SAPIENZA
UNIVERSITÀ DI ROMA

ML to predict time-to-failure of lab earthquakes from acoustic emissions

Andrea Gentilini (2043590)
Leo Vincenzo Petrarca (2087113)

Rome, December 17, 2024

Key topics and main questions

SAPIENZA
UNIVERSITÀ DI ROMA

01
Data availability
What were the available data?

02
Data handling
How did we handle raw data?

03
Data pre-processing
How did we build the dataset for training?

04
Features engineering
What were the key features for the learning?

05
Model and Training
What model was used for prediction and how did we handle parameters tuning?

06
Performance and conclusions
How does the model perform? Holes in the dataset and cutting it.

Project overview

Why?

Traditional statistical methods do not seem to work in accurately predicting seismic events in a short-term window. Conversely, empirical evidence suggests that **Machine Learning can** be effectively used to **predict time-to-failure** in laboratory earthquakes.

Our motivating idea

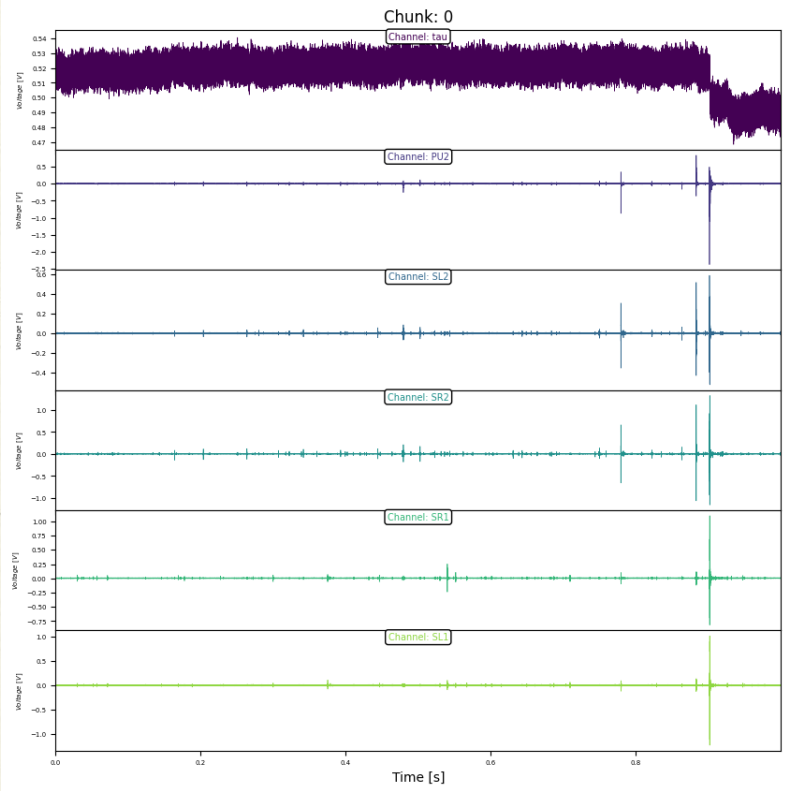
2019: Los Alamos National Laboratory hosted a **Kaggle competition**.
Objective: predict the time remaining before laboratory earthquakes occur, based on real-time seismic acoustic data. This was a **significant open question** at the time, attracting interest from numerous research institutions.

Our goal: reproduce the state-of-the-art model at that time and then develop our own, learning from new data (La Sapienza Rock Mechanics Laboratory) to achieve a better performance.

Data

Data from La Sapienza Rock Mechanics Laboratory are acoustic emissions collected using the TiePie oscilloscope.

- 6 channels:
  Shear Stress, PU2, SL2, SR2, SR1, SL1.
- Sample rate: 6 MHz
- Float type: 32 bit
- Organised in chunks
  1 second each
  200 chunks
  → 1.2 billion samples
- Binary format



Data handling

Inspired by the Kaggle competition, we processed raw data to achieve a dataset in the format:

Acoustic emissions	Time to failure
…	…
…	…

For acoustic emissions we selected the channel (not the ST) with the highest variance. Due to memory constraints, we were unable to compute the variance directly over a tensor of 1.2 billion elements:

```
# contains at [i,j] the variance of the audio of signal j recorded by channel i
variance_matrix = np.array([
    [np.var(create_data_chunk_single_channel(input_folder, ch, chunk + chunk_idx, float_type=float_type))
     for chunk_idx in range(number_of_chunks)]
    for ch in list_channels
])

chosen_channel = int(np.argmax(np.sum(variance_matrix, axis=1)) + 1)
```

Data handling

Chosen channel: PU2

Converting from voltage [V] to arbitrary units (integer representation):

- N = $2^{\#\text{ target bits}} = 2^{16} = 65536$
- Range of target arbitrary units: [0, 65536]

$$\mathrm{AU} = \left\lfloor \frac{(V - V_{\min})}{(V_{\max} - V_{\min})} \times (N - 1) \right\rfloor$$

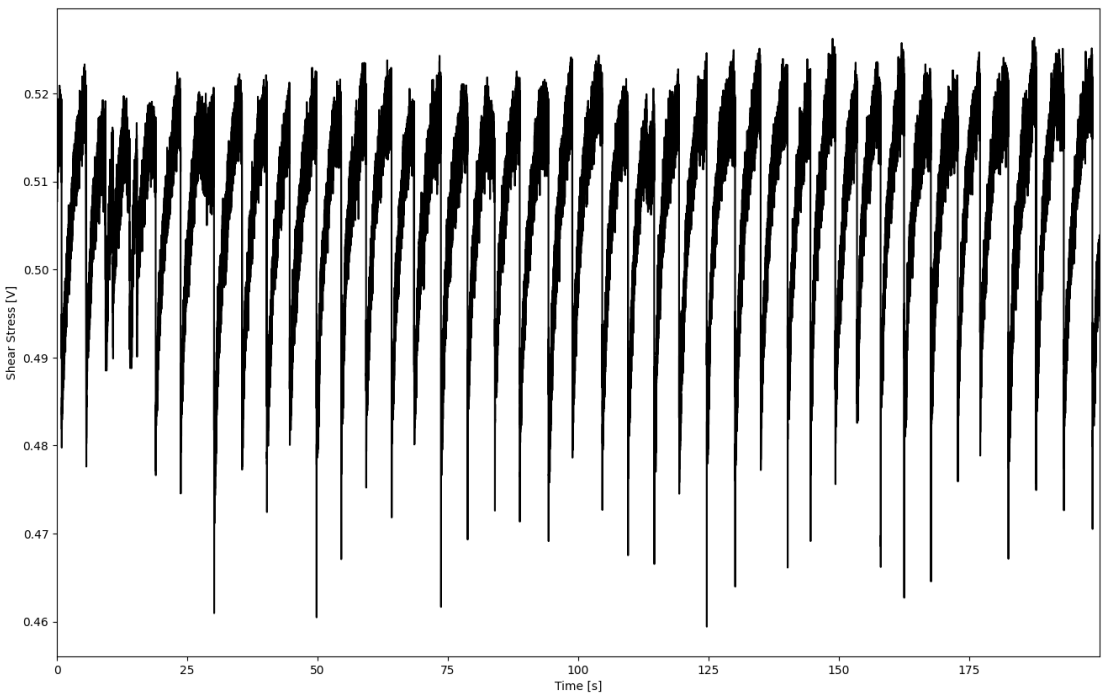
Instead, to build the time-to-failure signal, we had to downsample the data recorded by the first channel due to memory constraints, (we were not able to perform the necessary operations on the full-resolution signal). → Interpolation.

SAPIENZA
UNIVERSITÀ DI ROMA

Data pre-processing

Objective: to build the time-to-failure array

Input: shear stress tensor of shape (1.2b,)

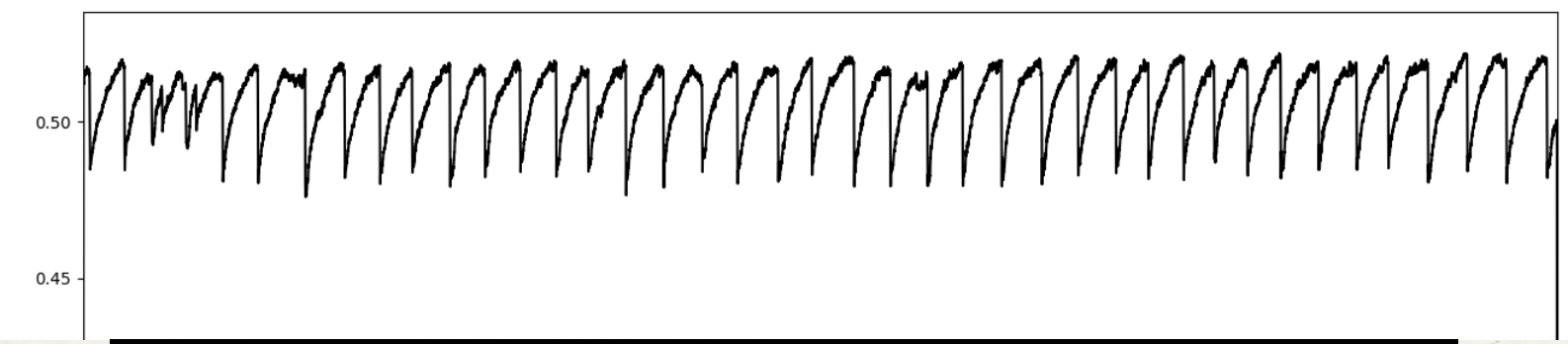


Data pre-processing

Downsampling: 6MHz → 1KHz

We now want to isolate the peaks to detect drops in shear stress.

Smoothing using a moving average



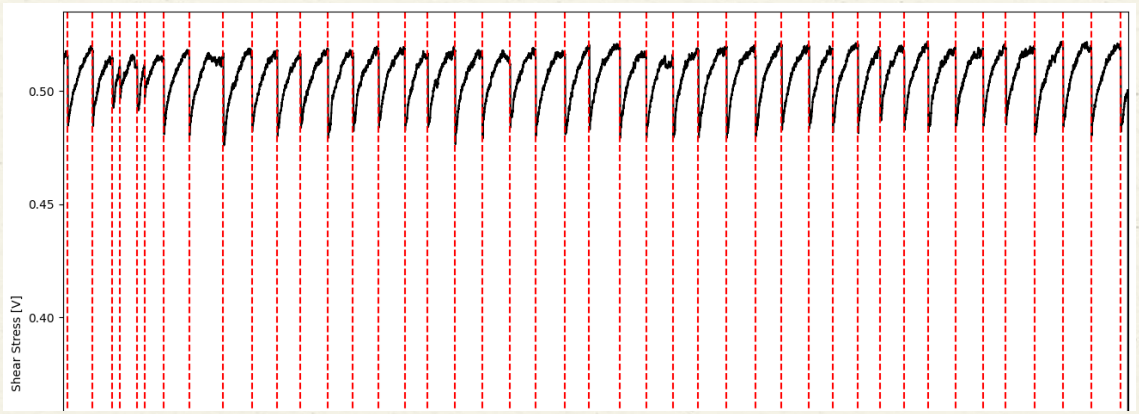

```
def moving_average(ST, window_size):
    # keep the original dimension while smoothing
    return np.convolve(ST, np.ones(window_size)/window_size, mode='same')

denoised_shear_stress = moving_average(shear_stress,50)
```

SAPIENZA
UNIVERSITÀ DI ROMA

Data pre-processing

Identify Rapid Changes in Stress

(Through derivative over a sliding window)



After detecting picks in the shear stress, we can build the time-to-failure array.

Remember: the shear stress is now sampled at 1 KHz.

Data pre-processing

```
TTF = np.zeros_like(shear_stress)  # Initialize output array (same size as shear_stress: sr = 1kHz)

# Loop through all shear_stress indices
for i in range(len(shear_stress)):
    # Find significant points occurring at or after the current index
    next_points = [point for point in significant_points if point >= i]

    if next_points:  # If such points exist
        # Compute time to the closest significant point
        TTF[i] = (min(next_points) - i) * (1 / fs_red)
    else:
        # No future events: set time to infinity
        TTF[i] = float('inf')
```

Shape of time-to-failure array: (200k,)

We now want to upsample the time-to-failure array back to 6MHz.
Why: later, we will perform some analysis on model's performance using less data (including downsampling). For this reason, we want to start with all the data available.

Data pre-processing

Upsampling usually introduces some error, which would not be ideal. Just because we lack sufficient memory, we should not compromise testing the model's full potential.

Great news: we are lucky. We do not actually need to interpolate the data since we already know how the time-to-failure evolves over time (it is time-dependent, after all).

It's like expanding the vector through time, but we already know the values because time is continuous. The only thing we need to consider is that **we want 6,000 points for each point** in the downsampled array (since 6 MHz / 1 kHz = 6,000).

Data pre-processing

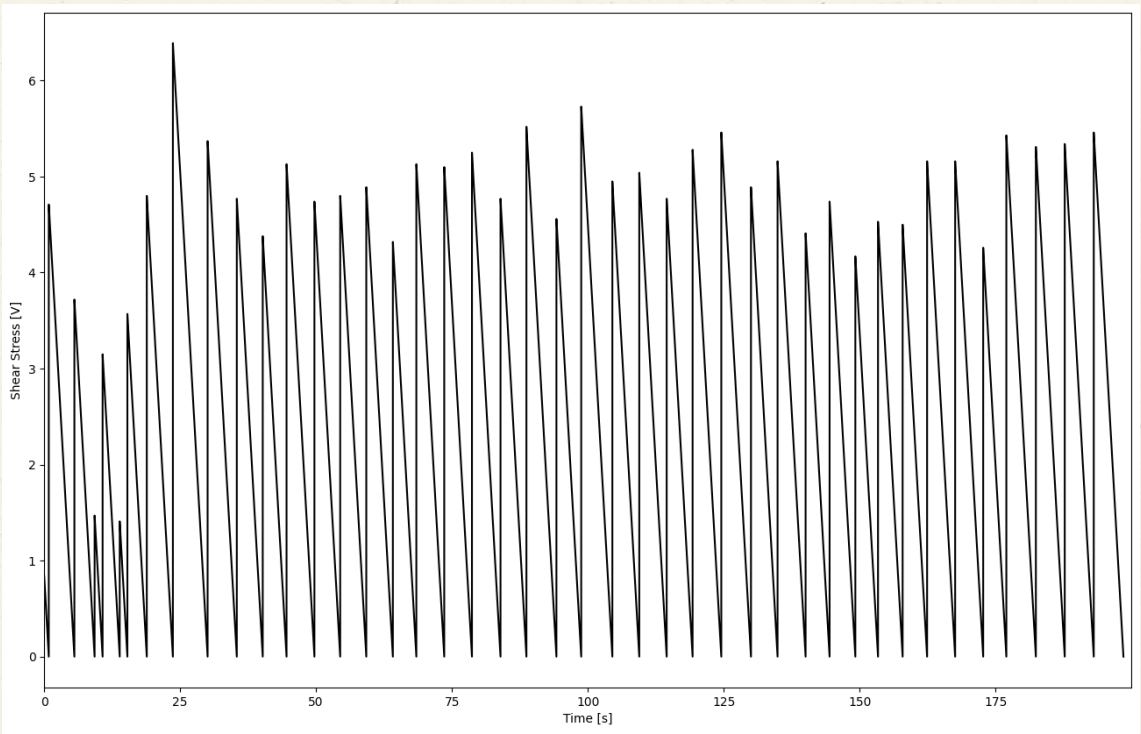
Here is how we did it:

```
# Compute the number of points to insert between each original point
points_each_point = int(fs / fs_red)

# Initialize the dense TTF array with a size scaled by the upsampling factor
TTF_dense = np.zeros(int(len(TTF) * points_each_point))

# Loop through the TTF array until the second-to-last element
for k in range(len(TTF) - 1):
    # Define the start and end indices in the dense array for the current segment
    start_index = k * points_each_point
    end_index = (k + 1) * points_each_point

    # If the current value is infinite, fill the remaining TTF_dense with infinity and exit the loop
    if np.isinf(TTF[k]):
        TTF_dense[start_index:] = np.inf
        break  # No need to continue interpolation past this point

    # Compute the step size (delta) between two consecutive values in the original TTF array
    delta = (TTF[k + 1] - TTF[k]) / points_each_point

    # Linearly interpolate the values between TTF[k] and TTF[k+1]
    for i in range(points_each_point):
        TTF_dense[start_index + i] = TTF[k] + i * delta
```

Now our time-to-failure is (almost) ready.

Data pre-processing

Additionally, we remove the infinite values from the time-to-failure array. Result:



(We also ensure the corresponding last elements are removed from the acoustic data array too)

Features engineering

Input: time-to-failure and acoustic data arrays. Both of length 1,191,240,001.

We could calculate a vast number of features from the acoustic data. However, we selected 30 commonly used features and performed optimization to identify the best subset of features for learning, limiting the subset size to 5.

The results showed that the model achieved better performance when using the following set of sample features:

- Number of picks in the signal
- 20^{th} percentile of the signal's standard deviation over a sliding window
- Mean (over time) Mel-Frequency Cepstral Coefficient 4 (out of 20)
- Mean (over time) Mel-Frequency Cepstral Coefficient 18 (out of 20)

Building dataset

- The full signal was divided into segments, to the tune of size 150,000 points (= 0.025 sec).
- For each segment, the selected features were calculated.
- The target value for each segment was defined as the time-to-failure at the end of the segment.
- Organised in Pandas dataframes.

```
--TRAINING DATAFRAME--
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6316 entries, 0 to 6315
Data columns (total 6 columns):
 #   Column                          Non-Null Count
---  ------                          --------------
 0   var_num_peaks_2_denoise_simple  6316 non-null
 1   var_percentile_roll50_std_20    6316 non-null
 2   var_mfcc_mean18                 6316 non-null
 3   var_mfcc_mean4                  6316 non-null
 4   start                           6316 non-null
 5   target                          6316 non-null
```

```
--TEST DATAFRAME--
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1624 entries, 0 to 1623
Data columns (total 6 columns):
 #   Column                          Non-Null Count
---  ------                          --------------
 0   var_num_peaks_2_denoise_simple  1624 non-null
 1   var_percentile_roll50_std_20    1624 non-null
 2   var_mfcc_mean18                 1624 non-null
...
 5   target                          1624 non-null
dtypes: float64(6)
```

Training

- Gradient Boosting
- 3-fold cross-validation (→ validation metrics)
- Training size: Approximately 80% of the full dataset (ensuring cuts align with full events)
- Hyperparameter optimization: Selecting the best parameters from a large search space
- Early stopping

```python
from lightgbm import LGBMRegressor
from sklearn.model_selection import GridSearchCV, KFold

param_grid = {
    'num_leaves': [4, 8, 16],
    'min_data_in_leaf': [5, 10, 20],
    'learning_rate': [0.01, 0.02, 0.05],
    'feature_fraction': [0.7, 0.9],
    'bagging_fraction': [0.5, 0.7],
    'reg_alpha': [0, 0.1, 0.5],
    'reg_lambda': [0, 0.1, 0.5]
}
```

```python
base_model = LGBMRegressor(
    boosting_type='gbdt',
    objective='fair',
    max_depth=-1,
    boosting='gbdt',
    bagging_freq=1,
    bagging_seed=0,
    metric='mae',
    verbosity=-1,
    max_bin=500,
    seed=0,
    n_jobs=-1
)
```

SAPIENZA
UNIVERSITÀ DI ROMA

Performance

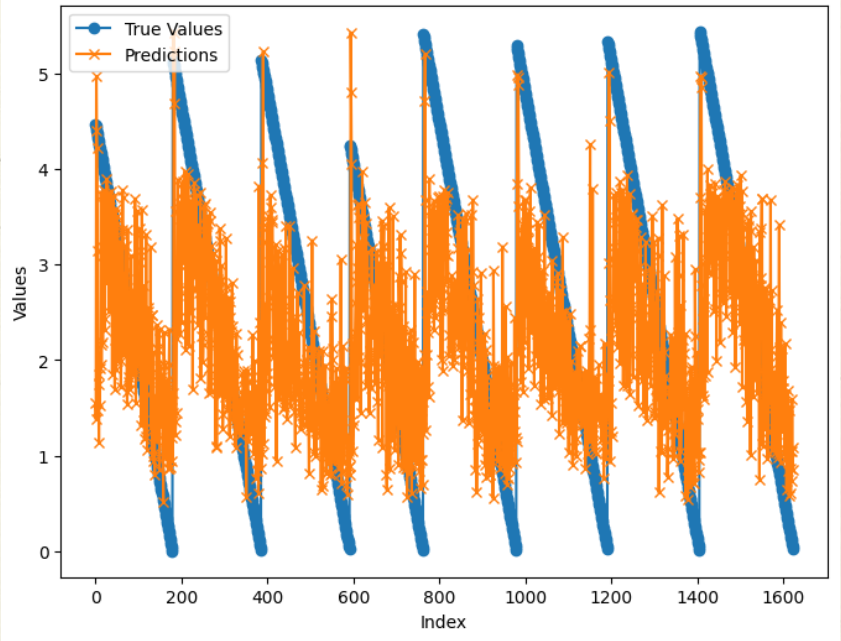
Our model vs. state-of-the-art model at the time of the Kaggle competition:

Metric on test set	B. Rouet-Leduc model	The Zoo model	Our model
MAE	0.9627	0.9558	0.9286
MSE	1.2304	1.2188	1.1930
R^2	0.3208	0.3318	0.3614

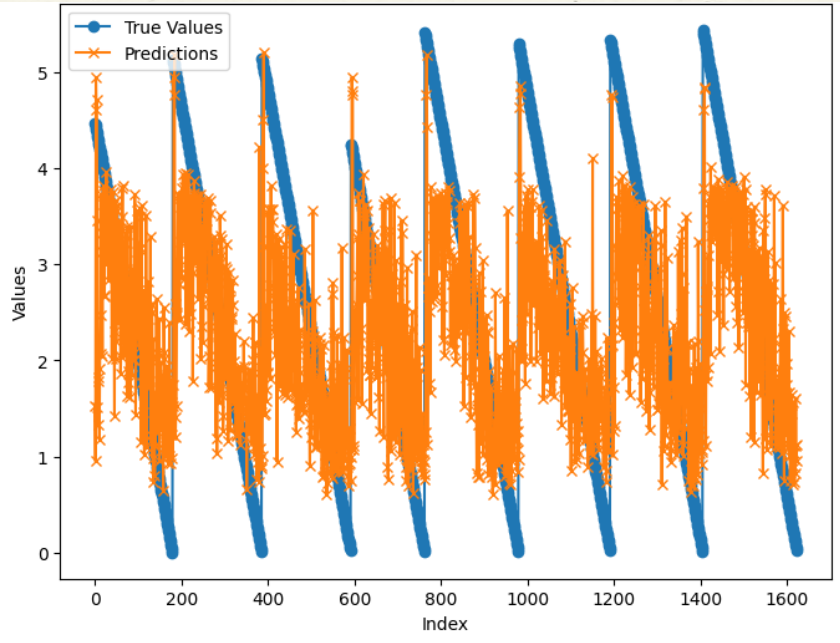
SAPIENZA
UNIVERSITÀ DI ROMA

Performance

The zoo model



Our model



SAPIENZA
UNIVERSITÀ DI ROMA

One more thing...

SAPIENZA
UNIVERSITÀ DI ROMA

Holes in the dataset

Is the Sampling Rate Too High? Selecting x samples every 40:

Metric on test set	x = 40	x = 30	x = 20	x = 10	x = 1
MAE	0.9286	0.9263	0.9369	0.9548	1.1061
MSE	1.1930	1.1930	1.2049	1.2226	1.3898
R^2	0.3614	0.3614	0.3487	0.3294	0.1334

SAPIENZA
UNIVERSITÀ DI ROMA

Using less events

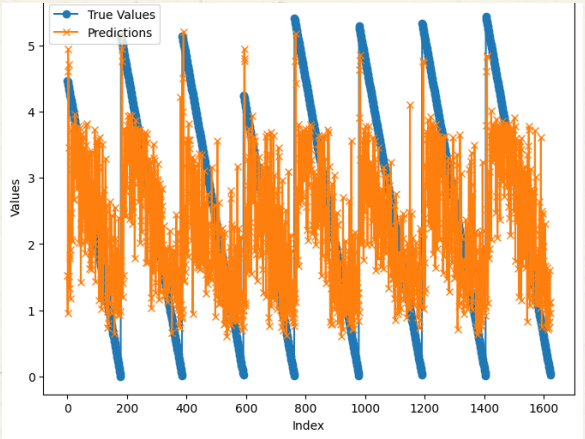
Using only fractions of the dataset:

Metric on test set	100%	75%	50%	25%	2.5%
MAE	0.9286	0.9395	0.9723	1.0083	1.1503
MSE	1.1930	1.2103	1.2410	1.2837	1.4220
R^2	0.3614	0.3428	0.3090	0.2607	0.0928

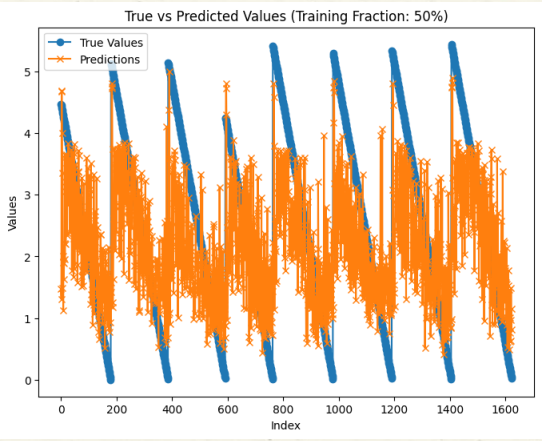
Holes in the dataset is better than reducing the dataset size by cutting it.

Using less events

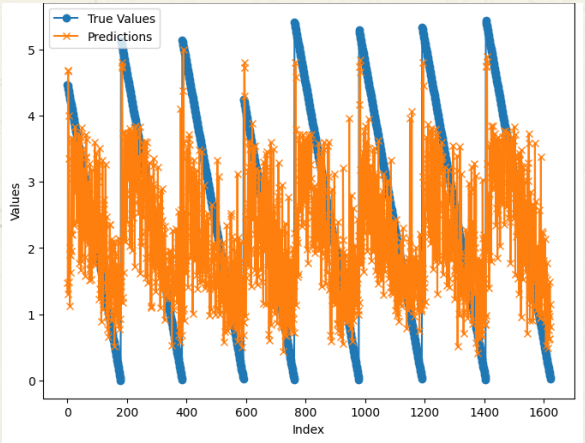
100%



50%



75%



2.5%