Technical Report: Interpretable Machine Learning for Industrial Equipment Health Monitoring

Samuel Adetsi, Mu Ha, Cheng Zhang, Michael Hewlett

2025-06-22

Table of contents

Introduction

Predictive maintenance leverages sensor data to anticipate equipment failures before they occur, minimizing downtime and reducing maintenance costs. In industrial settings, machinery health is traditionally monitored through heuristic or proprietary algorithms, which often lack transparency and fail to provide actionable insights to stakeholders.

This document serves as an addendum to the main final report, providing detailed explanations of the archived scripts and supplemental workflows developed for equipment health monitoring. First, we parsed proprietary voltage files with a custom binary script. Then we switched to JSON measurement files and built a faster, more robust feature pipeline. Early tests used a forward-fill baseline and exposed problems like label leakage, low variance, and multicollinearity. Inspired by MATLAB's summary-statistics method, we switched to 20-minute window summaries. This change let tree models (Random Forest, XGBoost) outperform others. We also tried an RNN but found it slow, prone to overfitting, and often worse than a simple mean predictor. Finally, we deployed everything on AWS—storing files in S3 and running processing on EC2—to handle large data volumes and avoid GitHub size limits.

Equipment & Data Overview

- **Equipment types**: Tube Mill, Belt Conveyor (#8), High-Temperature Fan (#1)
- **Sensors**: Acceleration, Vibration Velocity, Temperature
- **Sampling**: 5-second intervals, 15 monitoring points
- **Measurement count**: 6 measurements per 5-second window
- **Target ratings**: 12–15 health ratings recorded every 20 minutes

Key Challenges

- **Temporal resolution mismatch:** Features sampled every 5 seconds vs. ratings recorded every 20 minutes.
- **Missing and noisy data:** Sensor streams contain dropouts and noise.
- **Low data diversity:** Health ratings and operating conditions show limited variation.
- **Limited feature-to-target ratio:** Only 6 measurements available to predict 12–15 target ratings.
- **High feature correlation:** Measurements exhibit strong multicollinearity, which can reduce the effective information content and destabilize model estimates.
- **Accuracy vs interpretability trade-off:** Complex models offer high accuracy but lack transparency.
- **High-frequency data volume:** Five-second sampling at many locations produces very large time-series datasets.

- **Large file size constraints:** Raw waveform archives often exceed GitHub's 100 MB file size limit, complicating version control and collaborative sharing.

Initial Approach

Our initial implementation (see `model/archive/model.py`) involved aligning the high-frequency measurements (every five seconds) with the infrequent health ratings (every twenty minutes) by forward-filling each rating to all subsequent measurement timestamps. While this allowed us to utilize the complete measurement stream, it blurred the temporal relationship between inputs and the actual health states we aimed to predict. Forward-filled labels meant that measurements early in a window carried information about ratings far in the future, leading to significant label leakage.

Building on this aligned dataset, we deployed a DummyRegressor that predicted the global mean rating as our first benchmark. Remarkably, this simple model outperformed early regression efforts. Upon closer analysis, several factors emerged to explain its success. The forward-filling approach artificially boosted performance by matching stale labels rather than forecasting true future values. Additionally, pervasive missing readings and noisy spikes in the sensor data skewed model training, yet the mean predictor remained unaffected by these artifacts. The health ratings themselves exhibited minimal variance, clustering tightly around high scores, so there was little deviation for more complex models to capture. We also faced a sparse feature set—only six measurement channels to predict up to fifteen targets—and strong multicollinearity among those channels further reduced the information content available for regression. Under these conditions, a constant prediction of the mean naturally minimized error more effectively than any model attempting to learn weak or noisy signals.

This experience clearly demonstrated that naive alignment and basic feature sets were insufficient for genuine forecasting, highlighting the need for more robust temporal aggregation, enhanced preprocessing, and richer feature engineering.

Current Approach

Inspired by the existing MATLAB system's methodology, we examined its proprietary algorithms and discovered that they compute health ratings directly from raw voltage signals using summary statistics rather than pre-aggregated measurements. Motivated by these insights, we shifted away from forward-filling ratings and instead aligned our inputs to the twenty-minute rating windows through interval-based summaries.

Initially, a custom Python script (`model/archive/feature_eng_rar.py`) parsed raw `.Single` waveform files under `data/voltage/extracted/`. This script read each file's 4-byte float32 header to determine sampling rate, loaded the signal array, and calculated nine time- and

frequency-domain measurements (e.g., `velocity_rms`, `crest_factor`, `kurtosis_opt`) to help us understand the voltage characteristics and guide feature design. Table 1 lists the mathematical expressions for computing each measurement channel, which serve as the basis for interval summaries.

Once preprocessed JSON files became available, we rewrote the pipeline in `feature_engineer.py` to parse these JSON structures directly. The JSON-based workflow reads each file's six measurement channels at their original sampling frequency, applies quality checks, fills missing fields from metadata, and aggregates statistical features—such as mean, variance, maximum, and minimum—over each twenty-minute interval. This unified DataFrame is saved as `metrics_all.csv`, streamlining feature engineering by eliminating repeated raw-binary parsing and improving robustness to file-format changes.

Table 1: Formulas for each measurement used in feature engineering

Measurement	Formula
velocity_rms	$\sqrt{\frac{1}{N}\sum_{i=1}^{N} v_i^2}$
crest_factor	$\frac{\max_i \lvert v_i \rvert}{\sqrt{\frac{1}{N}\sum_{i=1}^{N} v_i^2}}$
kurtosis_opt	$\frac{\frac{1}{N}\sum_{i=1}^{N}(v_i-\bar{v})^4}{(\frac{1}{N}\sum_{i=1}^{N}(v_i-\bar{v})^2)^2}$
peak_value_opt	$\max_i$
rms_1_10khz	$\sqrt{\sum_{f=1}^{10\text{Hz}} \frac{\lvert X(f)\rvert^2}{N}}$
rms_10_25khz	$\sqrt{\sum_{f=10}^{25\text{Hz}} \frac{\lvert X(f)\rvert^2}{N}}$
peak_10_1000hz	$\max_{f\in[10,1000\text{Hz}]}$

Adopting these interval summaries yielded much better results. Tree-based models—specifically Random Forest and XGBoost—emerged as our top performers. Their ability to capture non-linear interactions, handle multicollinearity, and exploit hierarchical splits allowed them to leverage the richer, interval-based feature set effectively and distinguish subtle variations in equipment health.

RNN Implementation

To capture temporal dependencies in the measurements, we implemented a recurrent neural network (RNN) using Keras and wrapped it in a scikit-learn–compatible estimator (`RNNRegressor`). This allowed us to integrate sequential modeling seamlessly into our existing cross-validation framework.

First, we reshaped each twenty-minute interval's aggregated measurements into a sequence of five-second steps. The resulting input tensor had shape `(samples, timesteps, features)`,

where `timesteps=240` and `features=6` measurement channels. Missing steps within a window were forward-filled, and sequences shorter than 240 were zero-padded to maintain consistent input dimensions.

The model architecture consisted of:

- **Input Layer:** Accepts the (240, 6) sequence per sample.
- **SimpleRNN Layer:** 50 units with ReLU activation to learn temporal patterns across measurement steps.
- **Dropout Layer:** Rate 0.2 to mitigate overfitting given limited data diversity.
- **Dense Output Layer:** Fully connected layer with units equal to the number of target ratings (12–15), using linear activation for regression.

We trained the RNN for 10 epochs with a batch size of 32, using the Adam optimizer (learning rate 0.001) and mean squared error loss. Early stopping monitored validation loss with a patience of 3 epochs to prevent overfitting.

Cross-validation followed a 5-fold 'TimeSeriesSplit', identical to our tree-based models. For each fold, we recorded RMSE and R^2 scores and averaged them across folds. The RNN achieved moderate improvements over linear models in capturing short-term temporal effects but still lagged behind tree-based approaches due to the coarse interval aggregation and limited sequence length.

From our RNN experiments, the model learned small oscillations in the vibration signals. It found patterns that static regressors could not see. We added dropout and early stopping to reduce overfitting. Even so, the RNN sometimes learned noise spikes instead of true signals. Training the RNN took much longer. Each fold ran about five times slower than the Random Forest model. In several cross-validation folds, the R^2 scores were very large and negative—often below –10. This result means that simply predicting the mean would have been more accurate than the RNN under our interval approach. Overall, the RNN shows that sequence models can capture temporal information. However, its complexity, training time, and sensitivity to our data make tree-based models more reliable for this task.

Deployment Pipeline

To manage the high-frequency data volume and large file-size challenges, we built an AWS-based infrastructure. All raw waveform archives and JSON measurement files are stored in Amazon S3, which bypasses GitHub's 100 MB file limit and provides scalable, centralized access. For compute, we provisioned EC2 instances sized for batch processing of these time-series datasets. The EC2 hosts pull data from S3, run feature-engineering scripts in parallel, and write summarized CSVs back to S3. This design decouples storage from compute, enables horizontal scaling, and ensures that large files never reside in version control.

Model and dashboard updates also run on EC2, where scheduled jobs fetch the latest interval summaries from S3, apply trained models, and publish results to a web dashboard. By separating data storage and compute, and leveraging S3 and EC2, we handled both the data volume and file-size constraints without sacrificing performance or collaboration.

Conclusion

In this project, we developed an interpretable machine learning pipeline for industrial equipment health monitoring. We started with a naive forward-filling approach and a mean baseline, which highlighted data challenges such as label leakage, low diversity, and multicollinearity. By drawing inspiration from the MATLAB system and shifting to interval-based summaries, we improved predictive performance significantly. Tree-based models—Random Forest and XGBoost—proved most effective at capturing non-linear interactions and handling multicollinearity, outperforming both linear regressors and sequence models.

We also explored an RNN to capture temporal dependencies, but its complexity, long training times, and negative R^2 results under our aggregation scheme demonstrated that static, interval-based methods are more reliable given the data characteristics. Finally, we established an AWS-based pipeline using S3 for storage and EC2 for compute, addressing high-frequency data volume and file-size constraints without overloading version control.