

# Group Lasso Transformer: Fine-tuning and Optimization Experiments

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

## 1.Introduction

In Sprint 2, I completed the baseline code based on the Group Lasso framework, including training, validation, and testing, and submitted it to the leaderboard. The initial F1 score was 51.43%.

In Sprint 3, according to the requirements of User Story 3, I aimed to optimize the F1 score to above 55%. Therefore, my main focus in Sprint 3 was to fine-tune the model and improve the F1 score to 55.33% in accordance with the requirements of User Story 3.

In the following sections, I will continue to follow the fine-tuning plan mentioned in the Sprint 2 report, conducting experiments on model parameters and different strategies, and providing a summary and explanation of the results.

## 2.Experiments

Before conducting all the experiments, I first fixed a baseline version of the model parameters.

- Feature Grouping and Regularization Penalties:

Feature Grouping: No grouping, each feature treated as an individual group.

Regularization Penalties: No penalty applied,  $L1 = L2 = 0$ .

- Clipping Strategy Comparison:

Percentile = (0.5, 99.5).

- Handling Imbalanced Data:

None.

- Model Architecture and Hyperparameter:

### 1. **TransformerEncoderLayer**

```
encoder_layer = nn.TransformerEncoderLayer(  
    d_model=96,
```

```

        nhead=4,
        dim_feedforward=192,
        dropout=0.1,
        activation='relu',
        batch_first=True,
        norm_first=True
    )

```

## 2. Transformer

```

self.transformer = nn.TransformerEncoder(
    encoder_layer,
    num_layers=1
)

```

## 3. Output\_head

```

d_model=96
dropout=0.1

output_head = nn.Sequential(
    nn.Linear(d_model, d_model // 2),
    nn.LayerNorm(d_model // 2),
    nn.GELU(),
    nn.Dropout(dropout),
    nn.Linear(d_model // 2, 1)
)

```

Then, based on the controlled variable method, adjustments and comparisons will be made to analyze the impact of different parameters and strategies on the model and results.

### 2.1 Time Window Aggregation

The first experiment conducted was a comparison of different time windows, as this is an important step during the preparation of the training and prediction datasets.

For a 20-meter section of railway, different time windows are considered based on the historical data, which is then provided to the model for subsequent aggregation. So this test is the first step in all experiments and is the most important one.

I used the basic model without grouping features, calculating all features without modification, and only adjusted the time windows. The resulting data comparison table is shown below. These data were collected after submission to the leaderboard.

Training Data Time Window	Prediction Data Time Window (-7days ~ record date)	Prediction Data Time Window (-30days ~ record date)	Prediction Data Time Window (-30days ~ record datedays ~ 30)
(-30days ~ record date)	Accuracy: 66.21% F1: 34.62%	Accuracy: 66.00% F1: 51.43%	Accuracy: 66.53% F1: 42.60%
(-30days ~ break_day)	Accuracy: 56.84% F1: 46.61%	Accuracy: 58.42% F1: 50.06%	Accuracy: 51.16% F1: 43.00%

**Figure1. Performance Comparison Across Different Training and Prediction Time Windows**

Based on the collected data, it appears that larger time windows introduce more noise from historical data. Given the data performance, I have chosen the time window of -30 days ~ record date as the baseline for further fine-tuning.

## 2.2 Feature Grouping Selection

Based on the principles of Group Lasso, I divided the features into the following groups according to the semantics of the IF document:

```
prefix_dict_with_group = {

    # Twist
    "Wagon_Twist": ["Wagon_Twist14m", "Wagon_Twist2m"],

    # Bounce
    "Wagon_Bounce": ["Wagon_BounceFrt", "Wagon_BounceRr"],

    # Body rock
    "Wagon_BodyRock": ["Wagon_BodyRockFrt", "Wagon_BodyRockRr"],

    # Longitudinal position (LP)
    "Wagon_LP": ["Wagon_LP1", "Wagon_LP2", "Wagon_LP3", "Wagon_LP4"],

    # Speed / Brake / Traction
    "Wagon_Speed":
["Wagon_Speed", "Wagon_BrakeCylinder", "Wagon_IntrainForce"],
```

```

# ACC
"Wagon_Acc": [
    "Wagon_Acc1",
    "Wagon_Acc2",
    "Wagon_Acc3",
    "Wagon_Acc4",
    "Wagon_Acc1_RMS",
    "Wagon_Acc2_RMS",
    "Wagon_Acc3_RMS",
    "Wagon_Acc4_RMS"
],

# Track geometry
"Wagon_Track geometry":
["Wagon_Rail_Pro_L", "Wagon_Rail_Pro_R", "Wagon_Curvature", "Wagon_Track_Offset"
],

# Noise / Vibration
"Wagon_SND": ["Wagon_SND", "Wagon_SND_L", "Wagon_SND_R"],
"Wagon_VACC": ["Wagon_VACC", "Wagon_VACC_L", "Wagon_VACC_R"]
}

```

In feature selection, during the Sprint 3, I employed a gating technique. The gating technique first runs the model on all available features and ranks them based on their importance. Then, I selected the **top 4, top 8, and top 10 features** based on the importance ranking and compared the metrics. Ultimately, I chose the top 8 features as the basis for all subsequent experiments.

The gating technique is a method used to assess the relevance of each feature in a model by assigning an importance score to each one. The model first trains on the entire set of features, then evaluates and ranks each feature based on how much it contributes to the model's performance.

Top important features:	
1. Wagon_Acc1	importance: 0.7164
2. Wagon_Acc2	importance: 0.7164
3. Wagon_Acc3	importance: 0.7164
4. Wagon_Acc4	importance: 0.7164
5. Wagon_Acc1_RMS	importance: 0.7164
6. Wagon_Acc2_RMS	importance: 0.7164
7. Wagon_Acc3_RMS	importance: 0.7164
8. Wagon_Acc4_RMS	importance: 0.7164
9. Wagon_SND	importance: 0.6530
10. Wagon_SND_L	importance: 0.6530
11. Wagon_SND_R	importance: 0.6530
12. Wagon_Rail_Pro_L	importance: 0.6116
13. Wagon_Rail_Pro_R	importance: 0.6116
14. Wagon_Curvature	importance: 0.6116
15. Wagon_Track_Offset	importance: 0.6116
16. Wagon_LP1	importance: 0.6039
17. Wagon_LP2	importance: 0.6039
18. Wagon_LP3	importance: 0.6039
19. Wagon_LP4	importance: 0.6039
20. Wagon_Speed	importance: 0.5721
21. Wagon_BrakeCylinder	importance: 0.5721
22. Wagon_IntrainForce	importance: 0.5721
23. Wagon_VACC	importance: 0.5230
24. Wagon_VACC_L	importance: 0.5230
25. Wagon_VACC_R	importance: 0.5230
26. Wagon_BounceFrt	importance: 0.5097
27. Wagon_BounceRr	importance: 0.5097
28. Wagon_Twist14m	importance: 0.4815
29. Wagon_Twist2m	importance: 0.4815
30. Wagon_BodyRockFrt	importance: 0.4264
31. Wagon_BodyRockRr	importance: 0.4264

Figure2. Features Importance Score List

The Figure below shows the results after selecting **top 4**, **top 8**, and **top 10** features and performing model predictions:

Top n	4	8	10	20
Accuracy:	43.37%	49.89%	43.37%	54.53%
F1_Score:	43.96%	52.02%	40.09%	39.66%

Figure3. Performance Comparison Across Different Feature Selection Sizes

This result indicates that while more features can improve overall accuracy, they may not always contribute to a better F1 Score. Selecting an optimal number of features, like top 8, tends to provide the best balance between precision and recall, leading to a higher F1 Score.

2.3 Handling Imbalanced Data

Based on the results from section 2.2, after selecting the top 8 features and grouping them into a feature group, the F1 score performance was relatively better compared to other scenarios.

On top of selected features, I employed **WeightedRandomSampler** to adjust the distribution of positive and negative samples in each training batch, thereby addressing class imbalance during the sampling process. At the same time, **Focal Loss** and the **BCEWithLogitsLoss** method were utilized to adjust the loss weights during the calculation of the loss function, allowing for greater emphasis on harder-to-classify examples. The detailed parameters and results are provided in the table below:

- Focal Loss

alpha	gamma	$\lambda_2$	Metric
0.75	1.0	0	Accuracy: 48.11%, F1_Score: 52.55%
0.75	1.5	0	Accuracy: 47.89%, F1_Score: 50.35%
0.9	2.0	0	Accuracy: 49.26%, F1_Score: 51.21%

**Figure4. Performance Comparison of Focal Loss with Different Alpha and Gamma Values**

Based on the data performance, with **alpha=0.75** and **gamma=1.0** showing good results, I further adjusted the parameters of the L2 penalty factor and observed the outcomes.

alpha	gamma	$\lambda_2$	Metric
0.75	1.0	0	Accuracy: 48.11%, F1_Score: 52.55%
0.75	1.0	0.001	Accuracy: 50.95%, F1_Score: 51.66%
0.75	1.0	0.0001	Accuracy: 47.89%, F1_Score: 53.64%
0.75	1.0	0.0005	Accuracy: 49.26%, F1_Score: 48.95%

**Figure5. Performance Comparison After Adjusting L2 Penalty Factor with Fixed Alpha and Gamma**

- Sample Loader

After adjusting Focal Loss and seeing improvements in the data, I decided to **combine it with WeightedRandomSampler** to further investigate how different sampler strategies affect imbalanced data. Below is the code for the sampler.

```

y_np = dataset.y.numpy().astype(int)
N_pos = int((y_np == 1).sum())
N_neg = int((y_np == 0).sum())
N_total = len(y_np)

if N_pos == 0:
    raise ValueError("There are no positive samples in the dataset, unable to
perform resampling.")

# Calculate the weight factor alpha for positive samples
alpha = (target_pos_fraction * N_neg) / ((1.0 - target_pos_fraction) * N_pos)

# Assign weights to each sample
sample_weights = np.where(y_np == 1, alpha, 1.0).astype(np.float64)

# Number of samples
if num_samples is None:
    num_samples = N_total

# Construct the WeightedRandomSampler
generator = torch.Generator()
generator.manual_seed(seed)
sampler = WeightedRandomSampler(
    weights=sample_weights,
    num_samples=num_samples,
    replacement=replacement,
    generator=generator
)

```

This code defines a function called `make_balanced_sampler` that creates a `WeightedRandomSampler` to balance the classes (positive and negative samples) in each batch during training. This approach is especially useful for imbalanced datasets where one class (e.g., positive) is much less frequent than the other (negative).

I tuned the value of `target_pos_fraction` and recorded the results:

Target_Pos_Fraction	50%	60%	70%	75%	80%	90%
Accuracy:	47.89%	55.58%	54.11%	55.68%	61.79%	55.05%
F1_Score:	53.64%	52.69%	53.52%	53.89%	48.36%	48.74%

**Figure6. Performance Comparison of Accuracy and F1 Score with Different Target Positive Fractions**

- BCEWithLogitsLoss

Although I have already conducted experiments with Focal Loss, which is designed to address class imbalance by focusing more on hard-to-classify samples, it is still important to verify the effectiveness of BCEWithLogitsLoss with the pos\_weight parameter.

Focal Loss adjusts the loss dynamically based on the difficulty of each sample, **BCEWithLogitsLoss with pos\_weight offers a simpler approach by explicitly weighting the loss for positive samples, especially when dealing with imbalanced datasets.**

By tuning pos\_weight, I can explore whether this more straightforward adjustment provides any additional benefits in handling class imbalance, or if Focal Loss remains the better approach. This comparative testing helps ensure that I'm using the most effective loss function for the given dataset and task.

Pos_Weight	0.8	1	1.4	2
Accuracy:	56.63%	55.68%	62.63%	54.95%
F1_Score:	54.92%	54.29%	50.35%	47.16%

**Figure7. Performance Comparison of BCEWithLogitsLoss with Different Pos\_Weight Values**

## 2.4 Model hyperparameters tuning

In hyperparameter tuning, the primary goal is to avoid overfitting and ensure the model learns meaningful patterns. To achieve this, I used the **AdamW optimizer**, which includes key components such as **Learning Rate**, **Betas**, and **Weight Decay**. The optimizer is defined as follows:

```
optimizer = torch.optim.AdamW(
    [{"params": decay, "weight_decay": 1e-5}, {"params": no_decay,
"weight_decay": 0.0}],
    lr=lr,
    betas=(0.9, 0.99)
)
```

I also used a learning rate scheduler to adjust the learning rate over time:

```
from torch.optim.lr_scheduler import LambdaLR

def lr_lambda(current_step):
    if current_step < lr_warmup_steps:
        return float(current_step) / float(max(1, lr_warmup_steps))
    progress = (current_step - lr_warmup_steps) / float(max(1, total_steps -
lr_warmup_steps))
    return max(0.0, 1.0 - progress)

scheduler = LambdaLR(optimizer, lr_lambda)
```

#### Key Concepts:

- **Learning Rate:** Controls the step size for updating model parameters. A higher rate leads to larger steps, while a lower rate results in smaller steps.
- **Betas:** These parameters control the moving averages of the gradient and squared gradient, helping stabilize the optimization process.
- **Weight Decay:** Used as a regularization term to prevent large weights, helping reduce overfitting by penalizing overly large model parameters.

Based on the promising performance of BCEWithLogitsLoss with the parameters from Section 2.3, and as observed in Figure X from Section 2.3, the regularization penalty had a significant impact on the F1 score.

I adjusted the Weight Decay parameter and fine-tuned the BCEWithLogitsLoss by setting Pos\_Weight = 0.8. The results are recorded below.

weight_decay	1e-5	1e-4	5 * 1e-4	1e-3	5 * 1e-2
Accuracy:	55.68%	57.68%	54.32%	53.58%	59.79%
F1_Score:	54.29%	55.33%	49.88%	52.63%	48.79%

**Figure8. Performance Comparison with Different Weight Decay Values in BCEWithLogitsLoss**

## 2.5 Feature Engineering

Based on the **performance of Focal Loss**, I selected the best parameter set, **alpha=0.75**, **gamma=1.0**, and  **$\lambda_2=0.0001$** , and then conducted feature engineering testing and validation, focusing on the following aspects:

- Different Feature Aggregation Methods:

In the comparative experiment, the following 8 features were used:

Wagon\_Acc1, Wagon\_Acc2, Wagon\_Acc3, Wagon\_Acc4, Wagon\_Acc1\_RMS, Wagon\_Acc2\_RMS, Wagon\_Acc3\_RMS, Wagon\_Acc4\_RMS.

These features were tested using different aggregation methods: avg, P\_50, P\_95, and P\_95 - P\_50 for comparison. The results are as follows:

Method	Accuracy	F1 Score
avg	47.89%	53.64%
P_50	49.05%	51.41%
P_95	48.95%	52.59%
P_95 - P_50	60.74%	42.88%

**Figure9. Performance Comparison of Different Feature Aggregation Methods**

- New Feature Fields:

For the top 8 features Wagon\_Acc1, Wagon\_Acc2, Wagon\_Acc3, Wagon\_Acc4, Wagon\_Acc1\_RMS, Wagon\_Acc2\_RMS, Wagon\_Acc3\_RMS, Wagon\_Acc4\_RMS, feature engineering was applied.

I initially selected Wagon\_Acc1 and Wagon\_Acc2 for the creation of derived fields and testing, as shown in the following code:

```
aggregated_df = (
    aggregated_df
```

```

.withColumn("Wagon_Acc1_div_Acc2",
            F.col("Wagon_Acc1") / (F.col("Wagon_Acc2") + F.lit(1e-6)))
.withColumn("Wagon_Acc1_sub_Acc2",
            F.abs(F.col("Wagon_Acc1") - F.col("Wagon_Acc2")))
)

```

Two new columns were added: **Wagon\_Acc1\_div\_Acc2** and **Wagon\_Acc1\_sub\_Acc2**, resulting in the following performance metrics:

Metric : Accuracy: 47.47%, F1 Score: 49.95%

This experiment explored different feature aggregation methods and derived features. The average aggregation method may suffice due to its simplicity. And the addition of **new derived features** led to a **decrease in F1 score**, likely due to these features acting as noise, which prevented the model from effectively capturing relevant information during training.

### 3.Results

Based on the first version of the basic implementation, model training and prediction were carried out, and the results were submitted to the leaderboard:




be55df2e	RAIL-PG-2	Completed	7 days ago	68d781805425.csv	Competition 2 - Senna	★ Accuracy: 64.32%, AUC_PR: 46.57%, F1_Score: 57.68%	 
----------	-----------	-----------	------------	------------------	-----------------------	--	---

Figure10. Model training scores



Leaderboard

My Dashboard

Leaderboard

Competition 2 - Senna

Live updates active

F1\_score




Rank		Team	Entry	F1_Score	Date
1		RAIL-PG-3	68dfb4889939.csv	84.44%	Oct 03, 21:04
2		RAIL-PG-2	68d781805425.csv	57.68%	Sep 27, 15:49
3		Overfit and Chill	68d99d632079.csv	54.00%	Sep 29, 06:12
4		RAIL-UG-2	68d67f3e7340.csv	52.26%	Sep 26, 21:27
5		Group 742	68cd1e534848.csv	51.68%	Sep 19, 18:43
6		AetherCodex	68cb7efd2178.csv	51.35%	Sep 18, 13:11

Figure11. Leaderboard results

## 4. Conclusion

In the model development and experimentation process, key factors such as **feature selection**, **handling imbalanced data**, and **loss penalties** were found to significantly affect model performance.

- **Feature Selection:** Selecting the top 8 features based on their importance was crucial, striking a balance between model complexity and performance.
- **Handling Imbalanced Data:** Addressing class imbalance using WeightedRandomSampler and applying BCEWithLogitsLoss with Pos\_Weight=0.8 provided better results compared to Focal Loss, improving performance, particularly for difficult-to-classify positive samples.
- **Loss Penalties:** The Weight Decay parameter was crucial for regularization, and after testing,  $1e-4$  was chosen as the optimal value for decay to prevent overfitting and improve generalization.

These tuning strategies led to improvements in both accuracy and F1 score, ensuring the model is well-prepared for further testing and refinement.

## 5. Next plan

Continue to optimize the model and explore ways to further improve the F1 score. As per the requirements of User Story 4, the goal is to raise it to 60%.

Additionally, greater attention will be given to AUC-PR, as it measures performance across different thresholds and is more valuable for imbalanced datasets.