I trained all algorithms using both the first and second configurations. For each algorithms I have calculated Accuracy, Matthews Correlation Coefficient (MCC), F1-Score, Recall and Precision but in this document I will not report F1-Score values that however are available in the Google Colab Notebook. The Accuracy is the ratio between correct predictions and the total number of predictions made. This metric is sensitive to unbalanced dataset and then I computed also the MCC that is more robust and it is the only other metric that considers all values of the confusion matrix. In fact, to choose the best model at the end of this section, I have used both Accuracy and in praticular the MCC. The Recall metric is useful to understand how much false negatives each algorithm generates. In fact, the higher value of the Recall is 1 that means “no false negatives prediction” and then all positive, i.e. anomalies, have been detected correctly. The Precision metric is useful to understand how much false positives each model generates. In fact, the higher value of the Precision is 1 that means “no false positives prediction” and then all positive predictions made by the model are correct. The F1-score is a metric in which Precision and Recall are balanced and, in fact, its value is given by the harmonic mean of Precision and Recall. In the Google Colab Notebook are also available the values of the confusion matrix for each model.

2.2.1 Random Forest (RF)

I have trained RF using grid search with the following sets of parameters {'n_estimators': [10, 50, 150], 'max_depth': [10, 20, None]} and using the cross validation with 10 different validation set through the class GridSearchCV of scikit-learn. For both configurations the best model is the ones obtained using as parameters {'max_depth': None, 'n_estimators': 150}.
As shown in the following table, the model resulting from the second configuration outperform the other one.

RF	Accuracy	MCC	Recall	Precision
First Config	0.9611	0.9208	0.9905	0.9144
Second Config	**0.9644**	**0.9277**	**0.9942**	**0.9190**

As shown by Precision and Recall, both models have a more FPs than FNs.

2.2.2 eXtreme Gradient Boosting (XGBoost)

I have trained XGBoost using grid search with the following sets of parameters {'n_estimators': [50, 100, 300], 'max_depth': [10, 40, 70], ‘learning_rate: [0.2]’} and using the cross validation with 10 different validation set through the classGridSearchCV of scikit-learn. For both configurations the best model is the ones obtained using as parameters {'learning_rate': 0.2, 'max_depth': 10, 'n_estimators': 300}.
As shown in the following table, the model resulting from the second configuration outperform the other one.

XGBoost	Accuracy	MCC	Recall	Precision
First Config	0.9806	0.9594	0.9912	0.9590
Second Config	**0.9836**	**0.9657**	**0.9927**	**0.9652**

As shown by Precision and Recall, both models have more FPs than FNs.

2.2.3 Stacking

I made stacking of Random Forest + eXtreme Gradient Boosting + ADABoost and as meta-learner Support Vector Machine Classifier (SVC).
Algorithms have been trained with the following configurations (neither using grid search nor cross validation but using best configuration parameters of the previous models, except for ADABoost and SVC):

- Random Forest → {'max_depth': None (default), 'n_estimators': 150}
- XGBoost → {'learning_rate': 0.2, 'max_depth': 10, 'n_estimators': 300}
- ADABoost → {‘estimator’=DecisionTreeClassifier(max_depth=7), n_estimators=50, 'learning_rate': 0.2}
- Support Vector Machine Classifier (SVC)→ {‘probability’=True}, used as meta-layer classifier

Stacking	Accuracy	MCC	Recall	Precision
First Config	0.9833	0.9650	**0.9912**	0.9658
Second Config	**0.9853**	**0.9690**	0.9905	**0.9713**

As shown by Precision and Recall, both models have more FPs than FNs and the model resulting from the second configuration outperform the other one but the latter has fewer FNs than the former, as shown by the Recall metric.

2.2.4 Voting

I made Voting of Random Forest + eXtreme Gradient Boosting + ADABoost.
Algorithms have been trained with the following configurations (neither using grid search nor cross validation but using best configuration parameters of the previous models, except for ADABoost):

- Random Forest → {'max_depth': None (default), 'n_estimators': 150}
- XGBoost → {'learning_rate': 0.2, 'max_depth': 10, 'n_estimators': 300}
- ADABoost → {‘estimator’=DecisionTreeClassifier(max_depth=7), n_estimators=50, 'learning_rate': 0.2}

Voting	Accuracy	MCC	Recall	Precision
First Config	0.9775	0.9536	**0.9971**	0.9466
Second Config	**0.9783**	**0.9552**	0.9963	**0.9492**

As shown by Precision and Recall, both models have more FPs than FNs and the model resulting from the second configuration outperform the other one but the latter has fewer FNs than the former, as shown by the Recall metric.

2.2.5 Recurrent Neural Network → Gated Recurrent Unit (GRU)

GRU is a simplified version of LSTM (Long Short-Term Memory). The first has only a short-term states and then has less dense layers inside to control its gates (forget gate and input gate). In GRU, the output of the memory cell at each time step has the same value of its short-term state.
For both configurations are used:

- batch size = 16
- epochs = 100 with early stopping with patience set to 10 and restore_best_weights set to True
- time step = 15
- starting learning rate set to 0.005 that decreases using the exponential decay

Since the number of ‘pure’ sequences is only 60, the concept of sequence is described at the end of the section “2.1 – Dataset manipulation”, I studied a strategy trying to increase this number of sequences. After the shuffle between them, they can be considered as independent sequences. Within each sequence, data points are instead dependent between each other. Then, for each sequence I consider subsequences, each composed by 15 data points and, to avoid losing the temporal connections between data points in consecutive subsequences (of the same sequence), I set an overlap to 8. This means that the last 8 data points of the previous subsequence are the same first 8 data points of the current one. The function that performs these operations is the function *create_sequences* available in the Google Colab Notebook. Furthermore, there is no overlap between the last subsequence of the previous sequence and the first subsequence of the current sequence, since as mentioned before there are no longer ties between consecutive sequences after the shuffle between them, as described at the end of the section “2.1 – Dataset manipulation”.
The structure of the RNN is the following for both configurations (here I removed any reference to each configuration, for example for the first configuration there are TIME_STEPS_FC, instead of TIME_STEPS, and FEATURES_SHAPE_FC instead of FEATURES_SHAPE):

```
model= Sequential([
   Input(shape=(TIME_STEPS, FEATURES_SHAPE)),
   GRU(units=64, return_sequences=False, activation='tanh', kernel_regularizer=l1(0.001)),
   Dropout(0.3),
   Dense(32, activation='relu'),
   Dropout(0.3),
   Dense(16, activation='relu'),
   Dropout(0.3),
   Dense(1, activation='sigmoid')
])
```

As shown in the following table, the model resulting from the second configuration outperform the other one.

RNN	Accuracy	MCC	Recall	Precision
First Config	0.9784	0.9556	0.9848	0.9630

Second Config	**0.9846**	**0.9687**	**1**	**0.9635**

As shown by Precision and Recall, both models have more FPs than FNs and the model resulting from the second configuration outperform the other one. Furthermore, the former has zero false negatives, showing a perfect Recall with value 1!
However, I think that for a better RNN I was supposed to have a larger dataset with a larger number of entire sequences.

2.3 Save Model and Scaler

All models have better performances on the second configuration, then it is useful to remove also features with positive but low standard deviation. The best accuracy and MCC are reached by the Stacking algorithm on the second configuration and I will choose it to implement the anomaly detector but there are some consideration to do before choosing the model to use.
Although this model has reached the higher accuracy and MCC, this is one of the models with the highest number of false negatives (available in the Google Colab Notebook). In general, especially in safety-critical systems (SCS), criticality arises when the algorithm fails to identify an anomaly by producing a false negative that could then lead to a catastrophic failure. In a SCS it is better to notify a possible anomaly resulting in a false positive rather than not to identify an ongoing anomaly. But, since my laptop is not a safety-critical system, I prefer to choose the model with the highest accuracy and MCC but the previous considerations are important to be highlighted. Furthermore, all my models have more false positives than false negatives. I think this is due to temporary fluctuations in the percentage of usage for each logical CPU core when, during monitoring, I opened some applications. In fact, in the latter case, the CPU utilization varied a lot, for a small amount of time, which however may look similar to what happens for anomalies.
In the end, I also saved the standard scaler (not of RNN) used to scale the data for the second configuration.

3 – Building the Anomaly Detector

The anomaly detector is represented by the class *AnomalyDetector* in the file AnomalyDetector.py. An object of this class during its creation needs to receive as parameters a trained *StackingClassifier* and a trained *StandardScaler.* This class uses an instance of the *SystemMonitor* class to monitor the usage of resources in the system. The user, when using this class, has to call the method *start_anomaly_detection()* that, in another thread, calls the function *detect_anomalies()* to start a monitoring loop in which, at each iteration, data from the system are gathered using the *monitor()* method of the class *SystemMonitor.* Subsequently, from gathered data are removed all features that are not used by the ML model and, the resulting list is converted in a dataFrame whose data will be scaled using the scaler before entering the ML model. The latter will output its prediction plus the probabilities it calculated to classify the point. This function, *detect_anomalies()*, holds a counter that is incremented by 1 after an anomaly has been detected and decremented by 1 in the opposite case. The class *AnomalyDetector* has a constant variable *TRESHOLD_TO_RESET_FLAG* that is an integer number that represents the number of consecutive times that the anomaly detector says NORMAL after which the *SeverityLevel* is set to *LEVEL_5* (the lowest one), e.g. if after several consecutive anomalies the system is said to be normal for *TRESHOLD_TO_RESET_FLAG* consecutive times, the system is considered “out of danger”. A *SeverityLevel* (enum class in the file SeverityLevel.py) is assigned to the system. There are five of these levels, that range from *LEVEL_5* (the lower) to *LEVEL_1* (the higher and more critical) and the corresponding system status printed by the Anomaly Detector are *[NORMAL, UNDER OBSERVATION, CAUTION, SEVERE, CRITICAL]*. The system status pass from one level to another based on the counter for the number of consecutive anomalies detected (counter explained before that is holds in the function *detect_anomalies()*). After each monitoring/prediction the *SeverityLevel* is updated based on the value of that counter and a message is printed on the console/terminal to show the model’s prediction plus the current system status that is represented by the severity levels.
The class *AnomalyDetector* also has a method *log_system_info* that logs, in a CSV format, info about the system status and predictions made by the classifier into two log files whose names and path are represented by the other three constants, namely *OUT_FOLDER* (in this case with name log), *LOG_DATAPOINT_AND_PREDICTION_FILENAME* and *LOG_PREDICTIONS_AND_SEVERITY_LEVEL*.
The anomaly detector is created and started in the file main_anomaly_detector.py and to stop the anomaly detection is sufficient to click Ctrl+C to generate a *KeyboardInterrupt* thanks to which the method *stop()* of the class *AnomalyDetector* is called to stop the detection process.

4 - Information about my laptop (system used in this experiment)

The system under monitoring was my laptop that runs ubuntu 24.04 noble, with 16 GB of RAM and 1 GB intel iris xe integrated GPU. The CPU, an Intel® Core™ i7-1165G7, has 4 physical cores and 8 logical cores.

5- How to run the code

Instructions are available in the file *README.txt*.

6 – The following picture contains the folder structure

```
DCML-CPS_Project/
|
|---log/ # contains log files generated during the execution of the file main_anomaly_detector.py
|      |--- datapoint_with_predictions.log
|      |--- predictions_with_severity_level.log
|
|---output_folder/ # contains the CSV generated during the execution of the file main.py
|            |--- DCML_Project_dataset.csv # file CSV of the dataset
|
|---saved_models/
|            |--- best_model_stacking.pkl # file of the model with best Accuracy and MCC (Stacking Classifier)
|            |--- scaler.pkl # file of the standard scaler to use at run time (fitted on the training set of the model)
|
|---src/
|      |---monitoring/
|      |          |--- AnomalyDetector.py # class of the anomaly detector
|      |          |--- InjectionManager.py # class to handle injection in the system
|      |          |--- LoadInjector.py # class to load/start/stop injection
|      |          |--- SystemMonitor.py # class to monitor the usage of system's resources
|      |
|      |---utils/
|      |        |--- SeverityLevel.py # enum to represents the severity level of an ongoing anomaly
|      |        |--- SystemState.py # enum to represents the state of the system
|      |        |--- utilities.py # contains utility functions
|      |
|      |--- DCML_Colab_Project_Agatensi.ipynb # Notebook Google Colab for ML part
|      |--- debug_injectors.json # json used if debug is enabled during monitoring/injection
|      |--- injectors_json.json # json used if debug is disabled
|      |--- main_anomaly_detector.py # main to be executed to run the Anomaly Detector
|      |--- main.py # main to be executed to run the monitoring/injection to build the dataset
|
|--- test_Anomaly_Detector/
|            |--- stress_cycles.sh # shell's script to stress the system to test the final Anomaly Detector
|
|--- ProjectReport.pdf # Report of the project
|--- README.txt # contains instructions to run the code
|--- requirements.txt # python requirements to run all main*.py files
```