Alma Mater Studiorum - University of Bologna

Course of Fondamenti di Intelligenza Artificiale M
Academic Year 2022-2023

Riccardo Evangelisti

Anomaly Detection on High-Performance Computing real data using Autoencoders

Abstract

The study investigates the application of Autoencoders in Anomaly Detection using actual High-Performance Computing (HPC) data. It centres on data gathered from the Marconi100 supercomputer across two and a half years through the ExaMon framework. The goal is to develop a model that can distinguish between the normal and anomalous states of a node. Furthermore, the research examines whether the model can identify an anomaly before the manual intervention by system administrators.

The code employed is available in this [1] repository, while here [5] can be downloaded the ExaMon data.

1 Data collection and ExaMon

The data on which this research was developed come from a very large dataset collected over two and a half years on the Marconi100[2] supercomputer, located at CINECA (Bologna, Italy). The data were collected employing a holistic framework for monitoring and maintenance of HPC facilities, called ExaMon[3], from 2020-03 to 2022-09. There are a few hundred metrics collected on each computing node, plus dozens covering racks and rooms. Therefore, the data provided covers the entirety of the system, such as core loads, temperatures, frequencies, memory write/read operations, CPU power consumption, fan speed, GPU usage details, job details, etc. (for each different computing node). The complete description of the data collection campaign can be found here [4].

2 Nagios signal

Among the ExaMon plugins and metrics, the Nagios diagnostic tool monitors and visualises critical IT infrastructure components. It sends alerts to the management staff, which are then handled by automated scripts that mark the node status depending on the severity of the alert ("Normal", "Warning" and "Critical"), waiting for a system administrator to intervene[3]. If the alert is real, the involved computing nodes are removed from production ("**drained**") and this action is then registered in Nagios by the "*DOWN+DRAIN*" state of the node, represented in the dataset by the feature called nagiosdrained. When the issue is solved, the node is reactivated (and the Nagios node state is restored). Basically, the nagiosdrained is a boolean feature that has a value of 0 if the node is up and running, and a value of 1 if in the "*DOWN+DRAIN*" state.

Since the nagiosdrained is a signal that is manually changed by system administrators, it precisely gives the information whether a node had a failure or not.

This procedure implies that there is an implicit delay between the insurgence of an anomalous situation and the corresponding label on the Nagios log. Therefore, this research not only wants to develop a model that detects correctly the anomalous states from the normal ones but also wants to see if it's possible to detect the anomalous states when the node failure actually happens, training the model on all the provided ExaMon data.

In Figure 1 are identified six different phases[3]:

- (1) corresponds to the node in a normal state;
- (2) the node incomes in a failure but the system administrator has not noticed it yet
- (3) the node is recognised as faulty and removed from production
- (4) represents the maintenance period when the node is mostly in an idle state;
- (5) the administrator runs the final check (after having solved the issue) and prepares the node for re-insertion in production;
- (6) the Nagios flag is set again to a normal state and the node is made available.

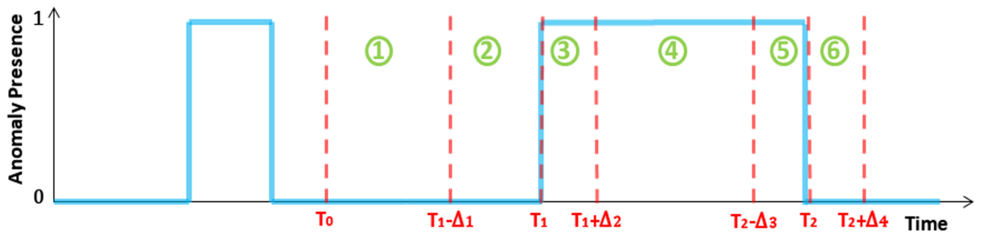

Figure 1: Nagios anomaly signal: value 0 if normal state, value 1 if anomalous state (from [3]).

Detecting an anomalous state before the intervention of system administrators means that *the model should have high accuracy in all phases except (2)*, as in this phase a faulty situation is already ongoing, albeit not detected yet. In this phase, a *high number of false positives* is the expected result.

3 Detect anomalies using Autoencoders

An autoencoder is a type of neural network that is designed to encode input data into a smaller or larger dimensional space and then reconstruct it as output. It consists of three main components: the Encoder, the Code, and the Decoder. Typically, the Encoder and the Decoder are of the same dimension (Figure 2).

Autoencoders are useful for anomaly detection, particularly in supercomputer nodes that rarely experience failures. They are trained exclusively on "normal" data to learn the node's typical state. This data serves as both the input and the target. By encoding the input data into a latent space with different dimensions, the model is prevented from learning a simple identity function. Consequently, it reconstructs the input into a similar, yet not identical, output, maintaining a low reconstruction error.

When the autoencoder is presented with anomalous data, which is assumed to be significantly different from the normal data, the reconstructed output will greatly differ from the input, resulting in a high reconstruction error. In this way, the data can be detected as anomalous.

After training, a **threshold** for the reconstruction error can be calculated. Data is classified as anomalous if the error exceeds this threshold, and as normal if it does not.

This model differs from supervised methods that need datasets with numerous examples of anomalous states, which are generally scarce and difficult to collect in real-world scenarios.

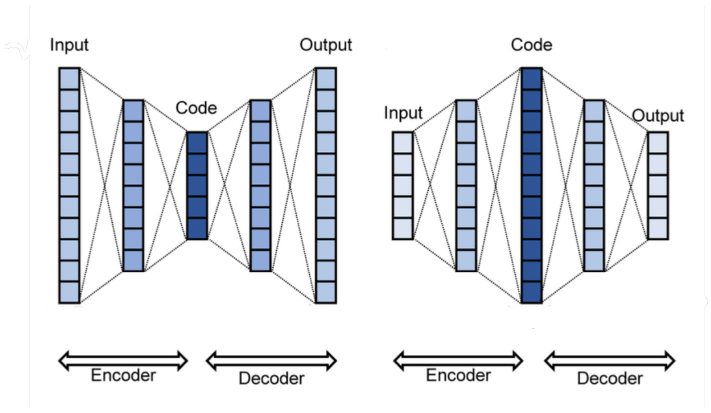

Figure 2: On the left an Undercomplete autoencoder; on the right an Overcomplete autoencoder

4 Dataset description

This research was developed only upon the last month (2022-09) collected data, as it's one of the periods with more active and available metrics. The data can be downloaded here [5].

The data is composed of ExaMon plugins, each one divided into their respective metrics (in *parquet* format). The provided query tool (available here [6]) is used in this research to aggregate the metrics into a pandas DataFrame, one for each node.

It follows a schematic description of all the exploration that has been done for each plugin. The code is available in the notebook called *Data_ Exploration*[1]. A more detailed description is available here [7][8].

4.1 Preliminary operations

Firstly, it has been detected which are the nodes with the most data available. Secondly, there were applied the following filters: (i) filter out the nodes that don't change in `nagiosdrained` state (that's the ones that have `nagiosdrained` values equal to 1 or 0, all along the period); (ii) filter out the nodes that start with the anomalous state (`nagiosdrained = 1`), because we'd like to observe the moments before an anomaly to occur; (iii) filter-in only the nodes with at least two separate anomalous states (i.e. normal state –> anomalous state –> normal state –> anomalous state).

4.2 Nagios plugin

Other than the `nagiosdrained` feature discussed before, it contains several metrics like system status, ping-alive signals, file system and graphic card states. The `nagiosdrained` sampling frequency is *15 minutes*.

Figure 3 displays the `nagiosdrained` signal compared with another metric. It's possible to notice how the metric once reported a faulty state, while the Nagios signal remained flat.

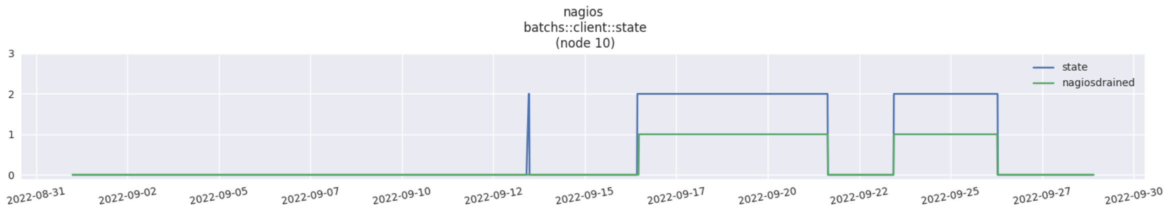

Figure 3: `nagiosdrained` signal compared with the metric `batchs::client::state`. On the x-axis the timestamps, on the y-axis the possible code states.

4.3 Ganglia and IPMI plugins

These plugins describe the system more in detail, for example: GPU and CPU temperatures, load and utilisation; network, disk and memory usage; node internal temperatures, fan speeds, input power, etc. For both of the plugins, it was applied the same data manipulations:

- Remove metrics that remain constant in value throughout the entire period under review, as unchanging data between normal and anomalous node states do not provide any useful information.
- Verify that were present only numerical metrics
- Verify that were not present null elements
- Visualise the number of samples for each metric and the unique values, to identify which one has few data and few unique values
- Visualise the first and last timestamp for each metric, since they can be different
- Calculate the mean, median, and maximum delta-time between consecutive samples for each metric. It was noticed that the mean and median values were nearly identical. This suggests that the metrics were sampled at a consistent rate most of the time. This consistency is likely due to the initial selection of nodes with the highest number of values, resulting in few gaps during the sampling of metrics.

To construct the final dataset, all this data will later be merged with the Nagios dataset using the timestamp column as the key. Consequently, the Ganglia and IPMI data have undergone an aggregation (a "resample") to align with the Nagios time-delta (the sampling rate of Nagios) of 15 minutes. The **mean** was selected as the aggregation operation.

Finally, all empty rows were removed and the new dataframe was saved locally.

4.4 Job Table plugin

It contains information on the jobs executed on the nodes: job state, job start and end time, CPUs requested and allocated, etc.

The "job_state" feature contains the job outcomes and can assume different values like "COMPLETED", "FAILED", "CANCELLED", "OUT_OF_MEMORY", etc.

Like the previous plugins, a procedure for rebuilding the plugin data was taken. The new dataframe is built with these characteristics:

- "timestamp" column: a series of timestamps, equally spaced by 15 minutes, from the start to the end of Nagios time range
- "state" column: the state of the job executing at that moment. At the end, this feature is given to a OneHotEncoder
- if any job is not executing at a given timestamp, the state is set to "IDLE"
- if more than one job is executing at the same timestamp, the state of the *longer* job is taken, albeit parallel jobs were very few. This solution was chosen because are not known all the possible states, so a complete priority mechanism was not possible.

In Figure 4 is possible to see the jobs labelled "COMPLETED" and "FAILED" of a single node.

However, a new real observation would never possess the job state at any given instant, because the state of a job is known at the end of the job itself. The only information that can always be obtained is whether the node is running any job or not: that is the "IDLE" column. Consequently, that's the only column taken.

Interestingly, the final results of the anomaly prediction model show little variation between training on all job states columns and training only on the "IDLE" column. The difference is worsening on the order of hundredths when using only the "IDLE" column for precision, recall and F1 score.

4.5 Logics, Schneider, Slurm plugins

Those plugins capture more information about temperatures, workload and external weather. However, the data is associated not with nodes but with panels and partitions. Since a mapping of the nodes on those other locations is not known, this data was not used.

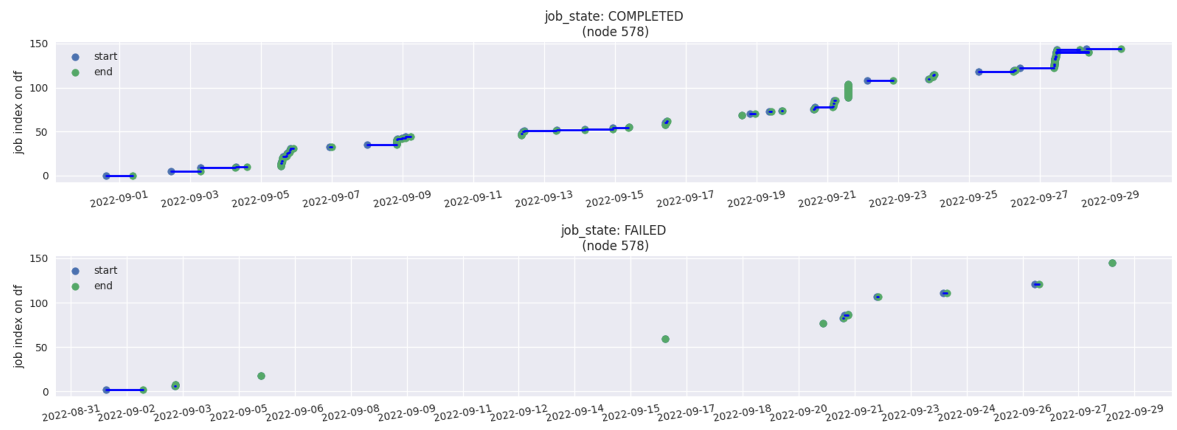

Figure 4: Completed and failed jobs on node 578. On the x-axis the timestamps, and on the y-axis the job id. The job start (azure dot) and the job end (green dot) are connected with a blue line.

4.6 Data aggregation

After the initial preparation, all plugin datasets are combined into a single pandas DataFrame, using the timestamp column from each dataset as the key for an outer join. The aggregated DataFrame is relative to only one node.

Given that each metric has a different sampling rate and period, the merged DataFrame may contain some missing values. To address this without compromising the data integrity, especially for the data that will end in the test set, no data interpolation is applied. Instead, rows with any missing value are removed (using the `df.dropna()` command).

However, before that operation, rows with missing Nagios (`nagiosdrained`) values are firstly dropped. Subsequently, only the *features* (metrics) with at least a certain percentage of non-null values relative to the total DataFrame size are retained. For example, if the threshold is 80% and there are 100 samples in total, only columns with at least 80 non-null samples are preserved:

```
df = df.dropna(thresh=df.shape[0] * NAN_THRESH_PERCENT, axis=1)
```

Without this step, features with a small number of values would be the reason of many dropped rows as a result of the `df.dropna()` command.

It follows an example of the results of this step, using `NAN_THRESH_PERCENT = 95%` and 80%. It is noticeable how a lower threshold implies more dropped rows, yet it also leads to a greater number of features.

```
NAN_THRESH_PERCENT = 95%
--------------------------------------------------------------
Number of rows: original(2680) - removed(210) = 2470 rows
Number of cols: original(173) - removed(52) = 121 cols

NAN_THRESH_PERCENT = 80%
--------------------------------------------------------------
Number of rows: original(2680) - removed(593) = 2087 rows
Number of cols: original(173) - removed(0) = 173 cols
```

5 Semi-Supervised approach

5.1 Extract anomalous data, split and preprocess

Given the aggregated DataFrame composed of data from the ExaMon plugins, the `nagiosdrained` column distinguishes anomalous data from normal data. A value of 1 indicates that the row is labelled as anomalous.

The data is split in a supervised manner, as in Figure 5. The train set for anomalous data is not needed because the model will be fitted only on normal data. The validation data is used to calculate the threshold. The percentages are indicative and can be modified, especially the ones for anomalous data. Indeed, if there are too few anomalous instances, the split may result in an error due to an empty validation set.

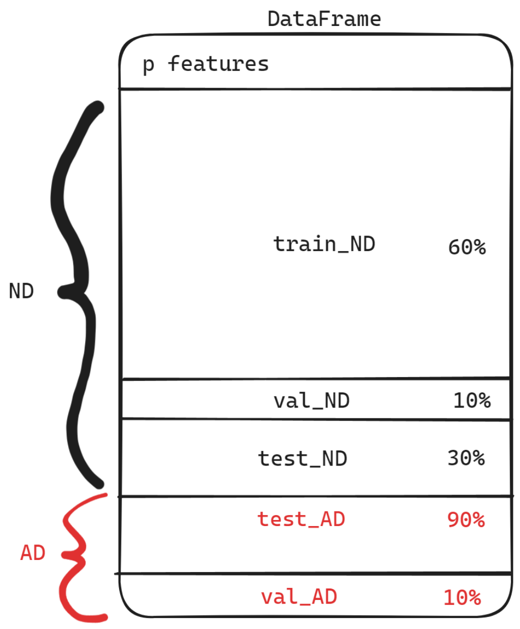

Figure 5: Data splitting. "ND" means normal data, "AD" means anomalous data.

This research aims to test the model particularly during the moments right before the Nagios signal flags an anomaly (phase (2) of Figure 1). Therefore, all non-anomalous observations occurring during that period are moved to the test set of normal data. In other words, this data is the "almost anomalous data" a delta-time before an anomaly has been flagged by Nagios signal. That delta-time is set to *2 hours*.

Finally, a MinMaxScaler is fitted on the train set of normal data and applied to all sets. No further preprocessing is applied.

5.2 Autoencoder definition

The autoencoder is an overcomplete one, with the following characteristics:

- A single hidden layer, of which dimension that is ten times larger than that of the input/output dimension
- A ReLU activation function is used on the hidden layer and a linear activation function on the output
- Adam as the optimizer and Mean Absolute Error as the loss function
- The training set of only normal data serves both as the input and the target
- The validation set is formed by only normal data

Since the data concerns exclusively to a single node, each node possesses its individual autoencoder. In Figure 6 is visualised an example of training and validation loss.

5.3 Threshold calculation

The threshold is used to categorise new data as anomalous or normal data. It's calculated this way:

1. The *validation sets* of both normal and anomalous data are fed into the autoencoder, producing the respective decoded validation sets.

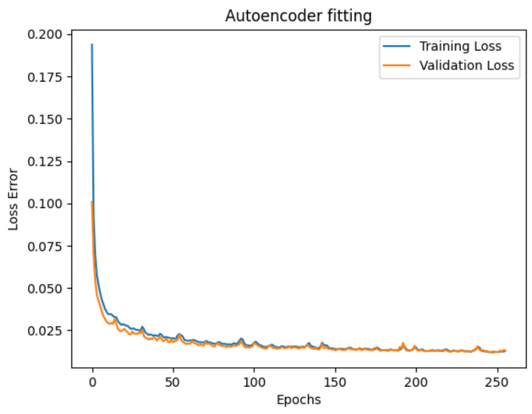

Figure 6: Training and Validation loss over 250 epochs

2. Consider only the validation set of normal data. Is created the reconstruction error matrix of the set, as the absolute difference between the original and decoded data. For each row of the matrix, the maximum value is taken and saved in a list. In other words, each element in this list is the **maximum reconstruction error** that each observation has made, calculated as the maximum absolute error of each row. Consequently, the length of the list corresponds to the number of observations in the validation set. The same procedure is repeated for the validation set of anomalous data.

```
max_errors_list_valid_ND =
    np.max(np.abs(decoded_val_ND - val_ND), axis=1)
max_errors_list_valid_AD =
    np.max(np.abs(decoded_val_AD - val_AD), axis=1)
```

3. The two error lists are concatenated:

```
errors =
    np.concatenate((max_errors_list_valid_ND, max_errors_list_valid_AD))
```

And a series of zeros and ones are created, representing the true classifications of the observations above:

```
classes =
    np.concatenate(([0] * val_ND.shape[0], [1] * val_AD.shape[0]))
```

4. Next, a first threshold value is calculated as the n-th percentile of the maximum error list of *normal* validation data:

```
error_threshold = np.percentile(max_errors_list_valid_ND, n_perc)
```

The following steps are repeated for n-th percentile from 1 to 100.

5. The error threshold is used to classify the error list created before (which contains *both normal and anomalous validation data*), generating a list of boolean predictions:

```
predictions = []
for e in errors:
    if e > error_threshold:
        predictions.append(1)
    else:
        predictions.append(0)
```

6. The list of boolean predictions is then compared with the correct classes:

```
precision_recall_fscore_support(
        classes, predictions, average="weighted", zero_division=0)
```

7. It's chosen the error threshold which generates the best F1 score. In Figure 7 are visualised the different scores and the n-th percentile with best F1 score.

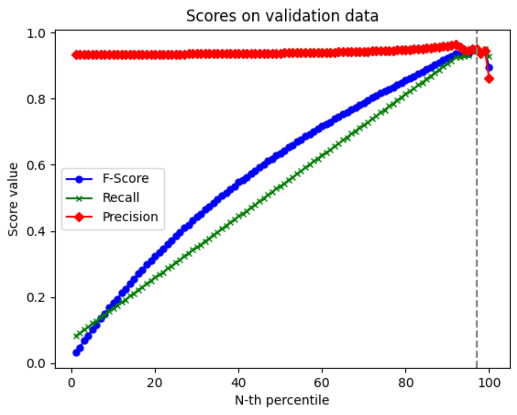

Figure 7: Precision, Recall and F-score of the predicted classes vs the true classes of validation data

5.4 Evaluate the model on unseen data

Feeding the model with unseen data means (1) decoding the new data using the autoencoder; (2) calculating the maximum absolute reconstruction error of the observation, as seen before; (3) classifying the data (as anomalous or not) comparing the calculated error with the error threshold chosen before.

The final results on the test sets vary from node to node. In general, the results are optimal, with scores (precision, recall, f-score) varying from 0.85 to 1. When considering the nodes with the most available data, all the scores are very close to a perfect prediction. Generally, the more samples are available, the more the model is accurate. Consequently, choosing the best threshold NaN-percentage during the dataset aggregation (in 4.6) is a critical phase, as it can significantly reduce the number of total samples.

It follows the results of the same node (578), changing the threshold NaN-percentage:

NAN_THRESH_PERCENT = 95%

```
                 precision    recall  f1-score   support

   Normal data      1.0000    0.9917    0.9959       727
Anomalous data      0.9130    1.0000    0.9545        63

      accuracy                          0.9924       790
     macro avg      0.9565    0.9959    0.9752       790
  weighted avg      0.9931    0.9924    0.9926       790
```

NAN_THRESH_PERCENT = 80%

```
                 precision    recall  f1-score   support

   Normal data      1.0000    0.9838    0.9918       617
Anomalous data      0.8507    1.0000    0.9194        57

      accuracy                          0.9852       674
     macro avg      0.9254    0.9919    0.9556       674
  weighted avg      0.9874    0.9852    0.9857       674
```

5.5 Detect False Positive of Anomalous Data

Finally, it's interesting to see how the model has classified the observations that happen in the time right before a Nagios signal rises. Almost on all the examined nodes, the model has classified those observations as false positives, detecting the anomaly from 15 to 90 minutes before Nagios.

It follows a series of results of different nodes, comparing the model predictions with the `nagiosdrained` signal. Data points are plotted in their original positions. Red crosses indicate incorrect predictions: a value of 0 signifies a false positive, while a value of 1 indicates a false negative. The false negatives are almost absent.

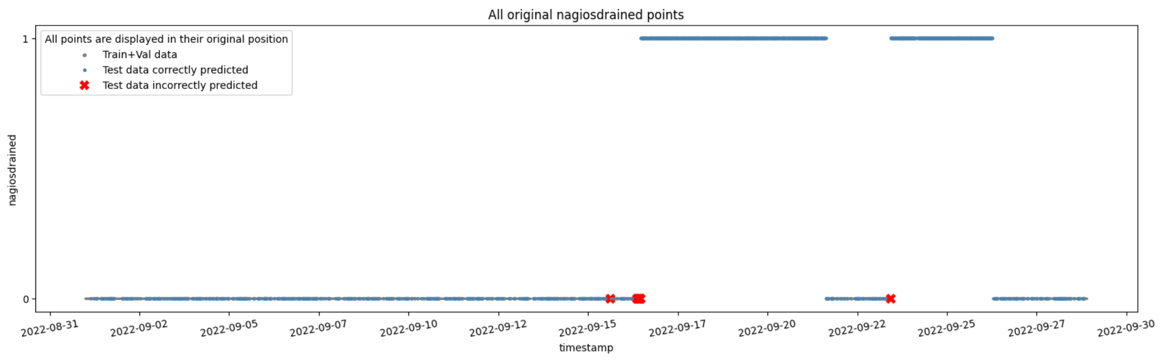

Figure 8: Node 10

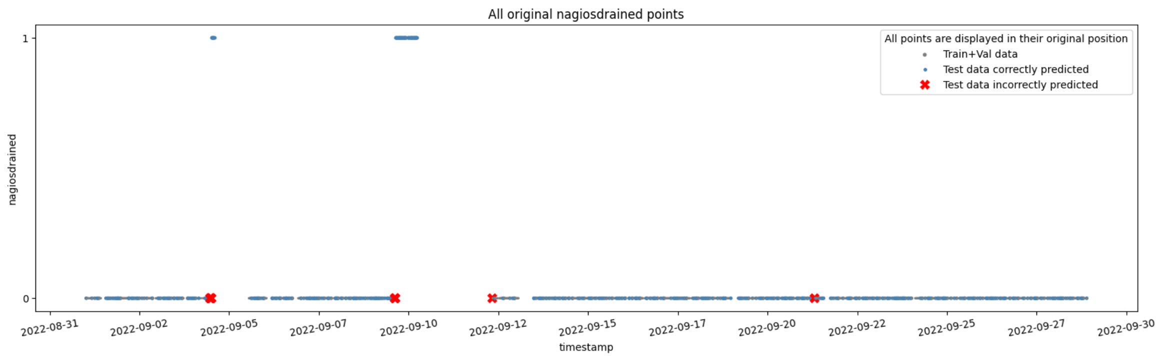

Figure 9: Node 578

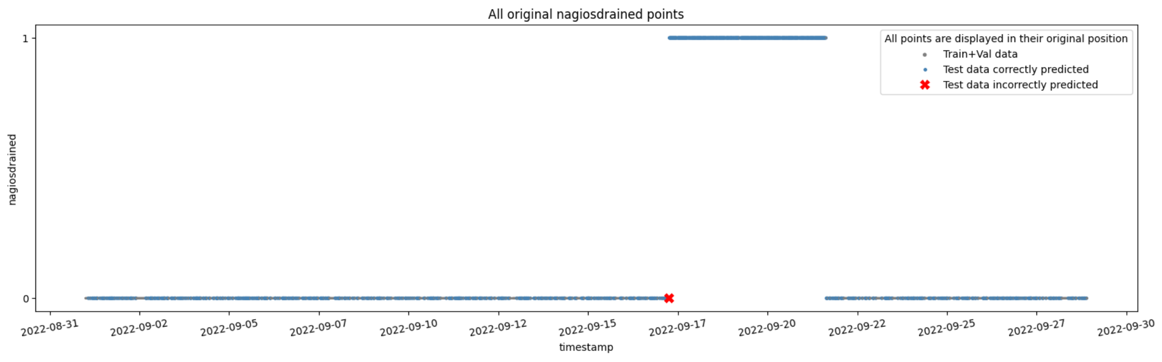

Figure 10: Node 335

6 Conclusions

The research presents a study using an overcomplete autoencoder for anomaly detection in real High-Performance Computing data. The model showed optimal results, with precision, recall, and F1 scores ranging from 0.85 to 1, varying from node to node. The high scores reflect the model's ability to correctly identify true positives and negatives while minimising false positives and negatives. The study also found that the choice of threshold for non-null values during data aggregation was important for the model's performance. Additionally, the model could detect anomalies up to 90 minutes before they were identified by the Nagios system.

References

[1] Code repository of the research.
https://github.com/RiccardoEvangelisti/HPC-Anomaly-Detection

[2] Marconi100 supercomputer details.
https://www.hpc.cineca.it/systems/hardware/marconi100/

[3] Borghesi, A., Molan, M., Milano, M., Bartolini, A. (2022). Anomaly detection and anticipation in high performance computing systems. IEEE Transactions on Parallel and Distributed Systems, 33(4), 739–750. https://doi.org/10.1109/tpds.2021.3082802

[4] Borghesi, A., Di Santi, C., Molan, M., Ardebili, M. S., Mauri, A., Guarrasi, M., Galetti, D., Cestari, M., Barchi, F., Benini, L., Beneventi, F., Bartolini, A. (2023, May 18). M100 EXADATA: A data collection campaign on the Cineca's marconi100 Tier-0 supercomputer. Nature News. https://www.nature.com/articles/s41597-023-02174-3

[5] Source of data analysed.
https://zenodo.org/records/7590583

[6] Query tool to easily access the data.
https://gitlab.com/ecs-lab/exadata/-/tree/main/parquet_dataset/query_tool

[7] ExaMon Plugins description, in detail.
https://gitlab.com/ecs-lab/exadata/-/tree/main/documentation/plugins

[8] A different ExaMon Plugins description.
https://gitlab.com/ecs-lab/exadata/-/blob/main/data_catalog/examon_data_catalog.ipynb