TUM Data Innovation Lab

Munich Data Science Institute (MDSI)

Technical University of Munich

&

PROCON IT GmbH

Final report of project:

Remaining Lifetime Estimation in Semiconductor Scenarios

Authors	Samed Bayer, Oliver Neumann, Deepak Raj, Yana Savva
Mentor(s)	Dr. Stephan Brehm and Dr. Aman Steinberg
Co-Mentor	Dr. Alessandro Scagliotti
Project Lead	Dr. Ricardo Acevedo Cabra (MDSI)
Supervisor	Prof. Dr. Massimo Fornasier (MDSI)

Jul 2023

Abstract

Thermomechanical fatigue represents one of the main root causes of transistor failures in power modules utilized in wind turbines, photovoltaic systems, and electric vehicles. It arises when materials within a transistor undergo cyclic mechanical loading and fluctuating temperature conditions, leading to accelerated degradation and eventual failure. By estimating the remaining lifetime of semiconductors, proactive maintenance, timely replacements, cost optimization, risk mitigation, and improved safety measures can be implemented to counteract thermomechanical fatigue.

When estimating the remaining lifetime of semiconductors, the first step involves implementing an efficient and scalable rainflow algorithm to count thermomechanical fatigue cycles. This algorithm serves as the foundation for training a machine learning model. Therefore, we first conduct a comprehensive study of the rainflow analysis and its various visualization techniques. During the data exploration and modeling phase, our focus lies on data acquisition, preprocessing, and a comprehensive discussion and evaluation of diverse machine learning models. After thorough analysis, we select the SGDRegressor as our final prediction model.

The scope of this project also includes the development of a web interface that presents the results of the rainflow analysis, allowing users to visualize the relationship between rainflow counts and various features. Furthermore, the application displays the estimated remaining lifetime of semiconductors, enabling practical implementation for real-world scenarios.

In conclusion, this work makes contributions to thermomechanical fatigue analysis and emphasizes the significance of proactive maintenance in critical systems.

Contents

1 Introduction

1.1 Background and Project Scope

Thermomechanical fatigue is one of the root causes of transistor failures observed in power modules utilized in wind turbines, photovoltaic systems, and electric vehicles. It occurs when materials within a transistor undergo cyclic mechanical loading and fluctuating temperature conditions, leading to accelerated degradation and eventual failure [3]. By estimating the remaining lifetime of semiconductors, proactive maintenance, timely replacements, cost optimization, risk mitigation, and enhanced safety measures can be implemented to counteract thermomechanical fatigue.
The starting point of this project is simulated data corresponding to a time series of temperatures measured by a sensor attached to the semiconductor system. Additionally, there is simulated data for the cooler temperature, power consumption, risk of failure, accumulated damage, and base failure rate, but these factors are not considered in this project. The simulation was performed using a physics-informed simulator developed at Procon IT GmbH. The initial task involves conducting a rainflow analysis to determine temperature stress cycles.
To accomplish this, an efficient and scalable rainflow algorithm needs to be implemented to count thermomechanical fatigue cycles. These rainflow counts are utilized as a foundation for training various machine learning models. Subsequently, the performance of these models will be assessed using a range of metrics. The top-performing model is then employed to estimate the remaining lifetime of the semiconductor.
Following that, the results of the rainflow analysis need to be presented through well-designed plots that demonstrate the relationship between rainflow counts and various features. Finally, we develop a web interface, featuring dashboards to display the plots and provide the estimation of the remaining lifetime.

1.2 Rainflow Analysis

Rainflow analysis is a method used to analyze and identify fatigue cycles in time series data [12]. Rainflow analysis utilizes a counting algorithm to detect and count the individual stress cycles. It is commonly used in engineering and material science to study the behavior of materials under cyclic loading. This algorithm follows several key steps to extract valuable information for fatigue analysis. To begin, the algorithm takes a sequence of cyclic stresses as its input, representing the time history of the system. It then proceeds by identifying the peaks and valleys in the data, which correspond to the maximum and minimum stress values, respectively. Next, the algorithm connects neighboring peaks and valleys, forming half-cycles. Each half-cycle represents a transition from a valley to a peak or vice versa. These half-cycles serve as the building blocks for the subsequent analysis. The rainflow counting step is crucial, as the algorithm pairs up the half-cycles to determine the number of full cycles. Both major (larger) and minor (smaller) ranges of the cycles are considered, ensuring comprehensive analysis. Once a full cycle is counted, it is eliminated from the data, allowing the algorithm to proceed to the next cycle. This step ensures that each cycle is properly accounted for and avoids duplication in subsequent calculations. Afterwards, the counting can be utilized to compute the damage incurred

by each cycle, employing well-established methods such as the Palmgren-Miner linear damage accumulation rule or other techniques for estimating fatigue life. This calculation provides valuable insights into the fatigue behavior of the system.
The rainflow counting algorithm has proven useful in predicting the fatigue life or lifetime of semiconductors. In our study, these techniques can be applied as follows:
Data collection: The first step involves gathering cyclic load or stress data that corresponds to the operating conditions of the semiconductors. This data can be obtained from real-world operational settings, simulated tests, or accelerated testing methods. In our case, we collect simulation data that tracks the datetime and temperature parameters (and other parameters).
Rainflow counting: Once the data is collected, we apply the rainflow counting algorithm to analyze it. This algorithm effectively identifies significant peaks and valleys within the cyclic load or stress history. By connecting neighboring peaks and valleys of temperature, half-cycles are formed.
By utilizing these techniques, we can gain insights into the fatigue life and make predictions about the lifetime of semiconductors in our study.

1.3 Palmgren-Miner Linear Damage Accumulation Rule

Miner's rule, specifically the Palmgren-Miner rule, is a commonly used approach in fatigue analysis. It allows for the estimation of cumulative damage resulting from multiple cyclic stress cycles on a material or structure. The fundamental concept underlying the Palmgren-Miner rule is that the accumulated damage experienced by a component is directly related to the ratio of applied load cycles to the fatigue strength of the material under a single load cycle [8]. The Palmgren-Miner rule is given by,

$$\sum_{i=1}^{k} \frac{n_i}{N_i} = 1 \tag{1}$$

where k = number of stress levels in the block loading spectrum, n_i = number of cycles at each stress level in the block loading spectrum and N_i = number of cycles to failure at each stress level [14].
The fatigue strength, also known as the endurance limit, is determined through experimental testing or obtained from relevant literature specific to the material being analyzed. However, since we used the simulation data, we actually do not have these material specific properties. This value represents the maximum cyclic stress that the material can endure indefinitely without failing.
In our case, these fatigue strength values are not present. The number of load cycles (N) corresponding to each distinct stress level or range is determined. This can be accomplished through techniques like rainflow counting or other cycle counting methods. For each stress level, the ratio of the number of load cycles (N) to the fatigue strength (Nf) is calculated. This ratio, known as the damage ratio (D), represents the relative damage caused by the corresponding stress level. The damage ratios for all stress levels are summed together to obtain the cumulative damage (D_{cum}) resulting from the combined load cycles. The cumulative damage (D_{cum}) is then compared to a predefined threshold value, typically set at 1. If the cumulative damage exceeds 1, it indicates that the

accumulated damage has surpassed the material's fatigue strength, implying a higher probability of failure. It's important to note that the Palmgren-Miner rule assumes a linear relationship between fatigue damage and the number of load cycles. However, it is an approximation and may not consider all factors influencing fatigue behavior, such as load sequence effects or material properties at different stress levels. Despite its simplifications, the Palmgren-Miner rule is widely accepted and offers a practical method for estimating cumulative fatigue damage in engineering applications. It assists engineers in making informed decisions regarding design life, maintenance intervals, and safety considerations for structures subject to cyclic loading.
In our analysis, we attempted to apply the Palmgren-Miner rule to our datasets using common fatigue strength values from other projects. However, these values did not yield satisfactory results when applied to our simulation data. Additionally, we wanted to take into account the effect of the cycle ranges since they highly affect the life time of the semiconductor. As a result, we decided to proceed with linear regression models instead of a simple implementation of the Palmgren-Miner rule.

1.4 Rainflow Analysis Visualization

There are several effective visualization techniques available to present the outcomes of the rainflow counting algorithm. Here are some commonly employed methods:

Cycle Count through Timeline. This Figure 1 displays the temperature over time. The cycles identified by the rainflow analysis are plotted here highlighted by red dots. The x-axis represents the timeline in seconds, while the y-axis represents the stress values - temperature in our case in Kelvin. The identified cycles are plotted within respective stress range bins, allowing for a clear understanding of their distribution throughout the timeline. [1]

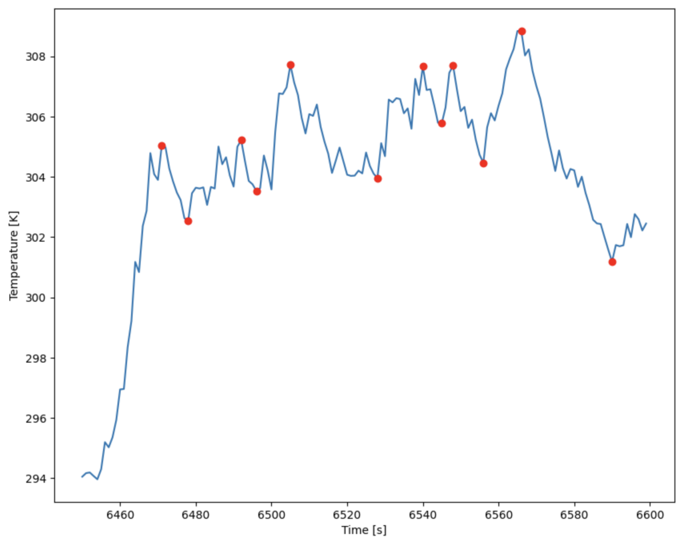
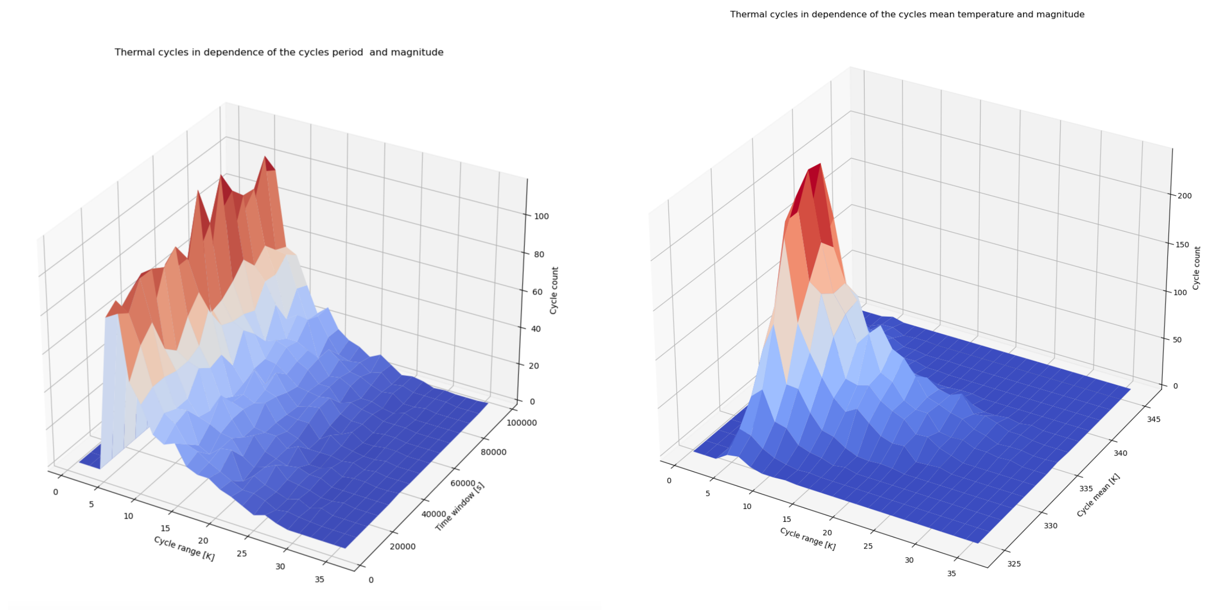


Figure 1: Temperature over time with highlighted cycles.

Cycle Range vs. Time vs. Count Plot. This plot showcases the temperature range on the x-axis in Kelvin, the time range denoted as cycle period on the y-axis in seconds, and the cycle count on the z-axis. Each data point represents an individual cycle, enabling the observation of how stress range and count vary within the dataset over time. Different markers or colors can be utilized to differentiate between various types or occurrences of load cycles.

This plot can be interpreted as follows: for all time windows we can see mostly cycles with low range. However, probably closer to the end of the simulation for some large time windows the range of the cycle start to grow which means that the semiconductor is about to die.

Cycle Range vs. Mean Stress vs. Count Plot. Similar to the previous plot, this visualization presents the stress range on the x-axis. The mean stress is displayed on the y-axis, the cycle count - on the z-axis. Individual cycles are represented by data points, enabling the examination of the relationship between stress range, count, and mean stress throughout the dataset. Distinct markers or colors can be used to differentiate between different load cycle types or occurrences [1].



(a) Thermal cycles in dependence of the time window and cycle range.

(b) Thermal cycles in dependence of the cycle mean temperature and cycle range.

Figure 2: Rainflow Matrices.

Here the most severe cycles have high thermal swings, which indicates how important it is to include those for the statistical data analysis, those most likely have the most influence on the remaining lifetime of the semiconductor. These visualization techniques offer valuable insights into the distribution, patterns, and characteristics of load cycles identified through the rainflow counting algorithm. They aid in the comprehension and analysis of the data, facilitating effective decision-making in fatigue analysis and structural health monitoring.

2 Statistical Data Exploration

2.1 Data Acquisition

The data is **simulated** using Python code. The code generates data by simulating the parameters of a semiconductor and saving the parameter values over time until the semiconductor shuts down. Those parameters include temperature of the transistor, temperature of the cooler, power usage, base failure rate (predicted failure), accumulated damage, risk of failure. The simulation is based on a configuration file written as Python script. Both Python script and config are provided by the client (PROCON IT GmbH).
The configuration file contains various parameters that define the characteristics of the simulation. They are related to semiconductor and cooler features, such as heat transfer coefficients, specific heats, masses, areas, and temperature ranges. Those parameters are known constants for the semiconductor environment.
The script initializes the simulation by setting the initial time, creating a power curve generator yielding normally distributed values based on the temperature at which the transistor enters thermal throttling. The generator produces the precentages of maximum power, providing a relative values instead of absolute ones. The steps this simulation includes are: calculating heat flux, heat loss, energy loss, temperature loss for each datapoint; generation of the temperature of the semiconductor for each value calculated by the power curve generator - this relation between the power curve and temperatures is deterministic, since the parameters of the semiconductor and the power curve are given in the config.
It then runs the simulation for each transistor, saving the generated data in separate CSV files for temperature, power consumption, failure rates, accumulated damage, and risk of failure. The data is organized into directories for each transistor, which are then compressed and uploaded to an S3 bucket for long-term storage.
The quantity of simulations has a significant impact on the model's overall performance. For this study, we were initially provided with 8 configurations, each consisting of 10 simulations. Thus, the results presented in this work can be further enhanced in future iterations by incorporating additional data to enrich the model's descriptive capabilities.

2.2 Data Preprocessing and Feature Engineering

To predict the lifetime estimation of the semiconductor, based on rainflow cycles, we initially conducted the rainflow analysis on the simulated temperature data to extract the temperature stress cycles. For this purpose, we employed the Python implementation of the standard practice ASTM E1049-85 rainflow cycle counting algorithm for fatigue analysis [12]. This process enabled us to obtain the rainflow cycles along with their associated features, such as cycle magnitude, duration, and temperature.
To develop a model predicting material failure, we transformed the problem into a regression task. As we lacked actual 'y' values (failure rates), we generated synthetic 'y' values based on the material's lifetime. We calculated the 'time-to-failure' for each data point as follows: If the material failed at time $'T'$, then for each time $t < T$, the 'time-to-failure' was set to T-t. For time $t >= T$, the 'time-to-failure' was set to 0. We used this 'time-to-failure' as the target variable for regression, with rainflow counts as the input

feature for each time point. As for the failure rate, it is typically defined as the number of failures divided by the total time. However, since we only have data for one failure from the simulation, calculating a failure rate is not meaningful in this context. Instead, the 'time-to-failure' prediction from the regression model serves as a more appropriate measure of when the material is expected to fail.
The next step is to calculate the rainflow counts for each data point and incorporate them as features in the regression models. To achieve this, we implemented a rolling window approach on the temperature data, computing the rainflow count for each window rather than individual data points. Each window encompasses rainflow counts per bin, indicating the cycle sizes. Consequently, we obtain a time series of rainflow counts. Notably, the bins are designed to have smaller sizes at the beginning of the simulation, gradually increasing towards the end.
Next, we utilized this time series of rainflow counts to predict the 'time-to-failure' calculated in the previous step. It is essential to highlight that the 'time-to-failure' is also transformed to represent the time to failure for each window rather than each data point. In our implementation, the window size determines the size of the window, while the step size determines how much the window moves for each step. In most cases, we set these sizes to be the same, but it's important to note that this choice can influence the model's performance.
In order to use processed data in the regression model, we introduced a new feature representing the cumulative sum of the rainflow counts, which helps accumulate the effect of these counts. Thus, higher rainflow counts lead to a faster increase in the cumulative sum, indicating a potential decrease in the remaining lifetime of the semiconductor. Finally, we prepared the 'X' and 'y' data for the model by reshaping them into 2D and 1D numpy arrays, respectively, and partitioned them into training and test datasets. Since this is a time series problem, we adopted a temporal train-test split approach. The training set comprises all data up to a specific time point (80% for our case), and the test set includes all data after that time point, ensuring a proper temporal evaluation of the model's performance.
For calculating the rainflow counts, we employed two different methods. The first method involves computing a weighted sum, where each count is multiplied by the corresponding bin weight, representing the cycle temperature magnitude range. This results in a single feature representing the accumulated rainflow count per window, depicted as follows:

$$x_i = \sum_{i=0}^{n} \text{rainflow\_count}(window_i) \times bin\_size \tag{2}$$

Since the amplitude of temperature cycles affects the failure rate, this weighted sum can be a more meaningful feature compared to just the total count. However, we assumed that there is an linear relationship between failure rate and cycle temperature magnitude in this case. In order to overcome this issue, we also treat each cycle magnitude range (each bin) as separate features and let the model learn which one of the bins affects most. The number of features corresponds to the number of bins in the simulation, allowing the model to assign weights to each bin's counts. Consequently, the input of the model consists of vectors, with each vector representing the rainflow counts for a specific bin. The number of these vectors equals the number of bins used in the simulation.

$$x_{ij} = \sum_{i=0}^{n} \text{rainflow\_count}(window_i, bin\_size_j) \tag{3}$$

In this Data Preprocessing part, we generate a sum of rainflow counts as input and 'time-to-failure' as output. These processed datasets are then utilized to train the regression models, explained in the subsequent chapters. We perform the train-test split in a temporal manner, where the model is trained solely on past data and tested on future data. The model's performance is evaluated using the metrics described in the following chapter.

3 Modeling

3.1 Metrics used to compare algorithms

To assess the accuracy of our prediction, we compare the predicted values with the actual values from the testing data. Common evaluation metrics for evaluating machine learning models for regression tasks include Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and R^2 [5][6]. In the following formulas, y_i represents the actual values, $\hat{y}_i$ the predicted values, $\bar{y}$ the mean value of the sample and n the sample size.
RMSE is calculated by taking the square root of the MSE. This transformation is valuable as it allows for a more intuitive interpretation of the average error magnitude, as the RMSE values are in the same units as the target variable. Consequently, it facilitates a more meaningful understanding of the average deviation between the predictions and the actual values. The RMSE is given by:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{4}$$

Some researchers have recommended using the MAE instead of the RMSE [15]. MAE measures the average absolute difference between the predicted values and the actual values, offering advantages in interpretability over RMSE. It provides a measure of the magnitude of the errors made by the model. The MAE is given by:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{5}$$

For both metrics (RMSE and MAE), a lower error value indicates better performance, indicating that the model's predictions are closer to the actual values [6]. Another relevant metric is R^2, which represents the proportion of the variance in the dependent variable (actual values) that can be explained by the independent variable (predicted values). It ranges from 0 to 1, where higher values indicate a better fit of the model to the data and a value of 1 represents a perfect fit [6]. The R^2 is given by:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{6}$$

In our analysis, we also introduce specific relative measures to evaluate the performance of our models in predicting the remaining lifetime of transistors. The Mean Absolute Error relative to whole lifetime calculates the mean absolute relative difference between the estimated remaining lifetime and the measured remaining lifetime, normalized by the measured lifetime of the whole simulation. This provides an overall assessment of the accuracy of the predictions relative to the entire lifespan of the transistor.

$$\text{Mean Absolute Error Relative to Whole Lifetime} = \frac{(Estimated\,Lifetime - Measured\,Lifetime)}{\text{Whole Lifetime}} \tag{7}$$

The Mean Absolute Error relative to ground truth lifetime measures the mean absolute relative difference, normalized by the measured remaining lifetime, providing an overall assessment of the accuracy of the predictions relative to the lifetime of each data point.

$$\text{Mean Absolute Error Relative to Ground Truth Lifetime} = \frac{(Estimated\,Lifetime - Measured\,Lifetime)}{\text{Measured Lifetime}} \tag{8}$$

Next, the overprediction rate quantifies the proportion of instances where the predicted remaining lifetime exceeds the actual measured remaining lifetime, offering insight into the model's tendency to overestimate the remaining lifespan of the transistor.

$$\text{Overprediction rate} = \frac{\sum_{i=1}^{n}(\hat{y}_i > y_i)}{n} \tag{9}$$

Similarly, the underprediction rate calculates the proportion of cases where the predicted remaining lifetime falls short of the actual measured remaining lifetime, indicating the model's inclination to underestimate the remaining lifespan of the transistor.

$$\text{Underprediction rate} = \frac{\sum_{i=1}^{n}(\hat{y}_i < y_i)}{n} \tag{10}$$

3.2 Anomaly Detection

In this project, we explored the algorithm of Anomaly Detection [11] by utilizing the Microsoft-provided Python code from their GitHub repository. However, we found that the algorithm did not suit our specific problem due to several limitations.
Firstly, the algorithm lacks the capability to indicate the precise time of the semiconductor's real death, which is a crucial aspect in our context. Additionally, the anomalies detected by the algorithm do not align with the actual death of the semiconductor, rendering it unsuitable for our purposes. Lastly, the algorithm does not account for the continuity of the data, which is a fundamental requirement in our scenario.
As a result, we determined that the algorithm described in the paper did not provide a viable solution for our specific requirements.

3.3 Linear Model

Approach

In this section, we explore a linear model approach that incorporates rainflow counts as features. Rainflow counting is a technique used to analyze and quantify fatigue damage

in engineering structures subjected to cyclic loading. By including rainflow counts as features in a linear model, we leverage this information to enhance model accuracy and effectiveness.
To incorporate rainflow counts as features in a linear model, we follow the steps described in 2.2. This way two setups are created: (1) - one feature as accumulated weighted cycle counts, where weights are the bin ranges; (2) - separate cycle count as features.
Finally, we train a linear model, such as linear regression, using the created dataset. The model learns the relationship between the rainflow count features and the target variable.

Table 1: Metrics for 1 simulation

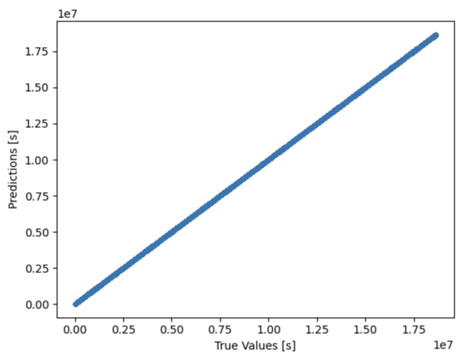
Case	MAE [s]	MAE relative to lifetime	MAE relative to GT	RMSE [s]
Setup 1	9238	0.0001	0.005	10947
Setup 2	6443	0.00006	0.003	7987

Case	R-squared	Overprediction rate	Underprediction rated
Setup 1	0.99	0.61	0.68
Setup 2	0.99	0.9	0.31

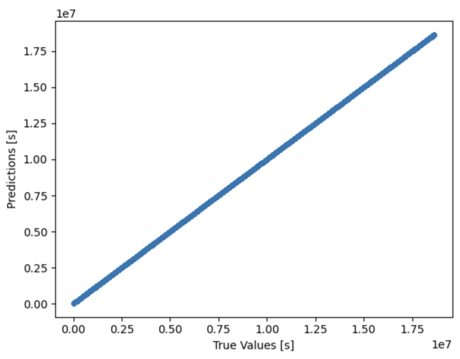
Table 1: The table presents 2 setup cases along with their corresponding metrics: 1 - one feature as accumulated weighted cycle counts, where weights are the bin ranges; 2 - separate cycle count as features.

Results

The obtained results demonstrate the model's strong predictive capability in estimating the remaining lifetime, particularly in scenarios where the lifetime spans several years. Notably, in the conditions when the whole lifetime can be years a tolerable error range was reached. To gain deeper insights into the prediction performance, it is customary to employ the prediction vs. ground truth plot, a widely recognized technique for visual assessment and evaluation of model predictions. In both cases an important take from those



(a) Prediction vs. True values - setup 1



(b) Prediction vs. True values - setup 2

Figure 3: Comparison of setup 1 and 2

plots is that they almost perfectly reflect the real ground truth. This is the case when

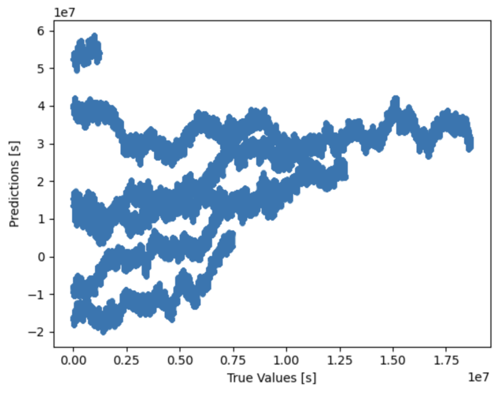
only one simulation is used for trainig - model might be overfitted. A logical continuation of those experiments would be adding more simulations into the training dataset. This result also shows us that setup 2 where we use separate bin counts as columns should be chosen for the future experiments.
The train-test split operation here should be conducted in a way where we keep the notion of time series data. There are 2 ways do do it: 1 - first 80% of each simulation is for training, last 20% is for testing; 2 - 80% of full simulations is for training, 20% is for testing.

Table 2: Metrics for 1 simulation

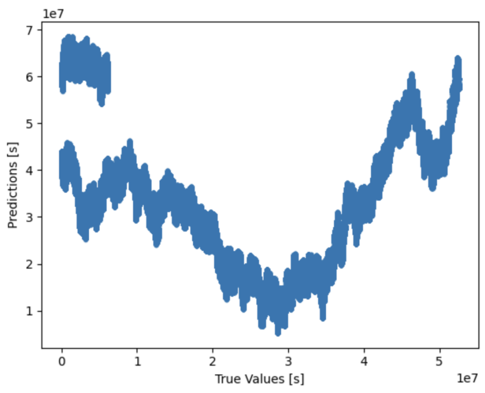
Case	MAE [s]	MAE relative to lifetime	MAE relative to GT	RMSE [s]
Setup 3	5374162	6.22	13.1	7869036
Setup 4	10034422	11.6	9.47	16871316

Case	R-squared	Overprediction rate	Underprediction rated
Setup 3	-4.83	0.77	0.23
Setup 4	-0.1	0.59	0.41

Table 2: The table presents 2 setup cases along with their corresponding metrics: 3 - first 80% of each simulation is for training, last 20% is for testing; 4 - 80% of full simulations is for training, 20% is for testing.



(a) Prediction vs. True values - setup 3



(b) Prediction vs. True values - setup 4

Figure 4: Comparison of setup 3 and 4

Throughout the course of this experiment, it has been deduced that the intrinsic linearity of the model is a valuable attribute within the domain of this dataset. Nonetheless, it becomes evident that some other algorithm is imperative to accurately capture the diversity exhibited by a set of simulations since the metrics gained from multiple simulations have a wider range.

3.4 SGD Regressor

We achieved favorable results with the Linear Model; however, our goal is to train the model incrementally with new simulation data, allowing it to learn from all simulations associated with the same type of semiconductor.

Approach 1

Therefore, we employ the SGDRegressor from the scikit-learn Python package [10], which facilitates incremental learning, also known as 'online learning.' This feature enables SGDRegressor to execute linear regression and be incrementally trained using the partial fit method. In the model class, when we call the train method with a set of temperatures, the model is trained with that data. Subsequently, if we call the train method again with a different set of temperatures, the model continues training with the new data. In essence, each call to the train method continues the training of the existing model with the new data stochastically. This ensures that the model retains the knowledge gained from previous epochs, preventing it from forgetting what it has learned when presented with new data. One of the challenges we encountered when transitioning to SGD Regressor was the requirement for feature scaling. Unlike the linear regression model, SGD is sensitive to feature scaling, making it advantageous to standardize the data before training. To achieve this, we utilized the StandardScaler from the scikit-learn Python package [10] for feature scaling in our modeling process. The StandardScaler standardizes features by removing the mean and scaling to unit variance. It fits and transforms the rainflow counts data before model training. It is curcial to apply the same transformation to any data passed into the model for prediction. Therefore, only the transform function is used on the testing data, not the fit feature of the scaler because the same scaling (mean and standard deviation) as the training data should be maintained. This approach guarantees that the model remains unbiased by any information in the testing data. Moreover, the SGDRegressor model can benefit from multiple epochs of training. Unlike Linear Regression, which analytically solves for model parameters, SGDRegressor employs gradient descent and might require several passes over the data to converge to optimal parameters. Consequently, we fine-tuned the hyperparameters, selecting different epoch numbers for various cases while retaining the default learning rate.

Result 1

Subsequently, the model was trained, and the corresponding results can be observed in Table 3. Initially, we utilized the cumulative sum of all temperature cycle ranges, which, as shown in Table 3, yielded superior results compared to other approaches. Setup 1 illustrates the metrics for one simulation, also employed in the Linear Model section. On the other hand, Setup 2 represents training the model with multiple features as separate cycle counts.

Contrary to our hypotheses, the results in Table 3 did not align, as the performance worsened with multiple features when assessed with the training data. According to the literature, bins with the highest cycle range should have higher impact than linear relationships. As a result, Setup 2 was anticipated to yield lower errors compared to Setup 1,

Table 3: Metrics for 1 simulation

Case	MAE [s]	MAE relative to lifetime	MAE relative to GT	RMSE [s]
Setup 1	10706	0.0001	0.004	13073
Setup 2	24947	0.0003	1.54	30225

Case	R-squared	Overprediction rate	Underprediction rated
Setup 1	0.999	0.74	0.26
Setup 2	0.999	0.64	0.36

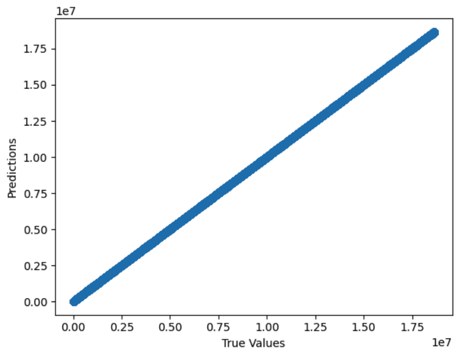
Table 3: The table presents 2 setup cases along with their corresponding metrics: 1 - one feature as accumulated weighted cycle counts, where weights are the bin ranges; 2 - separate cycle count as features.

given that the model would learn the importance of counts belonging to the specific bin from the data. However, this was not observed, even though the difference was negligible. The coefficients of the model are analyzed in Table 4 to understand the reasons behind the observed behavior. Surprisingly, higher values in certain bins do not correspond to higher importance, especially evident in the last bin (x_{13}), which exhibits the least coefficient value. This suggests that it has minimal impact on decreasing the semiconductor's lifetime according to the trained model.
Additionally, there is an issue with x_7 having a positive value, contrary to theoretical expectations, where it should reduce the semiconductor's lifetime when in use. Investigation reveals that large-value bins have small counts, leading to a sparse feature space that the model struggles to handle effectively.
In addition, the significant difference in the magnitude of counts among different bins can lead to challenges. Linear regression models tend to prioritize features with larger values, favoring the first bins in our scenario. Despite standardizing the features, the variance within each feature can still impact the model's learning.
Moreover, lower bins' counts exhibit more uniform variations across different cycles, providing the model with additional information to learn linear relationships from. As a result, the first bins might better explain the linearity, given their occurrence around each window size with almost the same probability.

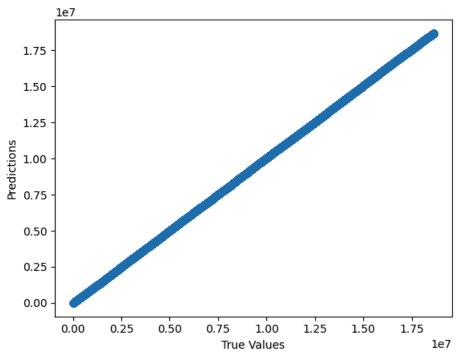
Table 4: Values of x_1 to x_{15}

x_1	x_2	x_3	x_4	x_5
-2330313	-2321103	-2302851	-2317542	-2379920
x_6	x_7	x_8	x_9	x_{10}
-2379054	21968	-780987	-997682	-2384306
x_{11}	x_{12}	x_{13}	x_{14}	x_{15}
-2573449	-753447	-41622	0.0	0.0

To gain deeper insights into the prediction performance, a prediction vs. ground truth plot is presented in Figure 5. The plot reveals that the SGDRegressor model's predictions closely resemble those of Linear Regression, indicating a similar explanatory capability for the dataset.



(a) Prediction vs. True values in seconds - setup 1



(b) Prediction vs. True values in seconds - setup 2

Figure 5: Comparison of setup 1 and 2 for SGD Regressor

Approach 2

To train the model incrementally with multiple datasets belonging to the same configuration, we implemented a loop that retrieves files from the S3 AWS Bucket and performs model training iteratively. The loop begins by initializing the SGD Regressor model. Then, for each dataset, it downloads the data, extracts the temperatures, and trains the model using the partial fit method. This approach ensures that the model updates with new data while retaining the knowledge from previous datasets. After training with all datasets, it is tested on two different datasets. The first test involved using the dataset used during partial training referred to as Setup 1. Subsequently, we tested the model on an unseen dataset to evaluate its performance on new data.

Result 2

We only present the results of the cumulative summation calculation of rainflow counts since its superior to other methods. Please refer to Table 5 for the detailed results.

Table 5: Metrics for using all simulations

Case	MAE [s]	MAE relative to lifetime	MAE relative to GT	RMSE [s]
Setup 1	2990744	0.032	0.244	3449618
Setup 2	7058605	0.134	0.573	8147195

Case	R-squared	Overprediction rate	Underprediction rated
Setup 1	0.589	0.999	0.001
Setup 2	-6.206	1.0	0.0

Table 5: The table presents 2 setup cases along with their corresponding metrics: 1 - test results on the dataset which is used as a training dataset first; 2 - test results on the dataset which the model has never seen before.

Based on the results, we can conclude that our model performs nearly as well on unseen data as it does on the seen dataset. However, it is worth noting that training the model with multiple datasets affects its performance on the first dataset, resulting in slightly worse performance compared to training it solely with the first simulation dataset.

3.5 Tree-based Methods

Approach

The Random Forest Regressor algorithm is based on the concept of decision trees, which are simple yet powerful models for the regression task. Random Forest Regressor takes this concept further by combining multiple decision trees to create an ensemble model. It introduces the idea of random selection of the subsets of the original training data, with replacement. This process is known as bootstrap aggregating or bagging. Each subset is called a bootstrap sample. Then the algorithm follows by building an individual decision tree for each bootstrap sample. At each node of the tree, the algorithm selects the best feature and split the data based on a criterion (Squared error in our case). However, instead of considering all features, Random Forest Regressor only considers a random subset of features for each split. This randomness adds diversity to the ensemble. From this decision tree setup an ensemble of decision trees is created by repeating the above step to build multiple trees. To make a prediction, each tree in the ensemble independently produces a prediction. For regression, the final prediction is typically the average of all individual tree predictions.
The AdaBoost Regressor algorithm is another supervised learning algorithm that falls under the category of ensemble methods. The AdaBoost Regressor algorithm follows a boosting approach, which involves sequentially training multiple weak learners and combining their predictions to form a stronger overall model. First step of the algorithm is to train a weak learner (decision tree with a small depth). Then by calculating the weighted error of the weak learner, updating technique is created: the weights of the misclassified samples are increased, while correctly classified samples receive lower weights. The final prediction is typically obtained by taking a weighted average of the individual weak learner predictions (weighting is based on their performance during training).
In this study, our primary hypothesis centered around the sequential nature of the data, particularly the stress calculations obtained through the rainflow counting algorithm. This algorithm accounts for the cumulative property of stress, indicating that as the semiconductor is utilized for a longer duration, it experiences greater stress. We explored the impact of tree-based methods on this property. Specifically, we employed two regression algorithms, namely Random Forest Regressor and AdaBoost Regressor. The features for these models included accumulated cycle counts, which were used either as a single feature or as multiple features for each bin.

Results

The utilization of a tree-based method introduces the possibility of neglecting the property of accumulation, which is a crucial aspect of the data. This issue became apparent during the simulation employed for training the models. As both approaches rely on the Decision Tree algorithm, samples corresponding to the end of the simulation were overlooked during feature splitting. Consequently, the algorithm consistently favored

Table 6: Metrics for 1 simulation

Case	MAE [s]	MAE relative to lifetime	MAE relative to GT	RMSE [s]
Setup 1	170918	0.19	6.8	178906
Setup 2	126852	0.15	5.45	136297
Setup 3	171599	0.19	6.8	178694
Setup 4	126918	0.15	5.45	136358

Case	R-squared	Overprediction rate	Underprediction rated
Setup 1	-11.87	1	0
Setup 2	-6.47	1	0
Setup 3	-11.84	1	0
Setup 4	-6.48	1	0

Table 3: The table presents 4 setup cases along with their corresponding metrics: 1 - Random Forest Regressor with one feature as accumulated weighted cycle counts, where weights are the bin ranges; 2 - same feature with AdaBoost; 3 - RFR with separate cycle count as features; 4 - same features with AdaBoost.

smaller bins and predicted a constant high value, implying that the semiconductor would never reach failure.

3.6 Support Vector Regressor

Approach

Support Vector Regression (SVR) is a machine learning algorithm for regression problems with a quantitative response inspired by Support Vector Machines (SVMs) for binary classification. Support Vector Machines (SVMs) is a class of supervised learning algorithms that aim to find an optimal decision boundary that separates different classes of data points. The key idea behind SVMs is to maximize the margin, which is the distance between the decision boundary and the nearest data points from each class. By maximizing the margin, SVMs promote better generalization and robustness to unseen data. SVMs achieve this by transforming the input data into a higher-dimensional feature space using a kernel function. In this transformed space, SVMs can find a linear decision boundary that corresponds to a nonlinear decision boundary in the original input space [7]. SVMs can be extended to regression problems by replacing the quadratic error function of simple linear regression with an ϵ-insensitive error function [13]. This error function ignores errors below a certain threshold ϵ and penalizes points outside the ϵ threshold (see Figure 6)[4]. It can be seen as fitting a flexible tube of ϵ width around the estimated function to handle errors above and below a certain threshold (see Figure 7)[9].
SVR offers the advantage of computational efficiency since its complexity is independent of the input space dimensionality. Moreover, it exhibits excellent generalization capability, leading to high prediction accuracy [2].As discussed before, we assume linearity between the input variables and the target variable. To account for this linearity, we utilize a linear kernel Support Vector Regression (SVR) model. The scikit-learn Python package offers the 'LinearSVR' model specifically designed for this purpose. It is similar to the

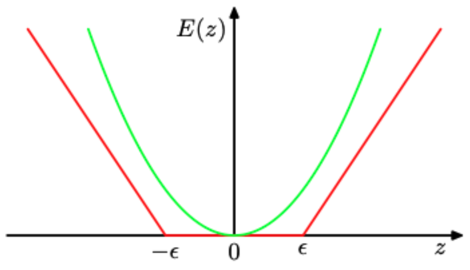


Figure 6: Plot of an ϵ-insensitive error function (in red) in which the error increases linearly with distance beyond the insensitive region. Also shown for comparison is the quadratic error function (in green).[4]

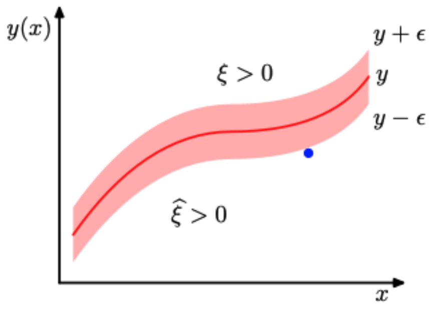


Figure 7: Illustration of SVM regression, showing the regression curve together with the ϵ-insensitive tube. Also shown are examples of the slack variables ξ and $\hat{\xi}$. Point above the ϵ-tube have $\xi > 0$ and $\hat{\xi} = 0$ points below the ϵ-tube have $\xi = 0$ and $\hat{\xi} > 0$, and points inside the ϵ-tube have $\xi = \hat{\xi} = 0$.[4]

SVR model with parameter kernel = 'linear' but scales better to large number of samples. Also, Support Vector Machine algorithms are not scale invariant. Consequently, the documentation strongly advises scaling the data. For instance, standardizing the data to have a mean of 0 and a variance of 1 is recommended. Accordingly, we employ the StandardScaler from the scikit-learn Python package to scale the data [10]. We focus on fine-tuning the ϵ-insensitive zone and keep all other hyperparameters at their default values. According to the scikit-learn documentation [10], the choice of an appropriate value for ϵ depends on the scale of the target variable. Since we standard scale all variables, we expect that low ϵ values would be appropriate. However, it must be added that we may expect overfitting for epsilon values of 0 or close to 0. We will explore a range of potential ϵ values (0.005, 0.010, 0.050, and 0.100) to train the linear SVR model. Following testing, the variables are rescaled back to their original representation, and we then compute the MAE for each configuration. Ultimately, we will select the ϵ value that yields the lowest MAE on the test set.

Results

Based on the data presented in Table 7, the linear SVR model achieves the lowest MAE when the ϵ value is set to 0.0. However, a linear SVR model without an epsilon value resembles a simple linear regression, resulting in comparable MAE. Nevertheless, we expect overfitting in this case. To assess the model's performance with a specific epsilon value, we compute other relevant metrics using ϵ set to 0.005 (as shown in Table 8).

Table 7: MAE's of linear SVR Model with different ϵ values

ϵ	MAE[s]
0.000	9139
0.005	117,226
0.010	251,212
0.050	1,328,388
0.100	2,676,987

The analysis for the linear SVR model followed a similar approach to the linear regression model, utilizing only one simulation for training. Although the linear SVR model with some epsilon performs less favorably than the simple linear model, which is easier to construct and interpret, it demonstrates that a more complex model might not be necessary in this context.
Additionally, as stated in section 2.3.4, our objective is to train the model incrementally with new simulation data, enabling it to learn from all simulations associated with the same type of semiconductor. Thus, we focused on elaborating the SGD Regressor model and, consequently, decided not to pursue further analysis with the linear SVR model. Instead, we selected the SGD Regressor model as our final and relevant choice.

Table 8: Metrics for 1 simulation

MAE [s]	MAE relative to lifetime	MAE relative to GT	RMSE [s]
117,226	0.001	0.034	118,399

R-squared	Overprediction rate	Underprediction rated
0.99	1.0	0.0

4 Application

4.1 Application Structure

After statistical data exploration and developing the prediction model, our task also involves creating a web interface with a dashboard that displays well-designed plots and provides estimates of the remaining lifetime of a transistor. Consequently, we designed a main dashboard for end users, offering two options on the homepage: the "Inference Results" page and the "Graphs" page 11.
The "Inference Results" page enables users to access predictions and inferences generated by the trained SGDRegressor Model. When landing on the page, the user is given the choice to click the "Fetch Results" button. Subsequently, the user will be presented with comprehensive metrics and an overview, including rainflow counts, predictions, status, and time to failure. Further details about this page's functionality will be explained in Chapter 3.4. To develop this application, we utilized the best-performing model (SGDRegressor)

described in Chapter 2.3.4, trained it, containerized it, and then uploaded it to a cloud platform. Although AWS was initially considered, we switched to the Google Cloud Platform due to time constraints.
The second option on the dashboard is the “Graphs” page, where users can select various configuration numbers and transistor numbers. Based on their input parameters, the corresponding plots are fetched from a dedicated AWS S3 bucket for plots. These plots were created beforehand using the plot generator module, which obtained time series data from the S3 bucket for transistor data, generated the necessary plots, and then uploaded them to the S3 bucket for plots. A schematic overview of the application structure can be seen in Figure 8.Further details about this workflow are provided in Chapter 3.3, offering a comprehensive description of the module's functionality.

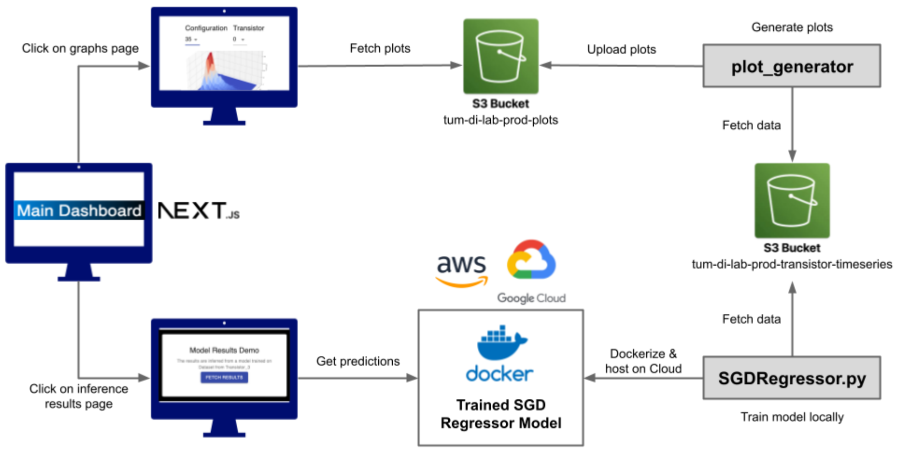


Figure 8: Schematic overview of the application structure

4.2 Modeling Workflow

The final Python program comprises two main classes: `DataPreprocessor` and `SGDRegressorFailurePredictor`. Each class fulfills distinct roles in the data processing and modeling pipeline. The `DataPreprocessor` class calculates and stores the rainflow counts for the dataset, while the `SGDRegressorFailurePredictor` class handles model training and predictions.

4.2.1 DataPreprocessor

The `DataPreprocessor` class preprocesses the semiconductor temperature data, computing rainflow counts (used as a feature for failure prediction) and time-to-failure data using sliding window calculations. The identifier differentiates datasets from different semiconductors, enabling preprocessing for multiple data sets, where it serves as a unique identifier for each dataset.
Rainflow counts and time-to-failure are calculated for each 'window' of data and stored in dictionaries (`rainflow-counts` and `time-to-failure`) using the dataset identifier as the key. If the data for a particular identifier is processed in multiple batches, the rainflow

counts for the new batch are added to the existing counts for that identifier so that we can refactor the data in real-time.

4.2.2 SGDRegressorFailurePredictor

The `SGDRegressorFailurePredictor` class utilizes the `DataPreprocessor` object to retrieve the preprocessed data. It employs an SGDRegressor model from Scikit-learn to predict the time-to-failure based on the rainflow count data. The identifier is used to select the appropriate data for each semiconductor. To evaluate the model's performance on unseen data, a train-test split is performed.
The separation of the two classes, `DataPreprocessor` and `SGDRegressorFailurePredictor`, follows good programming practice and enhances usability in the front-end. The `DataPreprocessor` object enables us to store intermediate results of rainflow counts. As new real-time data arrives for the same semiconductor, it can be aggregated with the previously saved rainflow counts for that dataset up to that point. This modular approach ensures that the `DataPreprocessor` class focuses on data preprocessing, while the `SGDRegressorFailurePredictor` class is dedicated to machine learning modeling. This separation results in code that is easier to read, maintain, and debug.
This design effectively separates data preprocessing from the regression model, although the calculation of rainflow counts and time-to-failure data from temperatures is conveniently encapsulated within a single class. The preprocessing is performed only once, and the preprocessed data is then passed to the FailurePredictor upon creation. This approach offers efficient handling and processing of large datasets by dividing the data into manageable batches for training the regression model and evaluating its performance.

4.3 Visualization Workflow

The final component of the project scope involves creating a user-friendly web interface that displays well-designed plots. To achieve this, we developed a specialized plot generator module. Initially, the module requires two input parameters: the simulation's configuration number and the transistor number. The time series data is stored in a dedicated S3 bucket named "tum-di-lab-prod-transistor-timeseries", organized in folders like "config33/transistor_0". Based on the input parameters, the module establishes a connection to the relevant S3 object, fetching the required data in the form of a zip file. The module then proceeds to process the data, unzipping the file and reading the temperature data from the CSV file. Once the data processing is complete, the module performs a rainflow analysis, generating two types of rainflow plots based on the obtained rainflow counts. These two plot types are the "Cycle Range vs. Time vs. Count Plot" and the "Cycle Range vs. Mean Stress vs. Count Plot", as discussed in Chapter 1.4. To achieve visually appealing results, we employ the Matplotlib Python package. The generated plots are subsequently uploaded to another dedicated S3 bucket named "tum-di-lab-prod-plots", organized in the same manner as the time series bucket. Eventually, when a user accesses the visualization page in the web interface, they can select the corresponding configuration and transistor numbers. This action triggers the fetching of the relevant two plots from the S3 bucket containing the plots.

A schematic overview of the current visualization workflow can be seen in Figure 9.

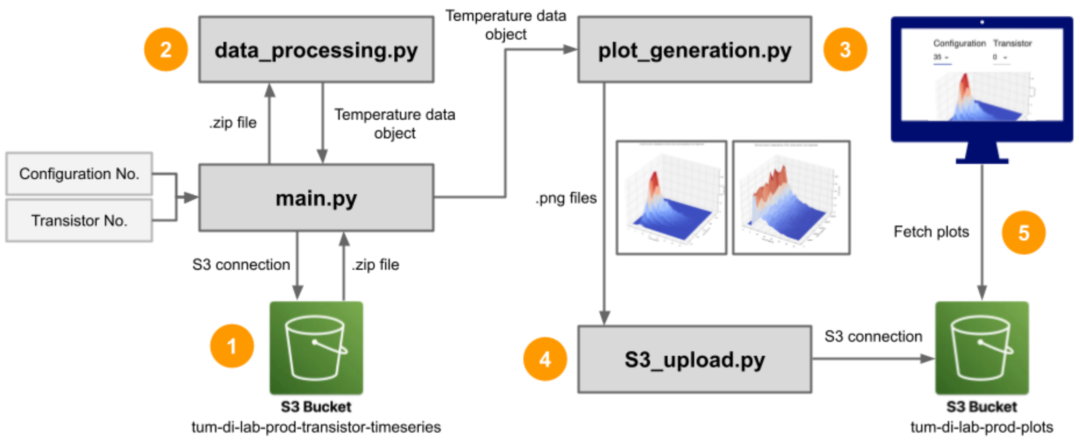


Figure 9: Schematic overview of the current visualization workflow

It's essential to highlight that the current workflow relies solely on the existing simulated time series data in the S3 buckets, consisting of 8 configurations, each with 10 transistors. Furthermore, in this initial implementation of the visualization module, the plot computation was carried out locally on our device. However, this preliminary version is static and lacks the ability to accommodate new data uploads and perform computations on cloud infrastructure. To improve and refine this workflow, we propose the following enhancements and developments:

1. **Enable Temperature Data Upload via User Interface:** The user interface should offer the capability to upload temperature time series data, which will be automatically saved to the timeseries bucket.

2. **Implementation on Cloud Infrastructure:** The entire process should be executed on cloud infrastructure. When data is uploaded, an AWS Lambda function should be triggered, responsible for waking up an EC2 instance. The Lambda function should pass the folder structure numbers (configuration and transistor number) to the EC2 instance.

3. **Automated Triggering of Plot Generation Module:** Upon receiving the parameters from the Lambda function, the EC2 instance will initiate the plot generation module. This module will process the new data, create the relevant plots, upload them to the designated S3 bucket, and display them on the web interface.

A schematic overview of the proposed future visualization workflow can be seen in Figure 10. By implementing these proposed changes, the workflow will become dynamic, allowing users to upload new data and visualize the plots seamlessly on the web interface. The utilization of cloud infrastructure will enable efficient and scalable processing of data and plot generation. These future enhancements are crucial steps towards optimizing the visualization module and ensuring its practical application in real-world scenarios.

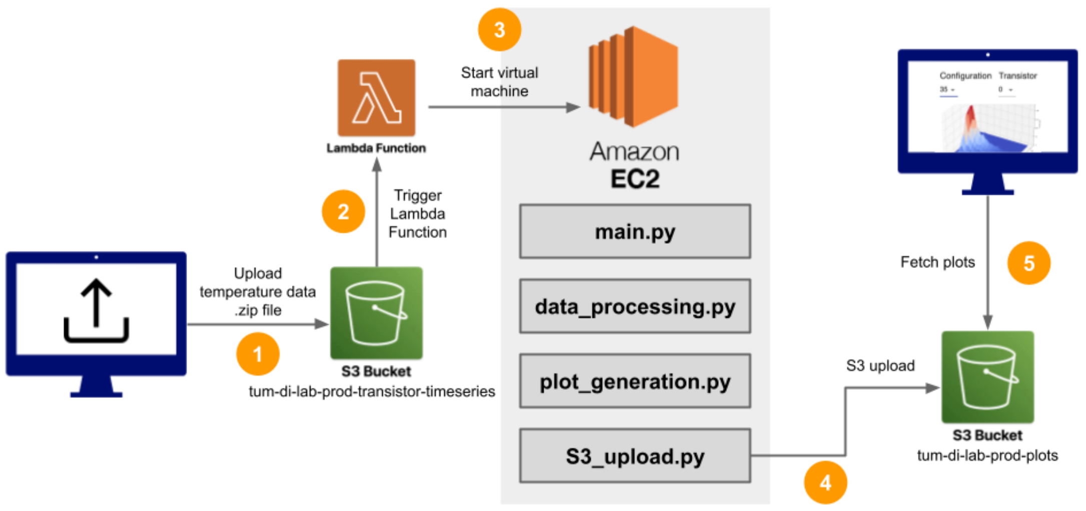


Figure 10: Schematic overview of the proposed future visualization workflow

4.4 Deployment and User Interface

As the final stage of the pipeline, we have implemented a model deployment code that allows the visualization of results through a front-end dashboard. This deployment code enables the model's use in a production environment, offering predictions through an API built with Flask, a lightweight web framework. The model's main objective is to predict the time to failure of a transistor using rainflow counts derived from temperature data. During our initial experimentation, the deployment model is trained with transistor data retrieved from S3 buckets. The trained model is then exported to desired formats such as .pb and .joblib for use during the inference stage. Python serves as the programming language for model training. However, conventional machine learning inference methods do not apply to this use case, as the model results depend on the entire dataset's rainflow counts. To address this challenge, we introduced an alternative approach utilizing the DataPreprocessor class. This class calculates rainflow counts and generates time to failure based on the given temperature data. The rainflow counts are computed using a sliding window approach, iterating over the temperature data to extract counts within each window. These counts are then accumulated to obtain a continuous count, enabling efficient processing of large datasets.

4.4.1 Dashboard

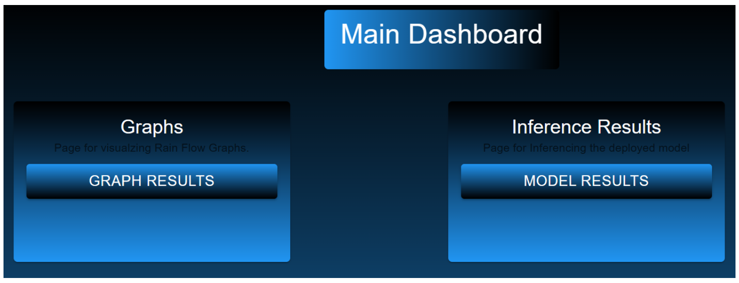
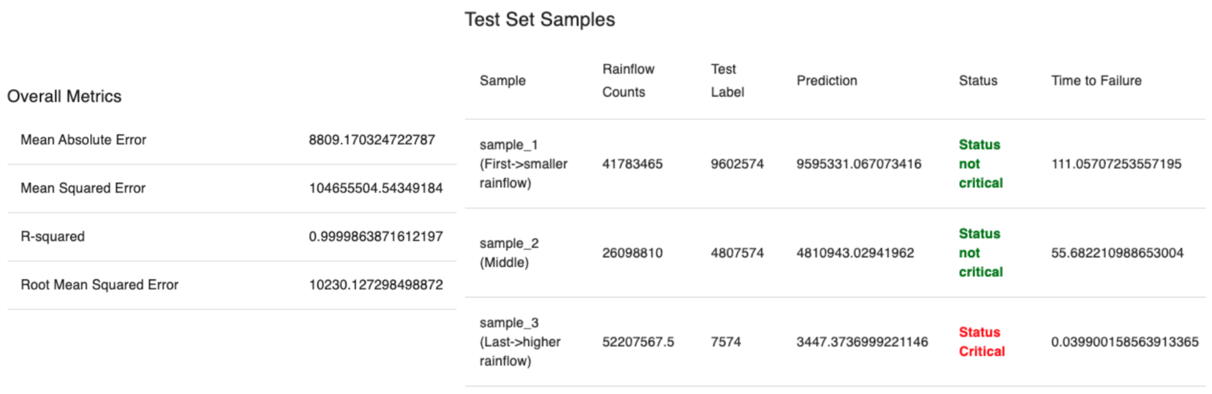

Main Dashboard
Graphs
Page for visualzing Rain Flow Graphs.
GRAPH RESULTS
Inference Results
Page for Inferencing the deployed model
MODEL RESULTS

Figure 11: Snapshot of Dashboard.

The front-end Dashboard of the application communicates with the model's API endpoint to retrieve predictions and present the results in a user-friendly manner. The deployment code provides an API endpoint at /predict, enabling users to make predictions using the trained Transistor Failure Prediction model. By sending a POST request with the necessary data, users can obtain the model's predictions, represented as time to failure [in days] and the corresponding status (critical or not critical). Furthermore, the user will be presented with comprehensive performance metrics (MAE, MSE, R^2 and RMSE) and an overview, including the sample used as a test set, the rainflow counts, test label and predictions on the test set [both in seconds] 12.



Overall Metrics

Mean Absolute Error	8809.170324722787
Mean Squared Error	104655504.54349184
R-squared	0.9999863871612197
Root Mean Squared Error	10230.127298498872

Test Set Samples

Sample	Rainflow Counts	Test Label	Prediction	Status	Time to Failure
sample_1 (First->smaller rainflow)	41783465	9602574	9595331.067073416	Status not critical	111.05707253557195
sample_2 (Middle)	26098810	4807574	4810943.02941962	Status not critical	55.682210988653004
sample_3 (Last->higher rainflow)	52207567.5	7574	3447.3736999221146	Status Critical	0.039900158563913365

Figure 12: Snapshot of Demo Results.

4.4.2 Cloud Deployment

The entire model and its dependencies are packaged and deployed as a Docker container. This container encapsulates the necessary Python libraries, the trained machine learning model, and the Flask application for serving the model. For efficient execution and scalability, we deployed the Docker container using Google Cloud Run, which provides a fully managed serverless environment. Cloud Run automatically scales the container based on incoming requests, ensuring continuous availability to handle user queries. Moreover, Cloud Run offers an API endpoint that allows external applications, including the frontend Dashboard, to interact seamlessly with the model.

4.4.3 Possible Future Works in Deployment

We propose the following future enhancements and developments to enhance the application's utility in real-world scenarios:

1. **Model Versioning:** Develop a versioning system for the deployed models to facilitate easy management and rollback to previous versions if needed.

2. **Automated Retraining:** Implement an automated retraining pipeline to periodically retrain the model on fresh data, ensuring it stays up-to-date and relevant.

These improvements aim to keep the deployed model up-to-date and easily manageable, thereby empowering the system to provide better results and improve the end-user experience.

5 Conclusion

In conclusion, this work tackles the important issue of thermomechanical fatigue leading to transistor failures. Through effective use of rainflow analysis and machine learning models, such as the SGD Regressor, we can now estimate the remaining lifetime of semiconductors, enabling predictive maintenance, cost optimization, and enhanced safety measures.

From initial statistical data exploration, we identified key features and data preprocessing needs, setting the stage for our modeling phase. This exploration highlighted the importance of rainflow counts for our predictive models.

The modeling phase of the project witnessed an exploration of various machine learning algorithms, including Trees based algorithms, Linear Regression, and SGD Regressor, each evaluated based on their predictive capabilities. However, after an in-depth comparison, the SGD Regressor emerged as the most viable option for our application due to its efficiency and performance in handling the large dataset. It gives the most reliable results with Linear Regression while it also handles the data efficiently because of its stochastic nature. Its performance on unseen dataset with MAE is 7058605 seconds (81 days) over 10 years period.

Our web application displays rainflow analysis and estimated lifetimes, alerting engineers when systems reach critical stages. Furthermore, the application displays the estimated remaining lifetime of semiconductors by using SGD Regressor, enabling practical implementation for real-world scenarios Utilizing Google Cloud Run, AWS, and Docker containers, our scalable deployment strategy ensures seamless operation across various environments.

This work contributes significantly to thermomechanical fatigue analysis, underlining the importance of predictive maintenance in key systems like wind turbines and electric vehicles. By aiding in failure prediction, operational reliability is improved and downtime minimized, driving a proactive maintenance approach.

As for future work, advanced techniques like Deep Learning and Semi-Supervised learning could further improve predictions. Also, more advanced methods or experimental methods can be used to calculate the ground truth. The scope of the work could be expanded to include more semiconductor components, and real-time data integration could make the system more dynamic. Finally, mobile and desktop versions of our application could improve accessibility and user experience.

References

[1] Markus Andresen, Giampaolo Buticchi, and Marco Liserre. Study of reliability-efficiency tradeoff of active thermal control for power electronic systems. *Microelectronics Reliability*, 58:119–125, 2016. Reliability Issues in Power Electronics.

[2] Mariette Awad and Rahul Khanna. *Support Vector Regression*, pages 67–80. Apress, Berkeley, CA, 2015.

[3] William T. Becker and Roch J. Shipley. *Failure Analysis and Prevention.* ASM International, 01 2002.

[4] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics).* Springer-Verlag, Berlin, Heidelberg, 2006.

[5] Alexei Botchkarev. A new typology design of performance metrics to measure errors in machine learning regression algorithms. *Interdisciplinary Journal of Information, Knowledge, and Management*, 14:045–076, 2019.

[6] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: with Applications in R.* Springer, 2013.

[7] Quentin Klopfenstein and Samuel Vaiter. Linear support vector regression with linear constraints. *Machine Learning*, 110(7):1939–1974, 2021.

[8] Milton A. Miner. Cumulative Damage in Fatigue. *Journal of Applied Mechanics*, 12(3):A159–A164, 03 2021.

[9] Osval Antonio Montesinos López, Abelardo Montesinos López, and Jose Crossa. *Support Vector Machines and Support Vector Regression*, pages 337–378. Springer International Publishing, Cham, 2022.

[10] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[11] Hansheng Ren, Qi Zhang, Bixiong Xu, Yujing Wang, Chao Yi, Congrui Huang, Xiaoyu Kou, Tony Xing, Mao Yang, and Jie Tong. Time-series anomaly detection service at microsoft. pages 3009–3017, 07 2019.

[12] ASTM Standard et al. Standard practices for cycle counting in fatigue analysis. *ASTM E1049-85, Am. Soc. Test. Mater. West Conshohocken, PA*, 1997.

[13] Vladimir N. Vapnik. *The nature of statistical learning theory.* Springer-Verlag New York, Inc., 1995.

[14] P.J. Vernon and T.J. Mackin. Fatigue failure analysis of a leg press exercise machine. In D.R.H. JONES, editor, *Failure Analysis Case Studies II*, pages 255–266. Pergamon, Oxford, 2001.

[15] Cort J Willmott and Kenji Matsuura. Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance. *Climate research*, 30(1):79–82, 2005.