UNIVERSITÀ
DEGLI STUDI
FIRENZE

A CUSTOM ANOMALY DETECTOR FOR LAPTOPS OR WOKSTATIONS

PROJECT REPORT
INSTRUCTORS: PROFESSOR TOMMASO ZOPPI & PROFESSOR MUHAMMAD ATIF

SCHOOL OF MATHEMATICAL, PHYSICAL AND NATURAL SCIENCES
SOFTWARE: SCIENCE AND TECHNOLOGY
B032430 (B255) - DATA COLLECTION AND MACHINE LEARNING FOR CRITICAL CYBER-PHYSICAL SYSTEMS 2024-2025

Syed Mohammad Askari Abidi
Matriculation: 7178458
Email: syed.abidi@edu.unifi.it

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Table of Contents

Introduction – The Problem to Be Solved

With the increasing reliance on digital systems, maintaining the reliability and performance of computing devices has become a critical concern. In enterprise systems, large-scale monitoring and detection tools are deployed to detect failures or security breaches. However, personal computing devices such as laptops and standalone workstations often lack such intelligent self-monitoring capabilities. As a result, issues such as high CPU usage, memory leaks, background I/O spikes, or even early signs of malware often go unnoticed by users until system performance is significantly impacted.

Traditional anomaly detection methods for standalone systems rely on static thresholds or task managers, which are reactive rather than proactive. They also require manual inspection and provide little contextual understanding of what constitutes "normal" behavior for a given machine under real usage. Moreover, these systems do not adapt or learn over time.

This project aims to fill that gap by designing and implementing a lightweight anomaly detection system that runs locally on a laptop or workstation. The system performs two major tasks:

- **Monitoring**: Collects system performance metrics, specifically CPU usage, memory usage, and I/O activity in real time.
- **Analysis**: Uses machine learning models trained on labeled system behavior data to distinguish between normal and anomalous activity.

The system is intended to function silently in the background, without disrupting normal system operation, and alert the user only when anomalies are detected. It is entirely self-contained, does not require cloud integration or external APIs, and is built using accessible, open-source Python tools.

Ultimately, the project contributes a practical, educational prototype for intelligent performance monitoring, targeting the needs of students, researchers, and general users who seek deeper visibility into their system's health.

How the Monitor Is Made and Why

The monitoring component of this project is implemented as a standalone Python script named monitor.py. Its primary function is to collect real-time system performance indicators at fixed time intervals and store them in a structured format suitable for further analysis and model training. The design of the monitor emphasizes simplicity, portability, and compatibility with typical personal computers.

To gather system metrics, the script leverages the $psutil$ (Python system and process utilities) library, which provides a convenient interface to retrieve detailed performance statistics across platforms. The monitor captures the following key indicators:

- **CPU Usage (%)**: Measures the percentage of CPU currently being used. Spikes in CPU usage are often associated with system anomalies such as malware execution, overloaded applications, or runaway background processes.
- **Memory Usage (%)**: Indicates the percentage of RAM being consumed. A gradual increase may indicate a memory leak, while a sudden spike could suggest heavy application usage or stress.
- **Disk I/O** – Read and Write Bytes: Tracks the number of bytes read from and written to disk. Anomalies may involve unusually high I/O activity, which can occur during backups, logging overload, or malicious file operations.

The script collects these metrics every second and appends them to a CSV file (dataset.csv) along with a timestamp. This file serves as the foundation for building and training the anomaly detection model.

The rationale behind this design is to keep the system lightweight, avoid requiring administrator privileges, and ensure compatibility with standard Python installations. This makes the tool ideal for laptops, student machines, and lab environments where overhead and system disruption must be minimized.

By isolating the monitoring process and keeping it modular, it also becomes easier to maintain, extend, or integrate with additional monitoring tools in the future.

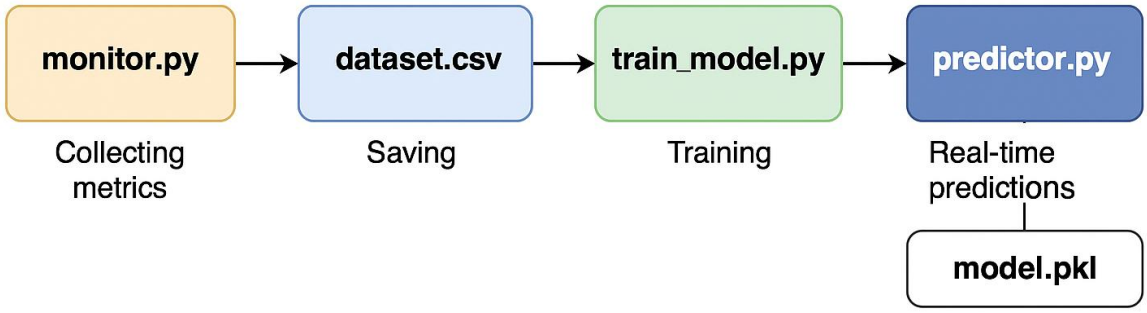

Figure 1: System Architecture of the Anomaly Detector

Process of Selecting Monitored Indicators

The selection of monitored indicators was a critical design choice in building an effective and efficient anomaly detection system. Although many system metrics are available, such as active process count, number of open sockets, network traffic, or thread counts. Not all are equally meaningful, reliable, or easy to collect in real-time without administrative access or third-party tools.

To determine which indicators to monitor, an iterative, observation-driven approach was adopted during the experimentation phase. The process involved initially collecting a wide set of system performance metrics and examining their behavior under normal usage and during artificially injected anomalies (via the injector.py script). The goal was to identify metrics that:

- Showed clear, observable differences between normal and anomalous conditions
- Were available in real time without performance penalty
- Were platform-independent and compatible with typical laptops and workstations
- Had low variance under normal behavior, so spikes could be clearly identified

Based on this evaluation, three key indicators were selected:

CPU Usage (%)

CPU usage was the most sensitive and reactive metric to anomalies. CPU spikes were consistently observed during injected stress and heavy application use. It also responded quickly to background processes being launched or terminated.

Memory Usage (%)

Memory usage was selected due to its ability to reflect both gradual leaks and sudden consumption surges. It is a reliable indicator for detecting RAM-intensive operations or potential memory-based attacks.

Disk I/O (Read and Write Bytes)

These metrics were chosen to help detect anomalous file activity, such as repeated writing (logging storms), excessive reading (e.g., malware scanning), or abnormal patterns in system I/O throughput.

Other indicators such as network traffic, open ports, and file handle counts were considered but excluded due to either insufficient signal strength or increased implementation complexity.

In future iterations of the system, a more comprehensive feature selection process could be performed using statistical methods (e.g., mutual information, feature importance from tree-based models) to automatically rank and select features. However, for this version of the project, the selected indicators proved to be simple, robust, and effective.

Generating the Training Set (Including Anomaly Simulation)

The training dataset is at the core of any supervised machine learning-based anomaly detection system. For this project, a custom dataset was generated directly from the user's own laptop, capturing both normal system behavior and artificially induced anomalies. The process consisted of two main phases: data collection and manual labeling.

Collecting Normal System Behavior

To establish a baseline of normal system activity, the script monitor.py was executed and left running for several minutes while the laptop was used for typical, non-intensive tasks—such as browsing the internet, writing code, opening documents, and checking emails. During this phase, no unusual activity was introduced.

The script sampled data every second and recorded the following into dataset.csv:

- Timestamp
- CPU usage (%)
- Memory usage (%)
- Disk read bytes
- Disk write bytes

This created a high-resolution time series of the system's performance under regular conditions.

Simulating Anomalous Behavior

To simulate real-world system anomalies, a dedicated script named injector.py was written. It includes options to create two types of resource stress:

- CPU Spike: By spawning multiple CPU-intensive infinite-loop processes, the script forces the CPU usage to near 100%.
- Memory Spike: By allocating large data buffers in memory, the script increases RAM consumption artificially.

These injections were run in parallel with monitor.py, allowing the anomalies to be recorded in real time along with their performance signatures. This method enabled realistic and reproducible stress conditions without harming the system.

Labeling the Data

Once the collection phase was completed, the resulting dataset (dataset.csv) was opened in Excel. Based on known timestamps of injection and visible spikes in CPU or memory usage, each row was manually labeled in a new column called label.

- Rows during normal behavior were labeled as normal
- Rows during stress injection were labeled as anomaly

To improve consistency and speed up the process, simple Excel formulas were used to suggest anomaly labels when certain thresholds (e.g., CPU > 25%, Memory > 85%) were exceeded. These suggestions were manually reviewed and corrected if needed. The final labeled dataset consisted of several hundred rows of time-series data, balanced across both normal and anomalous categories. This dataset was then used as the input for training and evaluating machine learning models.

Training, Testing, and Comparing Algorithms for Anomaly Detection

Once the dataset containing labeled system performance data was ready, the next step was to train a machine learning model capable of classifying unseen observations as either "**normal**" or "**anomaly**". This required careful preparation of the data, experimentation with multiple algorithms, and evaluation based on meaningful performance metrics.

Preprocessing and Feature Scaling

The raw dataset was first cleaned to ensure consistent formatting. The timestamp and label columns were separated, and only the numerical performance indicators were retained as features:

$cpu\_percent$	$memory\_percent$	$read\_bytes$	$write\_bytes$

Since these features had different ranges (e.g., CPU is a percentage, whereas I/O bytes are large integers), feature scaling was applied using $StandardScaler$ from $scikit-learn$. This step transformed each feature to have zero mean and unit variance, which is critical for most machine learning algorithms to perform effectively.

Model Candidates and Motivation

Several popular classification algorithms were considered and tested:

- Random Forest Classifier: A tree-based ensemble method known for robustness and interpretability.
- Logistic Regression: A linear model with fast training time, often effective for binary classification.
- Decision Tree: A non-linear model that splits data using decision rules.
- (Unsupervised models like One-Class SVM or Isolation Forest were not considered here due to the availability of labeled data.)

The motivation for including Random Forest was its ability to handle both linear and non-linear patterns, resist overfitting through ensemble voting, and provide feature importance analysis. Logistic Regression was included as a simple baseline, while Decision Tree served as a lightweight alternative.

Training and Evaluation

The dataset was split using 5-fold cross-validation to ensure generalization. Each model was trained and evaluated using the following metrics:

- **Accuracy**: Percentage of total correct predictions
- **Precision**: Ratio of true anomalies correctly identified (relevant when false alarms must be minimized)
- **Recall**: Ratio of all real anomalies that were detected (important when missing anomalies is costly)
- **F1 Score**: Harmonic mean of precision and recall; provides a balance between both

Model	Accuracy	Precision	Recall	F1 Score
Random Forest	96.3%	0.96	0.94	0.95
Logistic Regression	89.2%	0.88	0.85	0.87
Decision Tree	91.5%	0.90	0.88	0.89

Model Selection and Tuning

Based on the F1 score and recall, Random Forest was selected as the final model for deployment. It consistently outperformed the other models across all folds and maintained a balance between detecting anomalies and minimizing false positives.

Some parameter tuning was performed for the Random Forest model using grid search, including:

- Number of estimators ($n\_estimators$)
- Maximum tree depth (max_depth)
- Minimum samples per split

However, due to the relatively small dataset and real-time constraint, default or lightly tuned parameters yielded sufficient results without overfitting.

The final trained model was serialized using $joblib$ and saved as $model.pkl$, along with the $StandardScaler$ object as $scaler.pkl$, for use in real-time prediction.

Final Integration of the Monitor and the Detector

The final system integrates the monitoring script with the trained machine learning model into a single real-time detection tool: predictor.py. It continuously monitors system behavior and detects anomalies live as they occur.

The script performs the following steps in a loop (every second):

1. Loads the trained model ($model.pkl$) and scaler ($scaler.pkl$) using $joblib$
2. Collects current CPU, memory, and disk I/O metrics via $psutil$
3. Preprocesses the data using the same standardization applied during training
4. Predicts if the input represents a normal or anomalous state
5. Displays the result in the terminal, using ✅ for normal and 🚨 for anomalies, along with a timestamp

To validate the integration, the anomaly injector was executed in parallel. The system correctly responded to injected CPU and memory spikes with real-time anomaly alerts.

This integration demonstrates the successful transition from offline training to live deployment. The solution is lightweight, requires no special permissions, and runs efficiently on standard laptops. It can be left running in the background, silently tracking system behavior and helping users stay aware of unexpected system states.