UNIVERSITÀ
DEGLI STUDI
FIRENZE

A custom anomaly detector for laptops or workstations

Final project for data collection and machine learning for critical cyber-physical systems

Date: 03.01.2024

Student: Giamberini Giulia

Matriculation number: 7149574

Email: giulia.giamberini@edu.unifi.it

A.A.: 2023/2024

SOMMARIO

1 Introduction 3
2 Monitor design and development 3
2.1 Monitoring phase 3
2.2 Training, testing and evaluation of ML algorithms phase 4
2.3 Real-time monitoring phase 4
3 Selection of monitored indicators 4
4 Generating the training set 6
5 Training, testing and algorithm comparison 7
6 Final integration of monitor and detector 10

1 INTRODUCTION

The primary goal of this project was to create a monitoring system for personal laptops capable of automatically detecting anomalies, specifically focusing on CPU, memory, and disk-related irregularities. After the development of the monitor itself, the dataset was created and used for training some supervised and unsupervised machine learning algorithms. Through an evaluation, including the assessment of accuracy for each algorithm, the most effective one was identified and subsequently employed for real-time monitoring.

2 MONITOR DESIGN AND DEVELOPMENT

The monitor was developed using the Python programming language, utilizing key libraries such as **Pandas**, **scikit-learn** (sklearn), **pickle**, and **psutil**. It comprises a main script that guides the user through the process of creating the dataset, training supervised/unsupervised algorithms, evaluating them, and saving the best-performing model based on accuracy. Additionally, it facilitates live monitoring of the system to capture anomalies in real time.

The structure of the project is:

```
monitor_project/
├── anomaly_detector/
│   ├── anomaly_detector_log.txt
│   └── anomaly_detector_model.pkl
├── csvFolder/
│   ├── dataset.csv
│   ├── injection_timestamps.csv
│   └── monitorResult.csv
├── csvGenerator.py
├── injector.py
├── instruction.txt
├── labeling.py
├── liveMonitor.py
├── main.py
├── monitor.py
├── reducingCSV.py
├── stress_cpu.py
├── stress_disk.py
├── stress_memory.py
├── supervisedML.py
├── unsupervisedML.py
└── util.py
```

2.1 MONITORING PHASE

The core logic for the **monitoring phase** is implemented in the **monitor.py file**. The *monitor_all()* function is executed based on the user-defined number of observations to capture. During each execution, CPU, memory, and disk information are collected and transformed to fit the flat format of the CSV. Tuples and dictionaries are converted into lists of elements. Each observation is then saved in the **monitorResult.csv** file (utilizing **csvGenerator.py**). Progression during each iteration is visualized through a progress bar (implemented in **progressBar.py**). Once all iterations are completed, the file is labeled. This labeling involves assigning each row a label of 'normal' or 'anomaly' by cross-referencing it with the **injection_timestamp.csv**, which stores the timestamp of the start and finish for each injection.

During the monitoring phase, a series of injections (CPU, memory, and disk) are randomly executed, randomizing both the order and duration of execution. For each injection, upon completion, the injection number, start and end times, and type are stored in the injection_timestamp.csv file. This information will later be used for **cross-checking** with the monitored data to label them as anomalous or normal. If a record is detected during an injection phase, it will be **recognized by comparing the timestamp** and ensuring that it falls within a pair in the injection file.

The decision to split the **stress** into multiple files (**stress_cpu.py, stress_disk.py and stress_memeory.py**) was made due to the need to execute them during the real-time monitoring test phase as well.

All the CSV files are generated using the **csvGenerator.py** script, and they are stored in the csvFolder. If the folder is not present, it is created. During the first writing phase, the column names are also stored. It's important to note that executing two monitoring phases sequentially results in a single file where the observations are appended at the end. If this is not the desired outcome, it is recommended to either remove the file or the folder. This way, the code will generate a new CSV file.

2.2 TRAINING, TESTING AND EVALUATION OF ML ALGORITHMS PHASE

After creating the dataset, the phase of **training, testing**, and **evaluating machine learning algorithms** can be initiated. Columns with lower variation within the entire set are removed from the dataset, assuming that lower variation indicates a lower likelihood that the monitored data is influential for the algorithm's decision-making. The dataset is divided into 75% for the training phase and 25% for the testing phase. The choice of this split was determined after several attempts with different division ratios (20%, 25%, 30%, 40%, 50%). For each execution of **unsupervisedML.py**/**supervisedML.py**, the performance of each algorithm is stored and evaluated based on accuracy. The results are also presented to the user to understand which algorithm has been selected as 'optimal.'

The **util.py** file is used only for functions utilized across different files and to minimize code redundancy.

There is also a script called **reducingCSV.py** used to create a smaller dataset from the one generated by the monitor. This ensures that the results for each machine learning algorithm remain consistent, whether the dataset comprises 500 rows or 750 rows. In the test conducted on my personal laptop, I opted to capture 1000 observations to guarantee sufficient data for training and testing these algorithms.

2.3 REAL-TIME MONITORING PHASE

After completing all those phases, the user can execute the **liveMonitor.py** script for **real-time monitoring**. In this scenario, the model selected during the algorithm evaluation phase is loaded and initiated. The script prints the time of each observation to the console and the corresponding log file, indicating whether it is recognized by the model as a normal state or an anomaly.

3 SELECTION OF MONITORED INDICATORS

The selection of indicators was driven by the need to ascertain in advance whether the device under consideration is subject to excessive stress, particularly related to CPU, disk, and memory.

CPU Metrics and Load Average Metrics:

- user, system, idle, interrupt: these metrics help monitor the CPU utilization and identify how much time the CPU spends on different tasks (user processes, system processes, idle time, and interrupts)
- load_avg_0, load_avg_1, load_avg_2: load averages represent the average number of processes in the system's run queue over different time intervals (1 minute, 5 minutes, and 15 minutes). They indicate system activity and whether the system is under heavy load.

The CPU metrics are used to capture a snapshot of CPU usage, aiding the user in identifying abnormal CPU behaviors or spikes. Proactively identifying issues may help prevent system slowdowns, crashes, or performance-related problems. When combined with load_Avg, it can detect high CPU usage, which may be a symptom of various issues, such as inefficient code, malware, or hardware problems, potentially leading to overheating.

Memory Metrics, Disk Metrics and Disk I/O Metrics:

- memory_used, memory_free, memory_percent: these metrics show the current memory usage, free memory, and the percentage of memory used.
- disk_used_disk, disk_free_disk, disk_percent_disk: these metrics indicate the disk space usage, free space, and the percentage of disk space used on a specific disk.
- disk_io_read_count_disk_io, disk_io_write_count_disk_io: these metrics represent the number of read and write operations on the disk.
- disk_io_read_bytes_disk_io, disk_io_write_bytes_disk_io: these metrics show the amount of data read from and written to the disk in bytes.
- disk_io_read_time_disk_io, disk_io_write_time_disk_io: these metrics represent the time spent on disk read and write operations.

The disk and memory metrics are employed to track write and read operations on the disk and memory. Prolonged high disk activity can lead to increased heat generation. These monitors assist in efficiently allocating resources. For instance, if a specific disk is approaching full capacity, redistributing or archiving data to other storage devices might be necessary. Sustained high I/O activity poses an elevated risk of data corruption or errors.

4 GENERATING THE TRAINING SET

For generating the training set it was used the **psutil** library as mentioned in the 'Monitor design and development' chapter.

In the developed system, the generation of the training set is composed of a multi-step process, using a combination of data **monitoring** and anomaly **injection**. The monitoring script, 'monitor.py,' systematically captures system metrics such as CPU, disk, and memory usage, creating a baseline representation of normal system behavior.

In the meanwhile, the **injector.py** script introduces anomalies into the system by simulating stress scenarios (CPU, memory, and disk stress). It shuffles a list of possible stressors and injects them, waiting for the ending of the previous one. The *stress()* function is invoked and, based on the type of stress that the injection needs, it selects the corresponding function. A pool of processes is generated, with the number of processes for CPU stress being half of what is required for disk and memory stress. This decision was made because when the functions stress_memory or stress_disk were individually executed, their effects on the system were too minimal. Each process is mapped asynchronously and, after a duration_ms time, the pool is terminated. The record for the injection is saved in the corresponding CSV, and then the function returns.

label
normal
normal
anomaly
anomaly
anomaly
anomaly

This approach guarantees a diverse training set that comprises both regular and anomalous states. The **labeling.py** script annotates the dataset with labels distinguishing between normal and anomalous instances.

The stress of the **CPU** is based on **intense mathematical calculations** in an infinite loop. The stress of **memory** is based on continuously **allocating** a large amount of memory, and saving all the data in a list. The stress of the **Disk**, instead, is based on **writing and reading** continuously on the disk, creating a temporary file, writing and reading over it, and then closing the file.

All indicators are saved. The function **unchangedColumns**() is called before training to filter columns that change minimally. Using the dataset, the result list of columns is: *['cpu_count', 'interrupt', 'dpc', 'memory_sin', 'memory_sout', 'disk_total_disk', 'disk_percent_disk', 'disk_io_read_time_disk_io']*

Stressing results using 'activity management'

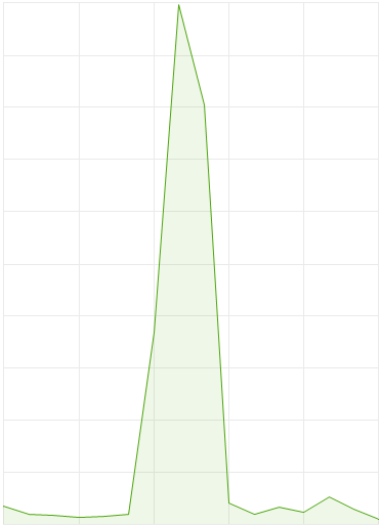

Figure 1: stress of the disk

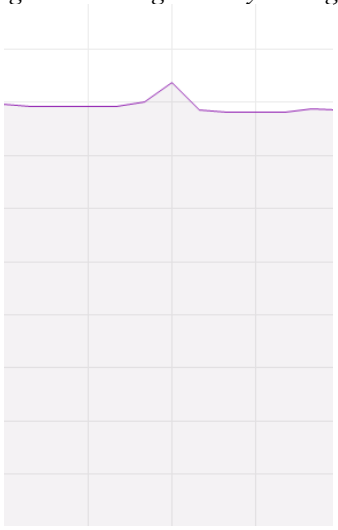

Figure 2: stress of the memory

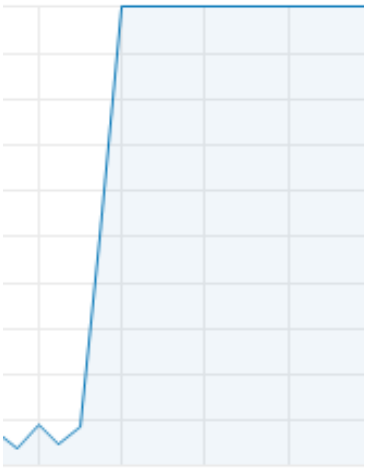

Figure 3: stress of the cpu

5 TRAINING, TESTING AND ALGORITHM COMPARISON

The training of the algorithms was done as follows: a selection of classifiers was saved in a list and iteratively used with the dataset to train them to recognize anomalies or normal data. The dataset was split into a training set and a test set, as is visible in the tables. Each classifier was fitted. The supervised ones needed the observations and the labels for training, while the unsupervised ones needed only the observations. After the training, the prediction was made over the remaining part of the dataset (the test set). After that, for each algorithm, the accuracy of the results was evaluated. Based on this indicator, the best algorithm was chosen and saved as a model to be loaded before real-time monitoring.

SUPERVISED ALGORITHMS		**Test set dimension**				
		20%	**25%**	**30%**	**40%**	**50%**
VotingClassifier (lda,nb,dt)	Accuracy	0.965	0.972	0.976	0.98	0.82
	Training time	15	24	0	8	11
	Testing time	5	6	0	3	4
VotingClassifier (lda,nb,rf(20))	Accuracy	0.96	0.976	0.973	0.9775	0.774
	Training time	56	58	47	46	49
	Testing time	4	4	14	10	8
StackingClassifier (lda,nb,dt).rf(10)	Accuracy	0.97	0.968	0.973	0.98	0.5
	Training time	84	93	66	91	90
	Testing time	3	3	15	3	4
StackingClassifier (lda,nb,dt).kn(11)	Accuracy	0.97	0.972	**0.98**	0.9825	0.906
	Training time	70	72	**80**	66	91
	Testing time	9	9	**0**	16	30
DecisionTreeClassifier	Accuracy	0.965	0.976	**0.98**	**0.985**	**0.988**
	Training time	5	5	**0**	**4**	**8**
	Testing time	1	0	**0**	**1**	**2**
GaussianNB	Accuracy	0.725	0.78	0.723	0.755	0.646
	Training time	2	1	0	1	2
	Testing time	0	1	0	1	2
LinearDiscriminantAnalysis	Accuracy	0.96	0.968	0.973	0.9775	0.786
	Training time	5	5	16	2	4
	Testing time	0	0	0	0	1
KNeighborsClassifier(11)	Accuracy	0.68	0.78	0.7	0.7625	0.744
	Training time	1	1	0	1	2
	Testing time	118	153	126	133	197
KNeighborsClassifier(25)	Accuracy	0.715	0.772	0.7	0.71	0.772
	Training time	2	2	0	2	2
	Testing time	9	16	16	18	26
RandomForestClassifier(3)	Accuracy	**0.975**	0.972	0.7	0.975	0.978
	Training time	**10**	8	15	6	8
	Testing time	**1**	1	0	1	2
RandomForestClassifier(10)	Accuracy	0.97	**0.98**	**0.98**	0.98	0.978
	Training time	33	**25**	**32**	20	26
	Testing time	1	**1**	**0**	1	2
RandomForestClassifier(20)	Accuracy	0.965	0.976	0.97	0.98	0.98
	Training time	57	44	68	41	49
	Testing time	2	2	2	3	3

Note: accuracy is expressed in a range of [0;1] and the training/testing times are expressed in milliseconds

Analyzing the accuracy, training times, and testing times for supervised machine learning algorithms, it is possible to observe that as the size of the training set varies, especially when reducing its size, the algorithm with the highest accuracy is no longer RandomForestClassifier but DecisionTree.

In our case study, testing the model obtained with DecisionTree on live monitoring shows that it detects anomalies much more sensitively. During normal device operation, even opening an application results in an anomaly.

If we compare those algorithms using not only accuracy as a comparing field but also **precision** and **recall**, we can see that those are not so different (not what was expected). In this case, if we try to order them based on accuracy, recall, precision, training_time and testing_time we can create the following report:

Classifier	**Accuracy**	**Recall**	**Precision**	**Training time**	**Testing time**
RFC (n=20)	0.996	1.0	0.9851	68	0
RFC (n=3)	0.992	1.0	0.9706	10	0
RFC (n=10)	0.992	1.0	0.9706	31	16
DTC	0.992	0.9848	0.9848	0	0
SC (LDA, NB, DT; FE=KNN)	0.988	0.9545	1.0	94	25
VC (LDA, NB, RFC 20)	0.984	0.9697	0.9697	48	0
LDA	0.984	0.9545	0.9844	0	0
VC (LDA, NB, DT)	0.984	0.9545	0.9844	35	0
SC (LDA, NB, DT; FE=RFC 10)	0.984	0.9545	0.9844	112	0
GaussianNB()	0.776	0.5	0.5893	0	0
KNN (n=25)	0.764	0.4545	0.5660	0	31
KNN (n=11)	0.748	0.0455	1.0	0	16

The decision to prioritize accuracy before and recall after is justified by the aim to ensure the live monitor **detects all anomalies** without compromising effectiveness.

While RandomForest consistently outperforms DecisionTree in recall and precision, against the hypothesis made before, their **behaviors** seem **similar** in the live monitor, suggesting potential variations during the training and testing phases. The irregular behavior observed during anomaly execution on the laptop could be likely influenced by external variables.

Despite that, across various test set dimensions, the report consistently supports **RandomForest** as the optimal choice for the live monitoring system, instilling confidence in the selected model. Regarding KNN(n=11), it displays a trade-off with low recall and high precision, indicating a tendency to label normal instances as anomalies. Adjusting the number of neighbors may enhance the balance between precision and recall.

In the case of anomaly detection with RandomForest, the preferred choice is **RandomForestClassifier(20).** Anomalies are detected only when the user initiates injections using specific functions, ensuring a controlled and user-friendly monitoring system.

UNSUPERVISED ALGORITHMS		Test set dimension				
		20%	**25%**	**30%**	**40%**	**50%**
HBOS(alpha = 0.01, cont = 0.1, n_bins = 20)	Accuracy	0.52	0.572	0.613	0.6	0.36
	Training time	2030	1876	2064	1948	2011
	Testing time	1	0	1	1	0
HBOS(alpha = 0.5, cont = 0.5, n_bins = 5)	Accuracy	0.455	0.512	0.58	0.5825	0.64
	Training time	3	2	5	2	3
	Testing time	1	1	1	0	1
HBOS(alpha = 0.01, cont = 0.1, n_bins = 45)	Accuracy	0.54	0.528	0.56	0.4	0.35
	Training time	4	4	3	3	3
	Testing time	0	0	1	0	0
HBOS(alpha = 0.1, cont = 0.3, n_bins = 100)	Accuracy	0.405	0.448	0.44	0.3	0.32
	Training time	3	3	3	2	2
	Testing time	0	1	1	1	1
ABOD(cont = 0.5)	Accuracy	0.325	0.292	0.303	0.3	0.3
	Training time	1196	1088	1088	1137	1124
	Testing time	21	22	29	52	37
ABOD(cont = 0.1)	Accuracy	0.34	0.408	0.3	0.3	0.338
	Training time	121	99	66	89	45
	Testing time	18	27	33	43	56
ABOD(cont = 0.001)	Accuracy	0.45	0.548	0.433	0.295	0.312
	Training time	108	118	105	69	53
	Testing time	17	40	31	48	49
ABOD(cont = 0.000000001)	Accuracy	0.42	0.58	0.406	0.285	0.356
	Training time	89	110	91	147	57
	Testing time	18	24	50	30	58
COPOD(cont = 0.1)	Accuracy	**0.71**	**0.728**	**0.706**	**0.7075**	**0.72**
	Training time	**120**	**125**	**162**	**110**	**115**
	Testing time	**3**	**3**	**3**	**2**	**2**

For the unsupervised algorithms, the logic behind the algorithm parameters is as follows: in the HBOS classifier, a lower alpha value implies reduced sensitivity to values that deviate more from the norm. Decreasing the contamination value indicates a smaller proportion of outliers in the dataset. For the ABOD classifier, a lower cont value enhances sensitivity, leading to more points being classified as outliers. The same principle applies to COPOD. While various contamination values were tested during the script's configuration phase, the results were largely similar. Consequently, the default contamination value was retained.

Regarding unsupervised algorithms, besides selecting different parameters for the three classifiers, we also introduced a different size for the training set. However, it was observed that the algorithm with the best performance (highest accuracy) is COPOD.

Despite this, the accuracy detected by the supervised algorithms is significantly higher, and therefore, preference has been given to their use.

Using the same dataset.csv file, reducing its size to 50% (500 rows) or 75% (750 rows), the algorithm with the highest accuracy, considering all the specified percentages of the test set (20%, 25%, 30%, 40%, 50%), continues to be the RandomForestClassifier. The accuracy remains very similar in the cases of both 3 and 10 estimators.

Note that during every run of either the 'supervisedML.py' or 'unsupervisedML.py' script, the optimal algorithm is systematically identified and preserved, leading to the overwrite of the previous selection. This decision stems from the observation that running the same script repeatedly yields only marginal variations in results.

6 FINAL INTEGRATION OF MONITOR AND DETECTOR

After saving the model, the user can run it to detect real-time anomalies. The liveMonitor.py script will load the model and print it out, in the console and also in the log file, the time of the observation and if it was considered an anomaly by the model.

```
2024-01-06 18:17:44,895 - 2024-01-06 18:17:44 - Detected: normal
2024-01-06 18:17:45,917 - 2024-01-06 18:17:45 - Detected: anomaly
2024-01-06 18:17:46,936 - 2024-01-06 18:17:46 - Detected: normal
2024-01-06 18:17:47,967 - 2024-01-06 18:17:47 - Detected: normal
2024-01-06 18:17:48,993 - 2024-01-06 18:17:48 - Detected: normal
2024-01-06 18:17:50,003 - 2024-01-06 18:17:50 - Detected: normal
2024-01-06 18:17:51,031 - 2024-01-06 18:17:51 - Detected: normal
2024-01-06 18:17:52,467 - 2024-01-06 18:17:52 - Detected: anomaly
2024-01-06 18:17:54,116 - 2024-01-06 18:17:54 - Detected: anomaly
2024-01-06 18:17:55,134 - 2024-01-06 18:17:55 - Detected: normal
2024-01-06 18:17:56,151 - 2024-01-06 18:17:56 - Detected: normal
```