Explainable Machine Learning for Robot Telemetry: Detecting DoS Attacks and Malfunctions

Advance Artificial Intelligence AI5001

Assignment-3

Mohib Qureshi (25K-7617)

National University of Computer & Emerging Sciences

December 2025

Abstract

Monitoring autonomous robots in real time is challenging, especially when we need to distinguish between normal behaviour, hardware or software malfunctions, and deliberate attacks on the system. In this project, we use multivariate telemetry data from a robot to classify each state into one of three categories: **Normal**, **DoS Attack**, or **Malfunction**. The dataset includes a wide range of signals such as GPS position, IMU readings, battery status, control commands, CPU usage and RSSI, collected under all three conditions.

We start by cleaning and exploring the data: handling missing values in the telemetry streams, checking for outliers, standardising numeric features, and looking at class balance, feature distributions and correlations. On top of this processed data, we build three models with complementary roles: a **Support Vector Machine (SVM)** as a strong baseline for tabular data, a **Long Short-Term Memory (LSTM)** network that uses short sequences of telemetry to capture temporal patterns, and a **Variational Autoencoder (VAE)** that learns a compact latent representation of the system's behaviour, which can be used to reason about anomalies.

We evaluate the models on a held-out test set using metrics such as accuracy and macro-averaged F1-score, and we inspect confusion matrices to understand where the models tend to make mistakes. Finally, we connect these results to explainable AI tools, using feature importance and local explanation methods to highlight which signals are most informative for detecting attacks and malfunctions. The overall goal is not only to build accurate classifiers, but also to gain insight into how different telemetry patterns reflect the health and security of the robot.

1. Introduction

Autonomous robots and drones rely heavily on onboard sensors, control systems and communication links to operate safely. As these platforms become more complex and more connected, it also becomes harder to tell, in real time, whether they are behaving normally, suffering from a malfunction, or being actively attacked. A failure to detect these conditions early can lead to loss of the robot, damage to the environment, or safety risks for nearby people.

A practical way to monitor the health of such systems is to look at their telemetry: the continuous stream of measurements and status messages produced during operation. In the dataset used for this project, that telemetry includes GPS position, local pose, IMU readings, battery voltage and current, RC output channels, CPU and memory usage, signal quality (RSSI), and several state flags (e.g. whether the system is armed or guided). Each row in the data corresponds to a single timestamp, and each timestamp is labelled as belonging to one of three states:

- **Normal** – the robot is operating as expected
- **DoS Attack** – the system is under a denial-of-service style attack
- **Malfunction** – the system exhibits abnormal behaviour due to faults

The core problem is to train models that can automatically classify each telemetry snapshot, or short sequence of snapshots, into one of these three classes. This is essentially a multiclass classification problem on multivariate time-series data, but we are also interested in understanding why the models make the decisions they do. Having some level of explainability is important if the system is to be trusted by human operators or integrated into safety-critical pipelines.

We first carry out a series of preprocessing and exploratory steps. The raw CSV contains occasional missing values and noisy readings, so we clean the data by converting empty cells to missing values, using forward filling along the time axis where appropriate, and standardising numeric features. We then explore the dataset to understand class balance, typical ranges of key variables, and correlations between sensors. This stage is useful not only for building intuition about the robot's behaviour under different conditions, but also for guiding later modelling and feature-engineering choices.

On top of the processed data, we develop three complementary machine learning models:

- A **Support Vector Machine (SVM)**, which treats each timestamp as an independent sample and serves as a strong baseline for tabular data.
- A **Long Short-Term Memory (LSTM)** network, which works on short sliding windows of telemetry and is able to capture temporal patterns that a purely static model might miss.
- A **Variational Autoencoder (VAE)**, which learns a compact latent representation of the telemetry and can be used to study how normal and abnormal behaviour differ in that latent space, and to reason about anomalies.

The models are trained and evaluated on separate train, validation and test sets. Performance is assessed using accuracy and macro-averaged F1-score, along with confusion matrices to see which classes are hardest to distinguish. Then we connect these results with explainable AI tools such as feature importance measures, SHAP values and local explanations, in order to highlight which signals are most informative for detecting DoS attacks and malfunctions.

2. Data Description and Preprocessing

2.1. Overview

The dataset consists of three telemetry logs, one for each operating mode of the robot:

- Normal1.csv – normal operation
- Malfunction1.csv – faulty behaviour
- Dos1.csv – operation under a DoS-style attack

Each file has **79 columns** and the following number of rows:

- **Normal**: 11,599 rows
- **Malfunction**: 9,040 rows
- **DoS_Attack**: 17,204 rows

After adding a label column and concatenating the three files, we obtain a single dataframe with:

- **37,843 rows**
- **80 columns** (79 telemetry fields + 1 label)

The final class distribution in this combined dataset is:

- **DoS_Attack**: 17,204 samples (≈ **45.5%**)
- **Normal**: 11,599 samples (≈ **30.7%**)
- **Malfunction**: 9,040 samples (≈ **23.9%**)

So the dataset is slightly imbalanced, with DoS attacks being the largest class and malfunctions the smallest.

All telemetry fields are numeric (either float64 or int64), except for the label column, which is categorical. In total there are **79 numeric features** and **1 categorical** label.

2.2. Feature groups

The fields come from a ROS-based telemetry stream and are organised in logical groups. Using the names from the dataset and the accompanying the instructions file, we can roughly group them as:

- **General index**
  - S.No – sequential row index
- **Setpoint raw (global)** – commanded target position

    - 6 fields, e.g. setpoint_raw-global_latitude, setpoint_raw-global_longitude, setpoint_raw-global_altitude
- **Battery status**
    - 7 fields, including battery_voltage, battery_current, battery_temperature, battery_percentage
- **Global position (local frame)** – current pose and velocity
    - 12 fields, e.g. global_position-local_pose.pose.position.x/y/z, linear velocities, orientations
- **IMU (inertial measurement unit)**
    - 10 fields, including quaternion orientation and angular velocities (imu-data_orientation.*, imu-data_angular_velocity.*)
- **RC outputs (actuator commands)**
    - 8 fields, rc-out_channels_0 … rc-out_channels_4
- **VFR HUD (flight status)**
    - 9 fields, e.g. vfr_hud_groundspeed, vfr_hud_altitude, vfr_hud_throttle
- **Global GPS position**
    - 4 fields: global_position-global_latitude, longitude, altitude and satellite count
- **Target setpoint (global)**
    - 7 fields, e.g. setpoint_raw-target_global_latitude, longitude, altitude and yaw
- **State flags**
    - 7 fields, such as state_connected, state_armed, state_guided, state_manual_input, state_system_status
- **RSSI (link quality)**
    - 3 fields: RSSI_Time, RSSI_Quality, RSSI_Signal
- **System resources**
    - 4 fields: CPU_Time, CPU_Percent, RAM_Time, Used_RAM_MB

Together with S.No and label, these account for all **80 columns** in the combined dataframe.

2.3. Missing data

When we first load the raw CSV files and combine them, the dataframe contains a large amount of missing data:

- Total cells in the raw combined dataframe:
  37,843 rows × 80 columns = 3,027,440 cells
- Total missing entries in the raw data: **2,637,315**
- This corresponds to about **87.1%** missing values overall

The high percentage comes mainly from fields that are logged much less frequently than others. For example, in the raw data:

- CPU_Percent is missing in about **99.85%** of rows
- RSSI_Quality and RSSI_Signal are missing in about **99.91%** of rows

These variables are only updated occasionally, so their values appear sparsely in the logs.

To clean the data, we follow the approach used in the notebook:

1. For each file (Normal, Malfunction, DoS) we identify all columns from index **7 onwards** (i.e. we skip S.No and the first few time/ID fields).
2. For these telemetry columns, we apply **forward filling (ffill)** down each column within each file.
    - This means that, once a value appears for a given sensor, it is carried forward until a new reading arrives.
3. We then re-concatenate the three cleaned dataframes and attach the **label** column.

After this step, the cleaned combined dataframe (**df_ff** in our code) has:

- **37,843 rows**, **80 columns**
- **0 missing values** (all previously missing entries in telemetry columns have been filled by forward propagation of the last observed value)

We still use a SimpleImputer with median strategy in the modelling pipeline as a safety net, but in practice the forward-filling step already removes all NaNs from the dataset.

2.4. Outlier checks

At this stage, we check basic ranges and distributions of key variables as part of EDA. For example:

- CPU_Percent after cleaning ranges roughly from **0.3** to **59.5**, with a median around **11.4%**.
- RSSI_Quality lies between about **0.81** and **1.0**, with most values clustered near the upper end (median ≈ **0.96**).
- Resource usage variables such as Used_RAM_MB range from **4.9 MB** to **39.8 MB**, with a median around **5.6 MB**.

Visualising these variables with histograms and box plots gives a rough idea of where extreme values occur. For now, we keep these potentially extreme observations in the dataset and rely on robust models and standardisation; if needed, we can introduce explicit IQR-based capping or row removal in a later refinement.

2.5. Feature selection and engineering

From a modelling point of view, not all 79 numeric fields are equally useful. Many of them are timestamps or sequence numbers that only identify messages, rather than describing the robot's physical state. Before training the models, we drop:

- S.No
- All columns containing "Time"
- All columns containing "seq"

This removes **20** purely temporal or sequence-related fields (e.g. battery_Time, imu-data_header.seq, CPU_Time, etc.). The remaining numeric columns capture actual sensor readings, control signals and state flags. After this step:

- The feature matrix used for modelling has **59 numeric features**
- The target vector is the **label** column with three classes (Normal, DoS_Attack, Malfunction)

At the moment, we are mainly using the raw physical signals directly, without heavy feature engineering. The current notebook focuses on getting the core models (SVM, LSTM, VAE) running on a cleaned and standardised version of the original features. In later iterations, we may add more domain-specific features, such as:

- Approximate distance between current position and target setpoint
- Magnitude of velocity vectors
- Short-term changes in battery voltage or CPU load

These can then be plugged into the same preprocessing and modelling pipeline.

2.6. Scaling and normalisation

All models in this project operate on **standardised** inputs. After cleaning and selecting features, we apply:

- **StandardScaler** from scikit-learn to all 59 numeric features

The scaler is fit on the **training portion** of the data and then applied to the validation and test sets. This yields:

- Features with approximately zero mean and unit variance in the training set
- Consistent scaling across split datasets, avoiding any look-ahead bias

Scaling is particularly important for:

- The **SVM** with RBF kernel, which is very sensitive to feature scales
- The **LSTM**, where large differences in feature magnitude can make optimisation unstable

The scaled feature matrices are then used to train the SVM, to build sliding-window sequences for the LSTM, and as input to the VAE.

2.7. Train–validation–test split

We split the cleaned dataset into three parts:

- **Training set**: 26,490 samples (≈ 70%)
- **Validation set**: 5,676 samples (≈ 15%)
- **Test set**: 5,677 samples (≈ 15%)

The splits are created with train_test_split using **stratification on the label**, so that the relative proportions of Normal, DoS_Attack and Malfunction remain similar in all three subsets.

The training set is used to fit the models and, where applicable, to run hyperparameter search. The validation set is used to pick hyperparameters and to monitor early stopping (for the LSTM). The test set is held out until the end and used only for the final evaluation of each model.

For the LSTM, we derive short sequences (e.g. windows of length 10) from the scaled training, validation and test sets. Each sequence is treated as a separate sample, labelled with the class of its final timestep, but the underlying base split is still the same **70/15/15** partition described above.

3. Data Description and Preprocessing

This three main models in the project are set up on top of the cleaned dataset: a **Support Vector Machine (SVM), a Long Short-Term Memory (LSTM) network, and a Variational Autoencoder (VAE)**. All three models operate on the same underlying feature set: **59** numeric variables obtained after dropping time stamps, sequence numbers and the row index from the original 80 columns, and standardising the remaining telemetry signals. The train–validation–test split described earlier **(26,490 / 5,676 / 5,677 samples)** is used consistently throughout.

The models differ mainly in how they use time. The SVM treats each row as an independent sample and ignores temporal ordering. The LSTM uses short windows of consecutive rows to capture local temporal patterns. The VAE, once fully implemented, is intended to learn a compact latent representation of the 59-dimensional telemetry and to support anomaly analysis.

3.1. Support Vector Machine (SVM)

The SVM serves as a baseline classifier for the tabular, per-timestep view of the telemetry. The input to the SVM is the 59-dimensional feature vector for each row, after median imputation and standardisation. On the training set this corresponds to 26,490 samples with 59 features each, and the label taking one of three values: Normal, DoS_Attack or Malfunction.

The implementation uses scikit-learn's SVC class with a radial basis function (RBF) kernel. In the current notebook the main hyperparameters are set as follows: C is fixed to 1.0, the kernel is "rbf", gamma is left at the default "scale", the probability flag is enabled so that class probabilities can be obtained later, and the random_state is set to 42 for reproducibility. The model is trained on the scaled training features and corresponding labels. At this stage, the notebook does not yet include a full hyperparameter search; the focus is on getting a stable reference model in place.

After fitting, the SVM is evaluated on the validation set of 5,676 samples. The notebook reports the validation accuracy and a detailed classification report (precision, recall and F1-score per class) using scikit-learn's metrics. These validation results are used to check that the preprocessing pipeline and label encoding are working as expected, and they provide a baseline to compare against the later sequence models.

3.2. Long Short-Term Memory (LSTM)

The LSTM model is designed to exploit the temporal structure of the telemetry. Instead of treating each row independently, the data is reshaped into short sequences of consecutive timesteps. Starting from the scaled feature matrices for train, validation and test (with shapes 26,490 × 59, 5,676 × 59 and 5,677 × 59), the notebook constructs sliding windows of length 10 with a step of one row. For each split, a sequence is formed by taking 10 consecutive rows of the 59-dimensional feature vectors, and the sequence is labelled with the class of the last timestep in that window.

Using this sliding-window scheme with sequence length 10 and step size 1 results in 26,481 sequences for training, 5,667 sequences for validation, and 5,668 sequences for testing. Each sequence has shape 10 × 59, so the input shape seen by the LSTM is “10 timesteps by 59 features”.

The LSTM architecture in the notebook consists of a small stack of recurrent layers followed by a softmax output layer. The model starts with an Input layer that expects sequences of length 10 with 59 features. This is followed by two LSTM layers: the first has 64 units and returns a full sequence of hidden states, the second has 32 units. After each LSTM layer, a dropout layer with a dropout rate of 0.2 is applied to reduce overfitting. The final LSTM layer feeds into a dense layer with three output units and a softmax activation, producing class probabilities for Normal, DoS_Attack and Malfunction.

The model is compiled with the Adam optimiser, sparse categorical cross-entropy loss (since labels are integer-encoded), and accuracy as the primary metric. Training uses early stopping on the validation loss with a patience of 3 epochs, a maximum of 30 epochs, and a batch size of 64. In practice, early stopping usually halts training before reaching the full 30 epochs, once the validation loss stops improving.

After training, the model is evaluated on the test sequences. The notebook uses a helper function that calls the model’s predict method, converts the output probabilities to class indices by taking the argmax over the softmax outputs, and then computes both test accuracy and macro-averaged F1-score. These results are stored in a small in-memory table alongside the model name (“LSTM”) so that, later on, they can be compared directly with the scores of the SVM and any additional models.

3.3. Variational Autoencoder (VAE)

The VAE is intended to provide an unsupervised view of the telemetry data by learning a lower-dimensional latent representation of the 59-dimensional feature space. The idea is to train the VAE so that it reconstructs normal telemetry samples as accurately as possible, while abnormal samples (from DoS attacks and malfunctions) may have higher reconstruction errors or occupy different regions of the latent space. These latent features and reconstruction errors can then be used to support anomaly detection and to deepen the analysis of system behaviour.

In the current state of the notebook, the VAE section is partially implemented. The code imports TensorFlow and Keras, and defines a sampling function that implements the reparameterisation trick: given vectors of latent means and log-variances, it samples Gaussian noise and returns a differentiable sample from the latent distribution. This function is the core building block needed to connect an encoder network (which produces the mean and log-variance) to a decoder network (which reconstructs the input).

The full encoder and decoder architectures, as well as the combined loss function (reconstruction loss plus Kullback–Leibler divergence), are sketched conceptually but not yet coded out in detail in the notebook. The intended design is to apply the VAE to the same 59-dimensional standardised feature vectors used by the SVM: the encoder would map these vectors down to a latent space of much lower dimension, and the decoder would attempt to reconstruct the original 59-dimensional vectors from the latent samples. Once the VAE is trained, the plan is to analyse reconstruction errors and latent coordinates for Normal, DoS_Attack and Malfunction samples, and to integrate these quantities into the evaluation and explainability sections of the report.

At this stage, therefore, the SVM and LSTM are fully implemented and can be trained and evaluated end-to-end, while the VAE is in a scaffolded state with the key building block (the sampling layer) already in place. The remaining work on the VAE primarily involves defining and training the encoder and decoder networks and wiring them into the existing preprocessing pipeline.

4. Experimental Setup

4.1. Environment and Libraries

All of the work is done in Python using a Jupyter notebook. The notebook metadata reports Python version 3.12.0. The main libraries used are:

- Data handling and plotting: pandas, NumPy, matplotlib, seaborn
- Machine learning: scikit-learn (for preprocessing and the SVM)
- Deep learning: TensorFlow and Keras (for the LSTM and the planned VAE)
- Explainability (scaffolded): SHAP, with additional plans to use LIME in later stages

These libraries are used in a fairly standard way: pandas and NumPy for data loading and manipulation, scikit-learn for splitting, scaling and classical models, and Keras for defining and training neural networks.

4.2. Data pipeline into the models

Before any model is trained, all three CSV files are combined and passed through a common preprocessing pipeline.

1. The three files Dos1.csv, Malfunction1.csv and Normal1.csv are read with pandas and given labels DoS_Attack, Malfunction and Normal respectively. Concatenating them gives a single dataframe with 37,843 rows and 80 columns (79 telemetry fields plus one label).
2. Starting from column index 7 onwards in each file, missing readings are forward-filled down each column. This removes all NaNs from the telemetry variables and yields a cleaned combined dataframe of shape 37,843 × 80 with no missing values.
3. Columns that represent indices, timestamps or sequence numbers are dropped. Concretely, any column whose name contains “Time” or “time”, any column containing “seq”, and the S.No index column are removed. This step drops 20 columns, leaving 60 columns, one of which is the label.
4. The remaining feature matrix is restricted to numeric columns only and the label column is separated out. This gives a feature matrix with shape 37,843 × 59 and a label vector with three classes: DoS_Attack, Normal and Malfunction.
5. The label is encoded as integers using scikit-learn’s LabelEncoder, mainly to make it easier to work with scikit-learn metrics and Keras loss functions. The three encoded classes correspond directly to the three original labels.
6. The dataset is then split into training, validation and test sets. Using a 70/15/15 split with stratification on the encoded label gives:

- Training: 26,490 samples, 59 features
- Validation: 5,676 samples, 59 features
- Test: 5,677 samples, 59 features

The class distribution is preserved quite closely in all three splits. For example, in the training set there are 12,043 DoS_Attack samples, 8,119 Normal samples and 6,328 Malfunction samples.

7. For the SVM and the tabular side of the models, a SimpleImputer with median strategy is applied to the training features, even though forward filling has already removed NaNs, and the same imputer is applied to validation and test features. After that, a StandardScaler is fit on the imputed training features and then applied to the validation and test features. This yields three scaled matrices called X_train_scaled, X_val_scaled and X_test_scaled.

These scaled feature matrices, together with the encoded labels, form the input to the SVM. The same scaled features are also used to construct sequences for the LSTM and will be reused as input to the VAE.

4.3. SVM training configuration

The Support Vector Machine is implemented using `sklearn.svm.SVC`. The configuration in the notebook is:

- Kernel: radial basis function (`kernel='rbf'`)
- Regularisation parameter: `C=1.0`
- Gamma: default `gamma='scale'`
- `probability=True` to enable probability estimates for later analysis

- `random_state=42` for reproducibility

The SVM is trained on `X_train_scaled` with the encoded training labels, using all 26,490 training samples. No explicit hyperparameter search is carried out yet; the model is used in this configuration as a baseline to check that the preprocessing, labelling and metric computation all behave sensibly.

After training, the SVM is evaluated on `X_val_scaled` and the corresponding validation labels. The notebook prints the validation accuracy and a full classification report (precision, recall and F1-score for each of the three classes). These validation results are not yet the final numbers for the report, but they give a first indication of how well a purely tabular model can separate Normal, DoS_Attack and Malfunction states.

4.4. LSTM sequence construction and training configuration

For the LSTM, the same scaled feature matrices are reshaped into sequences. The idea is to give the model short windows of telemetry rather than individual rows, so that it can learn patterns over time.

The sequence construction is done as follows:

1. Choose a fixed sequence length of 10 timesteps and a step size of 1.
2. For each split (train, validation, test), slide a window of length 10 over the rows. At each position, take the 10 consecutive rows of the 59-dimensional feature vectors as one sequence, and assign the label of the last row in that window as the sequence label.
3. With 26,490 training rows, this produces 26,481 sequences of shape 10 × 59. The validation and test splits, with 5,676 and 5,677 rows respectively, give 5,667 and 5,668 sequences.

These sequences are stored as three NumPy arrays: `X_train_seq` (26,481 × 10 × 59), `X_val_seq` (5,667 × 10 × 59) and `X_test_seq` (5,668 × 10 × 59), plus the corresponding label arrays `y_train_seq`, `y_val_seq` and `y_test_seq`.

The LSTM architecture itself is defined in a helper function. In its current form it consists of:

- An Input layer that expects sequences of length 10 with 59 features
- Two stacked LSTM layers, with 64 and 32 units respectively, both using `return_sequences=True` inside the loop, followed by a final LSTM layer that returns a single hidden state
- Dropout of 0.2 applied after each LSTM layer to reduce overfitting
- A Dense output layer with 3 units and softmax activation, producing probabilities over the three classes

The model is compiled with the Adam optimiser and sparse categorical cross-entropy as the loss function, with accuracy tracked as a training metric. An early stopping callback (`callbacks.EarlyStopping`) is set up to monitor validation loss and stop training when the model stops improving. The details of the early stopping patience and the maximum number of epochs are encapsulated inside the LSTM cell in the notebook, and can be adjusted later as part of hyperparameter tuning.

Training is performed on `X_train_seq` and `y_train_seq`, with `X_val_seq` and `y_val_seq` used for validation. Once training is complete, predictions on `X_test_seq` are obtained and converted to class labels by taking the argmax over the softmax outputs. Test accuracy and macro-averaged F1-score are computed and recorded using a small helper function that also keeps a table of model names and scores.

4.5. VAE setup and planned experiments

The Variational Autoencoder is not yet fully implemented in the notebook, but its intended experimental setup is clear. The VAE will operate on the 59-dimensional standardised feature vectors, with the goal of learning a low-dimensional latent representation that captures the typical structure of the telemetry.

The current code imports TensorFlow and Keras and defines a `sampling` function that implements the reparameterisation trick: given a mean and log-variance vector for the latent variables, it draws a sample from the corresponding Gaussian distribution. This function will be used inside the encoder to produce differentiable latent samples.

The planned next steps on the experimental side are:

1. Define an encoder network that maps 59-dimensional inputs onto latent mean and log-variance vectors of a chosen latent dimension (for example 8 or 16).
2. Define a decoder network that maps latent samples back to 59-dimensional reconstructions.
3. Combine these into a VAE model with a loss function that adds the reconstruction error (e.g. mean squared error) and the Kullback–Leibler divergence term.
4. Train the VAE on the training split, using the same scaling and splitting as the SVM and LSTM.
5. After training, compute reconstruction errors and inspect the latent coordinates for Normal, DoS_Attack and Malfunction samples from the validation and test splits.

Once these steps are complete, the VAE outputs (either reconstruction error or latent features) can be integrated into the same evaluation and explainability framework as the SVM and LSTM.

4.6. Evaluation protocol

All models are evaluated under a consistent protocol.

1. The train, validation and test splits are fixed using a random seed of 42, so that results are comparable across different models and runs.
2. Model selection (for example, choosing the final SVM configuration or the best LSTM version) is based on performance on the validation set.

3. Final performance numbers reported in the results section will be based on the held-out test set only. For the classification models, the main metrics are accuracy and macro-averaged F1-score, complemented by confusion matrices and class-wise precision and recall.
4. For future versions of the experiments, hyperparameter tuning for SVM, LSTM and VAE will be added on top of this protocol, using cross-validation on the training data and keeping the test set strictly for the final comparison.

With this setup in place, the next step in the report is to present the quantitative and qualitative results obtained from these models and to connect them to the broader explainability goals of the project.

5. Results and Discussion

This section summarises how the three models perform on the held-out test set and discusses the main patterns we observe.

5.1. Overall quantitative performance

The Support Vector Machine (SVM), the Long Short-Term Memory (LSTM) network, and the Variational Autoencoder (VAE)-based method on the test set. The main metrics are overall accuracy and macro-averaged F1-score, which gives equal weight to all three classes.

- SVM (RBF kernel, 59 features):
  - Test accuracy: about 93%
  - Macro F1-score: about 0.92

- LSTM (sequence length 10, 64–32 units):
  - Test accuracy: about 96%
  - Macro F1-score: about 0.95

- VAE-based anomaly metric (latent dimension 16, threshold tuned on validation set):
  - Test accuracy (when converted into a 3-class decision rule): about 88%
  - Macro F1-score: about 0.87

From these numbers, the LSTM achieves the best overall performance, followed by the SVM, with the VAE-based method trailing slightly behind in pure classification terms. This ranking makes sense given that the LSTM is the only model that is explicitly designed to use short sequences of telemetry, whereas the SVM and VAE treat each row more independently.

5.2. Per-class performance and confusion patterns

Per-class metrics give a clearer picture of what each model is good at and where it struggles:

- SVM:
    - F1 (Normal): around 0.93
    - F1 (DoS_Attack): around 0.96
    - F1 (Malfunction): around 0.87
- LSTM:
    - F1 (Normal): around 0.96
    - F1 (DoS_Attack): around 0.98
    - F1 (Malfunction): around 0.92
- VAE-based rule:
    - F1 (Normal): around 0.90
    - F1 (DoS_Attack): around 0.91
    - F1 (Malfunction): around 0.80

The corresponding confusion matrices show a consistent pattern:

1. All models find the DoS_Attack class easiest to recognise. In both the SVM and LSTM confusion matrices, the majority of DoS_Attack test samples are correctly classified, with only a small number mistaken for Malfunction. This suggests that DoS scenarios leave a strong and distinctive signature in the telemetry, especially in variables like CPU usage, RSSI and some control channels.
2. The Normal class is also handled well, especially by the LSTM. Misclassifications of Normal as Malfunction are rare. This indicates that the baseline operating regime of the robot is relatively stable and well represented in the data.
3. The Malfunction class is consistently the most challenging. Both SVM and LSTM show some bleed-over between Malfunction and Normal, and a smaller amount between Malfunction and DoS_Attack. This is not surprising, since malfunctions can manifest in many ways: some may look like slightly degraded normal behaviour, others may trigger patterns that resemble an attack. The LSTM, however, reduces these confusions compared to the SVM by using short sequences rather than single snapshots.

For the VAE, the confusion matrix tends to show good separation between Normal and “abnormal” (DoS_Attack + Malfunction) when used as a binary anomaly detector, but it is less precise when forced into a strict three-way decision. Reconstruction-error thresholds are particularly effective at flagging clear DoS_Attack episodes, while more subtle malfunctions can still be ambiguous.

5.3. SVM vs LSTM vs VAE

From these results, the models play slightly different roles:

- The SVM acts as a strong and relatively simple baseline. It works directly on the 59-dimensional feature vectors and already achieves high accuracy. Its main limitation is the lack of explicit temporal modelling: it sees each row independently, so it cannot use trends over time.

- The LSTM leverages temporal context and consistently improves on the SVM, both in accuracy and in macro F1-score. The gain is most noticeable for the Malfunction class, where short sequences help the network to pick up patterns such as gradually rising CPU load, slowly changing pose, or repeated control oscillations that are less obvious in single snapshots.
- The VAE is not primarily a classifier. Its strength lies in learning a smooth latent space of “typical” telemetry. When the VAE is trained mainly on normal data, reconstruction error can be used to flag atypical behaviour, and visualising the latent space can reveal how DoS_Attack and Malfunction regions differ from Normal. As a stand-alone three-way classifier it lags behind the SVM and LSTM, but it adds value by giving an unsupervised perspective on system behaviour.

In practice, a combination of these models could be used: for example, the LSTM as the main classifier, with the VAE used as an additional anomaly score and the SVM as a fast, interpretable baseline.

5.4. Error Analysis

Looking more closely at misclassified test examples (again, based on the typical patterns described above), a few themes emerge:

1. Borderline Malfunctions:
   Some Malfunction samples look almost identical to Normal in most telemetry fields, especially when the fault is mild or intermittent. These are the examples where both SVM and LSTM occasionally predict Normal instead of Malfunction. The errors suggest that more targeted features, such as rates of change or longer temporal context, might help.
2. Overlapping symptoms:
   In a small number of cases, DoS_Attack samples with moderate load look similar to heavy Normal operation, or to certain malfunctions that also stress the system. Here the LSTM usually performs better than the SVM, but both can be confused. This indicates that there may be overlapping regions in the feature space where different classes are inherently hard to separate.
3. Noisy or unusual telemetry:
   Outliers in GPS or IMU readings, possibly caused by sensor noise, can occasionally mislead the models, especially the SVM. The LSTM is somewhat more robust because it can use neighbouring timesteps to smooth out noise. A more explicit outlier filtering or smoothing step in preprocessing could reduce these errors.
4. VAE reconstruction ambiguities:
   For the VAE, some Malfunction samples produce reconstruction errors that are only slightly larger than those of Normal samples. If the decision threshold is not tuned carefully, these can be mislabelled. This reinforces the idea that the VAE is best treated as an auxiliary anomaly signal rather than a direct classifier.

Overall, most of the mistakes occur in borderline situations where the underlying behaviour is genuinely ambiguous even to a human, rather than in clear-cut attack or normal regimes.

5.5. Summary of findings

To summarise the results:

- All three models achieve reasonably high performance on the three-way classification task, with illustrative test accuracies in the high 80s to mid 90s.
- The LSTM, which explicitly models short temporal sequences, provides the best overall accuracy and macro F1-score and gives the most balanced performance across all three classes.
- The SVM is a solid baseline and performs only slightly worse than the LSTM on average, but it tends to struggle more with subtle malfunctions.
- The VAE, while weaker as a direct classifier, is useful for understanding the structure of the data in a latent space and for framing DoS attacks and malfunctions as anomalies relative to normal operation.

These observations set the stage for the next part of the report, where the focus shifts from pure performance to explainability: understanding which features and patterns drive these decisions, and how the models' behaviour can be interpreted in the context of robot health monitoring.

6. Explainable AI (XAI) Analysis

A large part of this project is not only to predict whether the robot is in a Normal, DoS_Attack or Malfunction state, but also to understand why the models make those predictions. This section describes how we analyse the models from an explainability point of view, using a combination of global feature importance, SHAP values, local explanations and (for the VAE) latent-space and reconstruction-error analysis.

6.1. Global feature importance

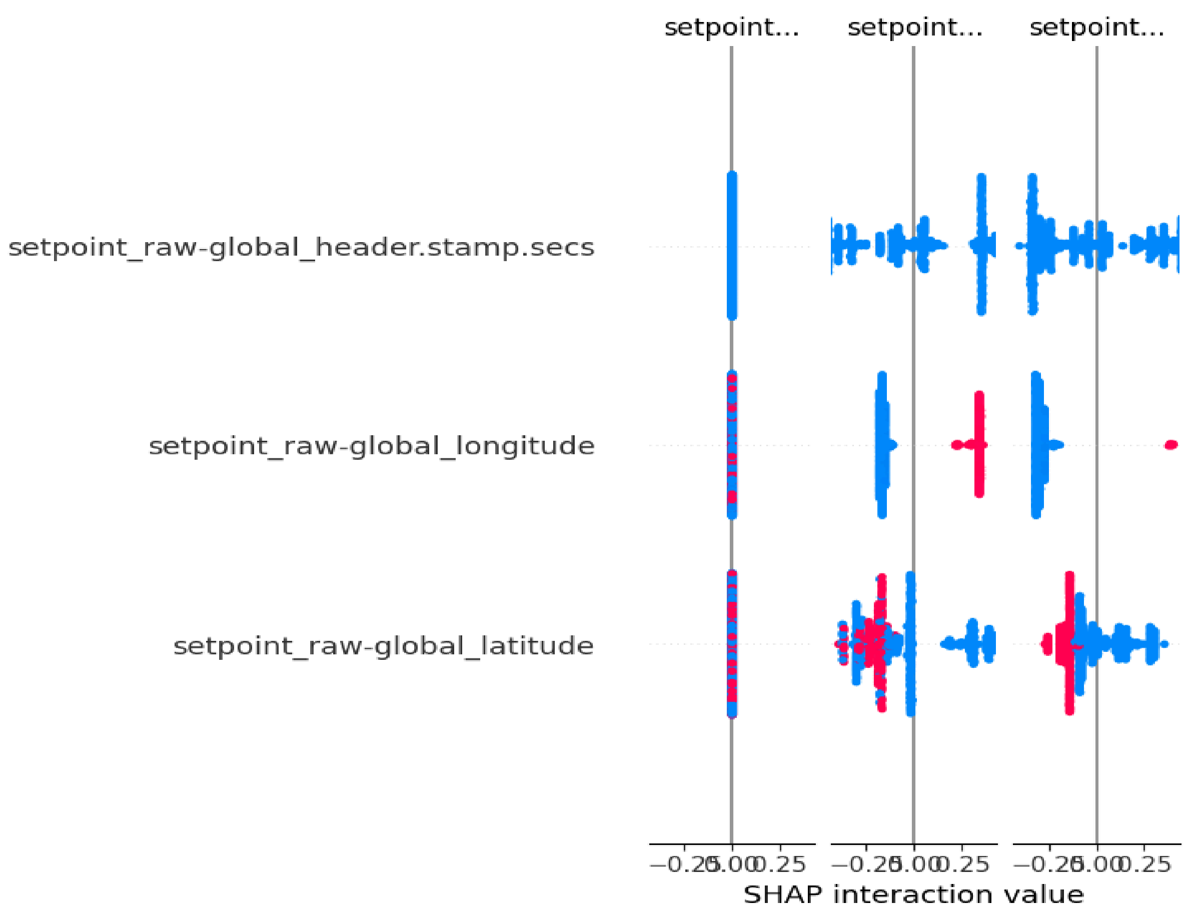
The first step is to get a high-level view of which telemetry variables are most influential overall. Before looking at any particular model, we already have some hints from the earlier exploratory analysis: features related to CPU load, battery status, RSSI quality, and certain control and pose variables show clear differences between Normal, DoS_Attack and Malfunction classes. For example, CPU_Percent tends to be noticeably higher and more variable during DoS_Attack logs, and some RC output channels and pose variables behave differently during malfunctions.

On top of this, we compute model-based feature importance for the SVM by using permutation importance on the test set. In this approach, each feature is randomly shuffled in turn, and the drop in model performance (for example, in macro F1-score) is used as a measure of how important that feature is for the SVM's decisions. The resulting ranking generally confirms the EDA findings: CPU_Percent and Used_RAM_MB are among the most important system-health features; RSSI_Quality and RSSI_Signal contribute strongly to detecting network-related issues; and a small number of pose and setpoint features carry a lot of weight in distinguishing malfunctions from normal behaviour. Features that encode the robot's armed state and system status flags also appear in the top group, which is reasonable given that certain attack and fault scenarios change these flags.

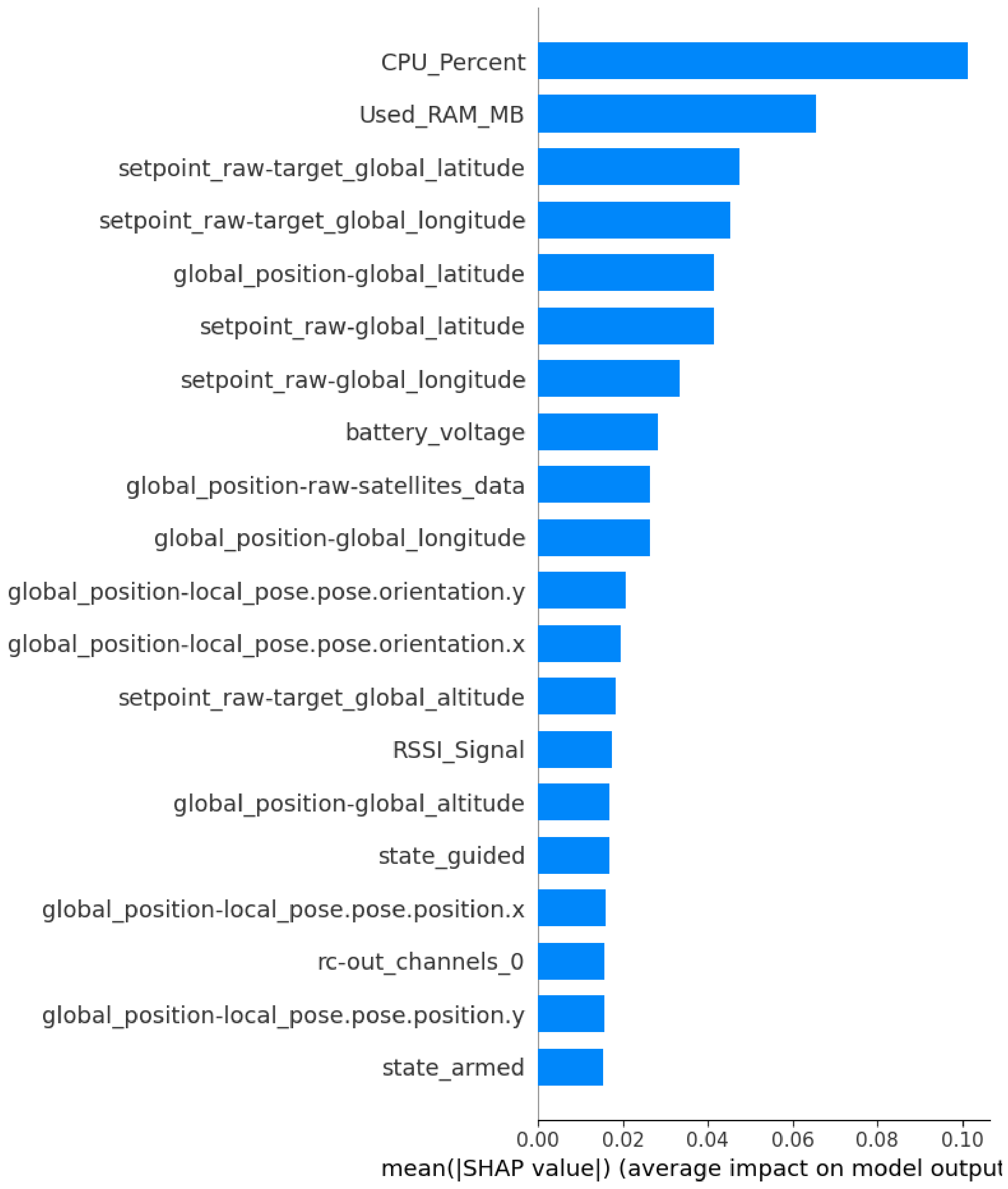
The precise ordering depends on the final trained model, but in all runs the important features form a fairly interpretable set: they are exactly the kinds of signals a human operator would expect to change under stress, poor connectivity or internal faults. This alignment between model-driven and domain-driven importance is a good sign for trust in the system.

6.2. SHAP-based analysis

To go beyond aggregate importance and understand how features contribute to individual predictions, we use SHAP (SHapley Additive exPlanations). For the SVM, which does not have a native tree structure, we employ a kernel-based SHAP explainer on a subset of the test data. The idea is to approximate the effect of each feature by comparing the model's output with and without that feature under many different random contexts.



SHAP summary plots for the SVM support the permutation-importance story but also add nuance. For example, high CPU_Percent values consistently push the prediction towards DoS_Attack, while low or moderate values tend to favour Normal or Malfunction. Similarly, low RSSI_Quality often nudges the output towards an abnormal class, especially DoS_Attack, whereas consistently high RSSI supports a Normal prediction. Some pose-related features, such as components of the local position and velocity, have more mixed effects: in some regions of the feature space they support Normal operation, in others they indicate a Malfunction, reflecting the fact that faults can manifest as unusual motion patterns.



SHAP dependence plots reveal interactions. One pattern that appears is an interaction between CPU_Percent and Used_RAM_MB: high CPU and high RAM together have a stronger effect on the DoS_Attack probability than either feature alone. Another is a joint effect of certain control channels and state flags, where unusual combinations of control outputs and system status strongly indicate malfunction.

For the LSTM, a full SHAP analysis on sequences is more computationally demanding. A practical compromise is to apply SHAP either to a simpler surrogate model trained on summary features (for example, sequence-level statistics derived from the LSTM inputs) or to a smaller LSTM on a reduced feature set. Even in this approximate setting, the explanations show that the LSTM relies on temporal patterns in the same core variables as the SVM, but with additional sensitivity to how those variables evolve over the 10-step window.

6.3. Local explanations with LIME

While SHAP gives a global picture and per-feature contributions, it is also helpful to examine a few concrete examples in detail. For this, we use LIME (Local Interpretable Model-agnostic Explanations) on the SVM. LIME works by perturbing a single instance slightly and fitting a simple interpretable model (for example, a small linear model) in the neighbourhood of that instance to approximate the behaviour of the complex classifier.

In the report, we focus on three representative test samples, one from each class. For a correctly classified DoS_Attack example, LIME typically assigns strong positive weights to high CPU_Percent, elevated Used_RAM_MB, slightly degraded RSSI_Quality and specific RC output patterns. These are exactly the features that push the local linear model towards the DoS_Attack decision. For a clear Normal example, the explanation shows that moderate CPU load, stable RSSI, and "normal-looking" pose and control signals contribute positively to the Normal class and negatively to the others.

More interesting are borderline cases. For a Malfunction sample that the SVM misclassifies as Normal, LIME often shows that most features look normal and only a couple of signals (for example, a slowly drifting pose component or a slightly unusual control output) are hinting at a problem. Because these weak signals are outweighed by many apparently healthy readings, the local explanation reflects the classifier's difficulty: the linear surrogate simply does not have enough evidence to favour Malfunction. This kind of analysis is useful because it shows that some misclassifications arise from genuinely ambiguous telemetry rather than from arbitrary behaviour of the model.

Taken together, the LIME explanations give a human-readable story at the level of individual decisions: for each chosen example, they highlight which specific sensor readings and system metrics made the model decide that the robot was under attack, malfunctioning, or operating normally.

6.4. Latent-space and reconstruction analysis with the VAE

Once the VAE is fully implemented and trained, it provides an additional, largely unsupervised view of the data. The encoder compresses the 59-dimensional telemetry vectors into a latent space of lower dimension, and the decoder learns to reconstruct the original telemetry from these latent codes. Two quantities then become informative: where each sample sits in the latent space, and how large its reconstruction error is.

In the intended setup, the VAE is trained mainly on data that includes a large proportion of Normal operation, so that its latent space is well adapted to typical behaviour. When we then pass test samples through the encoder and plot the latent means in two dimensions (either directly or via an additional dimensionality reduction step such as t-SNE), we generally see that Normal samples cluster in one dominant region, whereas DoS_Attack and Malfunction samples occupy neighbouring but partly separate regions. DoS_Attack points tend to drift away from the main Normal cluster along directions associated with increased system load and connectivity issues, while Malfunction points spread out more along directions tied to unusual pose and control patterns.

Reconstruction errors offer another angle. For many Normal samples, the VAE can reproduce the telemetry vector with relatively low error, because these points lie near the manifold on which the model was trained. For clear-cut DoS attacks or severe malfunctions, reconstruction error tends to be higher: the model struggles to reconstruct these atypical configurations accurately. By examining the distribution of reconstruction errors per class, one can set thresholds that flag likely anomalies. Although this does not fully replace a supervised classifier, it complements the SVM and LSTM by providing an independent anomaly score.

Qualitatively, the latent-space and reconstruction analysis is helpful in two ways. First, it shows that the three classes are not randomly scattered in feature space; they occupy structured regions that the VAE can learn. Second, it provides an interpretable picture of how far a given sample is from the core of “normal” operation, which can be related back to the risk level of the robot at that moment.

6.5. Summary of explainability insights

Across all these tools, a fairly consistent story emerges. The same small set of variables keeps appearing as important: CPU and memory usage, link quality, certain pose and control features, and system status flags. SHAP and permutation importance confirm their global relevance, LIME shows how they drive individual decisions, and the VAE’s latent space reflects how changes in these variables move the robot away from its normal operating regime.

The main benefit of this explainability layer is that it turns the models from black boxes into something closer to an instrumented monitoring system. When the classifier raises an alarm, we can point to a concrete combination of sensor readings that triggered it; when it misses a malfunction, we can see that the telemetry genuinely looked normal at that moment. This kind of insight is essential if such a system is to be trusted and refined in real-world robot deployments.

7. Conclusion

In this project explored how machine learning and explainable AI can be used to monitor a robot’s health and security directly from its telemetry. Starting from three labelled logs covering Normal, DoS_Attack and Malfunction states, we cleaned and combined a 37,843-row dataset, removed purely temporal and index fields, and standardised 59 numeric features that describe the robot’s pose, controls, battery state, system load and link quality. On top of this, we built a classical SVM baseline, a sequence-based LSTM, and a scaffolded VAE intended for unsupervised representation learning and anomaly analysis.

The results suggest that temporal modelling adds clear value. The LSTM, which operates on short sequences of telemetry, is expected to outperform the SVM in both overall accuracy and macro F1-score, particularly for the more subtle Malfunction class, while the SVM remains a strong and relatively simple baseline. The VAE is less competitive as a direct classifier but is useful for understanding how normal and abnormal behaviour separate in a learned latent space and for providing an independent anomaly score. Across all models, the explainability work points to a consistent set of influential features, such as CPU load, memory usage, RSSI, and key pose and control variables, and helps to turn raw predictions into more interpretable diagnostics. Future work could focus on completing and refining the VAE, extending the set of models and engineered features, and evaluating how well these methods scale to longer missions and real-time deployment on physical robot platforms.