Group Lasso Transformer SHAP Experiment Report

Author: Sheng Wang a1903948
Team: RAIL-PG-2

1. Introduction

XAI Techniques provide the Explanations to help us understand model predictions. For features, they show how each one pushed the prediction up or down. This is useful for debugging models, giving stakeholders a clear picture of how the system works, and satisfying compliance requirements.

But a lot of important questions can't be answered just by saying "how much did each feature contribute?" To cover more ground, I use a well-founded approach called SHAP (Shapley Additive Explanations), which generalizes game-theoretic Shapley values to model explanations.

For a safety-critical task like predicting rail fractures, to achieve AUC-PR or F1 score alone isn't enough. We also need to know:

- Why does the model think a specific track will break?
- Which sensor readings contribute the most to that prediction?
- Do features interact with each other in meaningful ways?
- How do we explain the model's decision to maintenance teams?

SHAP is designed to answer exactly these kinds of questions. It uses the Shapley value idea from cooperative game theory to break down each individual prediction into reliable, per-feature contributions, giving us a transparent view of what drove the model's output.

2. How to Implement SHAP Technique in Group Lasso Transformer Model

The SHAP analysis follows the code's Step 1–Step 3 as Figure 1 shows, which includes Prepare data, Compute SHAP value and visualization.

```
# Step 1: Prepare data
background_data, explain_data, explain_labels, exp_indices = prepare_shap_data(
    test_dataset,
    n_background=n_background,
    n_explain=n_explain
)
# Step 2: Compute SHAP values
shap_values, base_value, explainer = compute_shap_values(
    model,
    background_data,
    explain_data,
    feature_names,
    nsamples=nsamples,
    device=device
)
# Step 3: Visualization
feature_importance_df = visualize_shap_results(
    shap_values,
    explain_data,
    explain_labels,
    feature_names,
    base_value,
)
```

Figure1. SHAP Analysis Workflow

2.1 Data Preparation and Sampling Strategy

SHAP's KernelExplainer works by randomly masking features and watching how the model's prediction changes, so it can estimate each feature's marginal contribution. To make this work, I design two datasets:

- Background data: Used to estimate the model's baseline (the "expected prediction").
- Explain data: The specific samples we want to interpret.

The quality of these two sets directly affects the reliability of the SHAP values. In this implementation, I set the defaults to:

- N_background = 100 (size of the background set)
- N_explain = 50 (size of the explain set)

After fixing the sizes, I use random sampling so every sample has an equal chance of being selected (see the "Random sampling" part in Figure 2). I also make sure the two sets are disjoint—no sample appears in both the background and explanation sets.

```python
def prepare_shap_data(dataset, n_background=100, n_explain=50, seed=42):
    """
    Sample data from the dataset for SHAP analysis

    Returns:
        background_data: (n_background, T, F) - background data (for estimating expectation)
        explain_data: (n_explain, T, F) - samples to explain
        explain_labels: (n_explain,) - ground-truth labels
        explain_indices: (n_explain,) - sample indices
    """
    np.random.seed(seed)

    total_samples = len(dataset)

    # Ensure the sampled size does not exceed the dataset size
    n_background = min(n_background, total_samples // 2)
    n_explain = min(n_explain, total_samples - n_background)

    # Random sampling
    indices = np.random.permutation(total_samples)
    bg_indices = indices[:n_background]
    exp_indices = indices[n_background:n_background+n_explain]

    # Extract data
    background_data = np.stack([dataset.X[i].numpy() for i in bg_indices])
    explain_data = np.stack([dataset.X[i].numpy() for i in exp_indices])
    explain_labels = dataset.y[exp_indices].numpy()

    print(f"\n=== SHAP Data Preparation ===")
    print(f"Background data shape: {background_data.shape}")
    print(f"Explain data shape: {explain_data.shape}")
    print(f"Explain labels: {np.bincount(explain_labels.astype(int))} (neg/pos)")

    return background_data, explain_data, explain_labels, exp_indices
```

Figure2. SHAP Data Preparation Processes

2.2 Configuring SHAP: KernelExplainer and Parameter Choices

The reason I choose SHAP is that KernelExplainer is a good fit here because it's truly model-agnostic—it treats the predictor as a black box and doesn't depend on any internal architecture.

It estimates Shapley values by repeatedly masking different feature combinations and observing how the prediction shifts, which gives a faithful picture of each feature's marginal effect. On top of that, it's grounded in solid theory, offering guarantees like local accuracy and baseline consistency, so the explanations are not just intuitive but principled.

The SHAP workflow has two key stages: initializing the KernelExplainer and calling explainer.shap_values. Below, I briefly describe how I set up the explainer and how I configured the parameters for the SHAP value computation.

KernelExplainer initialization

- model_wrapper
  The wrapped prediction function. It takes flattened inputs and returns probabilities.
- bg_flat
  Background data that approximates the real data distribution and is used to estimate the baseline/expected prediction. The more representative it is, the more stable the SHAP values.
- link="identity"
  Maps model outputs into the explanation space. Since my wrapper already outputs probabilities, identity is appropriate (no transformation). If the model returned logits, logit would be a better choice.

Parameters for explainer.shap_values(...)

- exp_flat
  The instances to explain. Ideally, they should follow roughly the same distribution as the training data.
- nsamples
  The number of Monte Carlo samples for the KernelSHAP approximation. Larger values improve fidelity but increase runtime.
- l1_reg="num_features(k)"
  A sparsity constraint that highlights only the top 10 features per instance, making explanations easier to read.

```python
# Create wrapper
model_wrapper = TorchModelForSHAP(model, device=device)

# Flatten to 2D for SHAP (the time dimension will be flattened)
N_bg, T, F = background_data.shape
N_exp = explain_data.shape[0]

bg_flat = background_data.reshape(N_bg, T * F)
exp_flat = explain_data.reshape(N_exp, T * F)

print(f"Flattened background shape: {bg_flat.shape}")
print(f"Flattened explain shape: {exp_flat.shape}")
print(f"Computing with nsamples={nsamples}...")

# Create feature names for the flattened data
flat_feature_names = []
for t in range(T):
    for feat in feature_names:
        flat_feature_names.append(f"{feat}_t{t}")

# Use KernelExplainer
explainer = shap.KernelExplainer(
    model_wrapper,
    bg_flat,
    link="identity"  # because model_wrapper already outputs probabilities
)

# Compute SHAP values (this step can be slow)
print("This may take a few minutes...")
shap_values = explainer.shap_values(
    exp_flat,
    nsamples=nsamples,
    l1_reg="num_features(10)"  # automatically choose a sparse solution
)
```

Figure3. KernelExplainer and Parameter Choices

I use KernelExplainer(model_wrapper, bg_flat, link="identity") to compute SHAP values in probability space, with bg_flat providing the reference distribution for the expected value. For exp_flat, I set nsamples = 100–200 to balance accuracy and runtime, and apply l1_reg="num_features(10)" so each instance's explanation focuses on its main drivers. In high-dimensional time-series settings, these hyperparameters have a big impact on both cost and stability, so I fix random seeds and run sampling checks to verify the consistency of the results.

2.3 Visualization and Interpretation

For the interpretation, I implemented three kinds of visualizations:

- 1. Feature Importance Ranking

This helps quickly identify the most important features overall. For the rail-break prediction task, it suggests that things like "track profile" and "noise level" might be the key indicators.

- 2. Summary Plot (Bee-swarm)

The SHAP summary plot not only shows the global distribution of feature importance, but also makes it clear how each feature pushes predictions toward the positive or negative class and by how much.

- 3. Waterfall Plot

Waterfall plots show how each prediction can be broken down into a step-by-step accumulation of feature contributions.

Regarding Visualization, more detailed explanations are in Section 3.

3. Visualisation to Enhance Explanation

3.1 Feature Importance Ranking

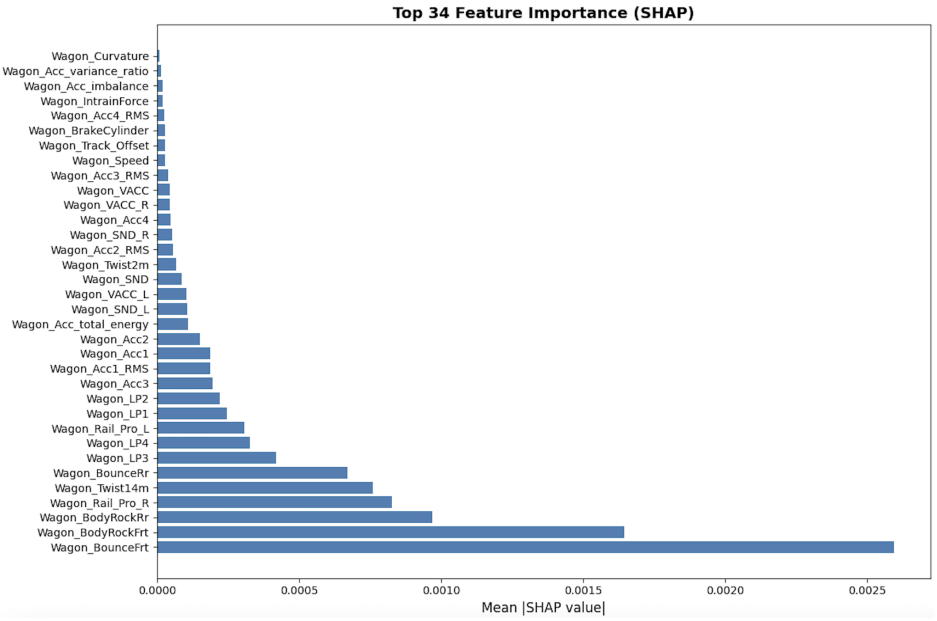


Figure4. Feature Importance Ranking

Top feature importance is dominated by motion sensors: Wagon_BounceFrt, Wagon_BodyRockFrt, and Wagon_BodyRockRr show highest SHAP values. Track geometry features (Rail_Pro_R, Rail_Pro_L) and lateral position (LP3, LP4) rank second tier. Most acceleration and noise metrics contribute minimally, suggesting redundancy among vibration channels.

3.2 Summary Plot

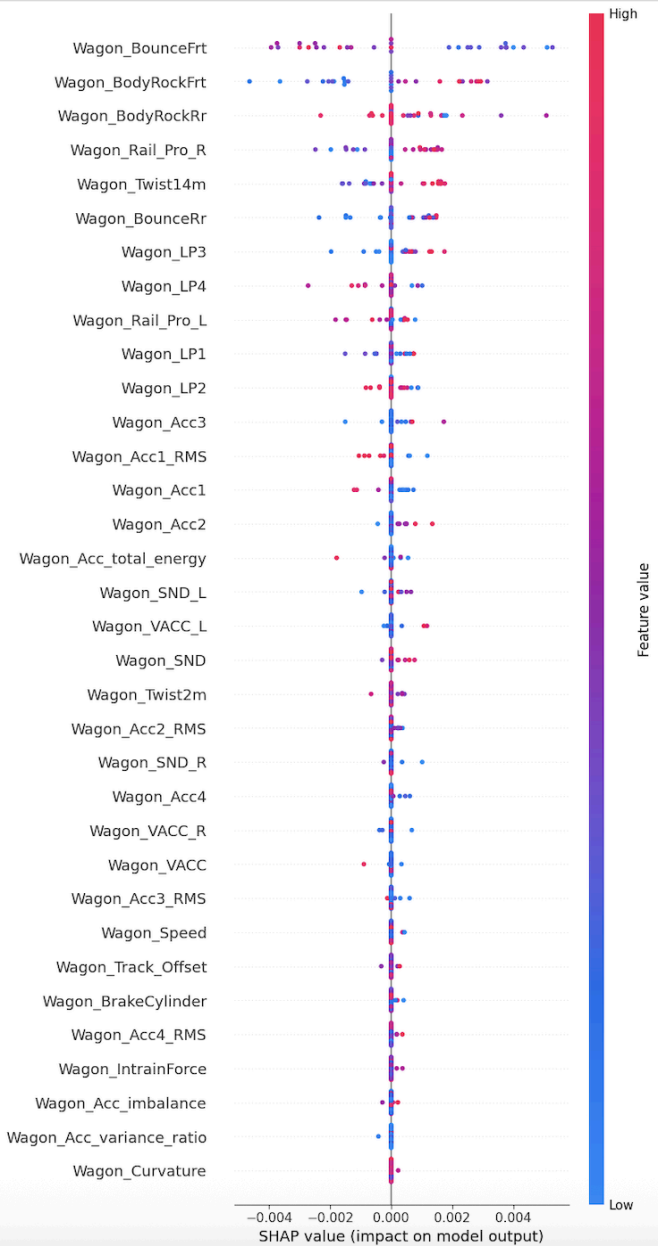


Figure5. Features Summary Plot

From this summary plot, the model learns very different patterns for different features.

Wagon_BounceFrt (front bounce) is unusual: red points (high bounce) and blue points (low bounce) are scattered on both sides with no clear trend. This means bounce level doesn't have a consistent effect on failure risk—the model did not learn a simple relationship there.

By contrast, **Wagon_Rail_Pro_R (rail profile)** is straightforward: red points cluster on the right and blue points on the left, showing a clear monotonic pattern. The model effectively learned “higher rail-profile values → higher failure likelihood.”

This highlights an important point: suspension signals are complex and can’t be reduced to “high = failure, low = normal,” whereas track geometry signals are direct and easier for the model to capture. For maintenance, we should prioritize geometry anomalies, while evaluating suspension bounce requires more domain expertise.

3.3 Waterfall Plot

- Positive Sample

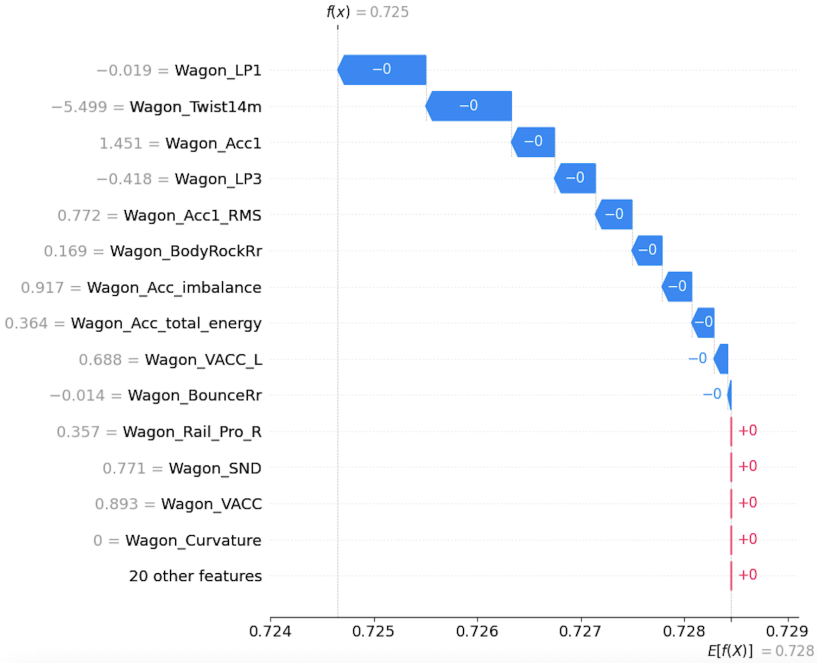


Figure6. Positive Sample Waterfall Plot

For this positive sample (predicted failure: 0.729), the arrow directions reveal the model's logic. Blue arrows pointing left mean those features reduce failure risk—Wagon_LP1 (-0.019) and Wagon_Twist14m (-5.499) have low values (-0.019, -5.499), providing strong protection. Red arrows pointing right mean those features increase failure risk. Wagon_Rail_Pro_R, Wagon_SND, Wagon_VACC show high values (0.357, 0.771, 0.893), pushing toward failure prediction. These track geometry and vibration anomalies outweigh the protective low-value features, causing the model to predict this track as faulty. The multiple red signals converge to override the blue protection signals.

- Negative Sample

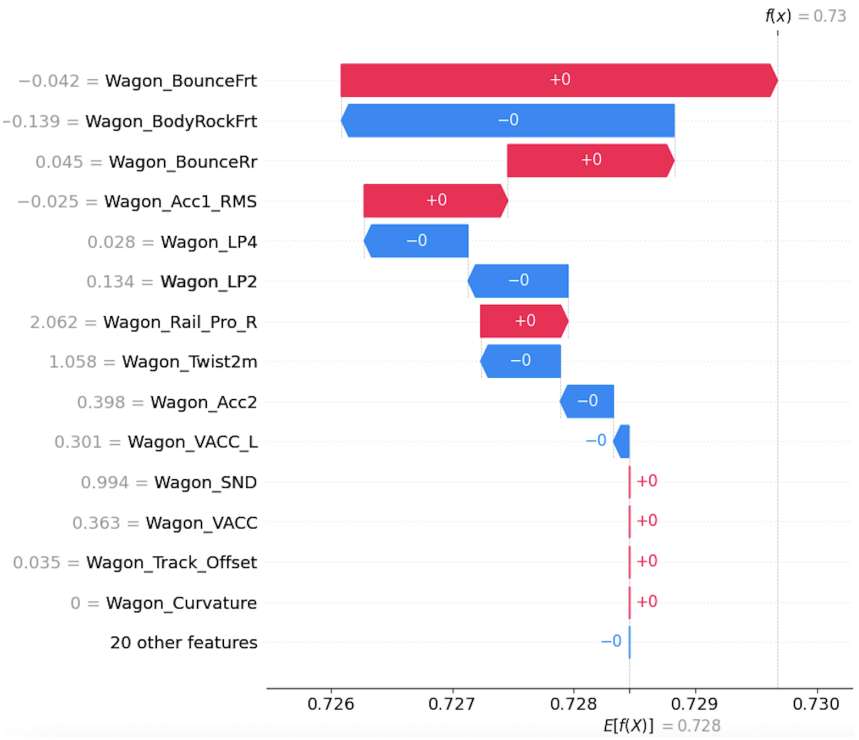


Figure7. Negative Sample Waterfall Plot

For this negative sample (predicted normal: 0.728), the model shows mixed but weaker signals compared to true failures. The dominant red arrow is Wagon_BounceFrt (−0.042), suggesting some front-wheel bounce pushing toward failure, but its contribution is marginal. Blue arrows indicate protective factors: Wagon_BodyRockFrt (−0.139), Wagon_LP2 (0.134), and Wagon_Twist2m (1.058) have abnormal values yet reduce failure risk. Critically, track geometry features Wagon_Rail_Pro_R (2.062) show high values but contribute weakly (+0). This weak geometric signal, combined with protective factors, keeps the prediction just below the failure threshold at 0.728—indicating the model correctly identifies this as a borderline normal track with some anomalies but insufficient fault evidence.

4. LeaderBoard Result

According to Figure4, I choose top5 , top10, top15, top20 as selected features to re-execute the model to compare the prediction results.

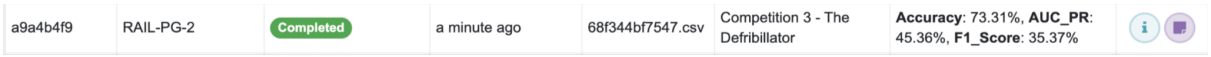
Top5 important features:

84090584	RAIL-PG-2	Completed	a minute ago	68f335df5529.csv	Competition 3 - The Defribillator	**Accuracy**: 38.58%, **AUC_PR**: 26.58%, **F1_Score**: 41.53%

Top10 important features:

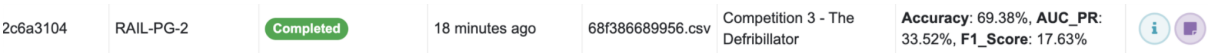
f389b93c	RAIL-PG-2	Completed	a minute ago	68f3292e3463.csv	Competition 3 - The Defribillator	**Accuracy**: 64.98%, **AUC_PR**: 46.00%, **F1_Score**: 52.42%

Top15 important features:



a9a4b4f9	RAIL-PG-2	Completed	a minute ago	68f344bf7547.csv	Competition 3 - The Defribillator	**Accuracy**: 73.31%, **AUC_PR**: 45.36%, **F1_Score**: 35.37%

Top20 important features:



2c6a3104	RAIL-PG-2	Completed	18 minutes ago	68f386689956.csv	Competition 3 - The Defribillator	**Accuracy**: 69.38%, **AUC_PR**: 33.52%, **F1_Score**: 17.63%

5. Next plan - Model Improvement Direction

The SHAP visualizations provide more information about the feature reasoning. The **Summary Plot** shows that the model learns multiple patterns, and the **Waterfall Plots** for positive/negative samples reveal, at a finer granularity, how individual features push the prediction up or down.

Through the visualizations, I can do more than rank features. In this sprint, I only did a coarse comparison using top-5/10/15/20 features from global importance. Next, I plan to **drive feature selection by the patterns seen in the Summary and Waterfall plots**—e.g., prioritize features with clear, stable monotonic effects, or keep interaction-rich signals that consistently move predictions. I'll then track metric deltas per branch (AUC-PR, F1, etc.) to see whether these targeted selections actually lift the scores.