# Introduction

This study aims to prevent unexpected system failure by estimating the remaining useful life (RUL) using predictive analytics derived from multiple sensors that continuously monitor the system's degradation process.

Although turbojet aircraft engines were the focus of this study, the methods and models can be applied to other industries that face similar system scheduling and maintenance problems.

Any business that wants to reduce downtime and improve operational efficiencies by modelling system degradation.

Reliably estimating system health and RUL has the potential for considerable operational savings by:

* avoiding unscheduled maintenance and downtime
* increasing equipment usage and efficiency
* avoiding unexpected system failures
* increasing operational safety

## EDA

### Dataset Overview

There are four datasets of increasing complexity (see below). Each dataset consists of three files:

1. Training set
2. Test set
3. RUL set (actual RUL of test set)

For this study's purpose, we will focus on the dataset FD001. This is because FD001 is the least complex, and we wanted to develop a solid understanding of the data before proceeding to the more complex datasets.

Each dataset includes simulations of multiple turbofan engines over time until a fault develops. The engine degradation simulation was carried out using NASA's Modular Aero-Propulsion System Simulation (C-MAPSS) software. Simulation data is used as real run-to-failure data for turbofan engines would be far too costly.

Each engine starts with different degrees of initial normal wear and manufacturing variation, which is unknown. The engine starts in a normal state of operation before developing a fault at some point. The presence of noise further complicates this data.

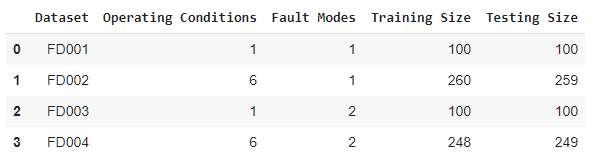


Table 1: Datasets overview

### Sensor Descriptions

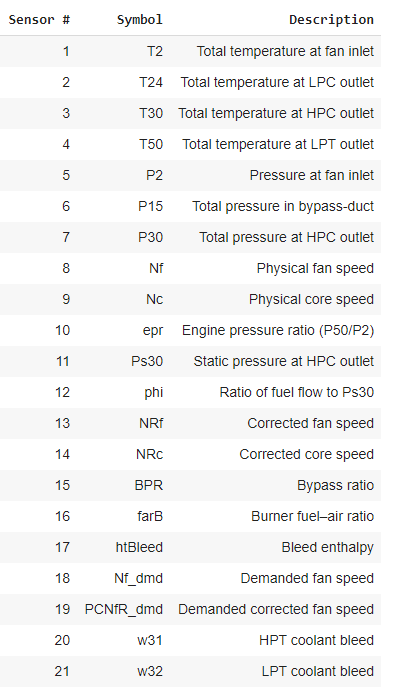
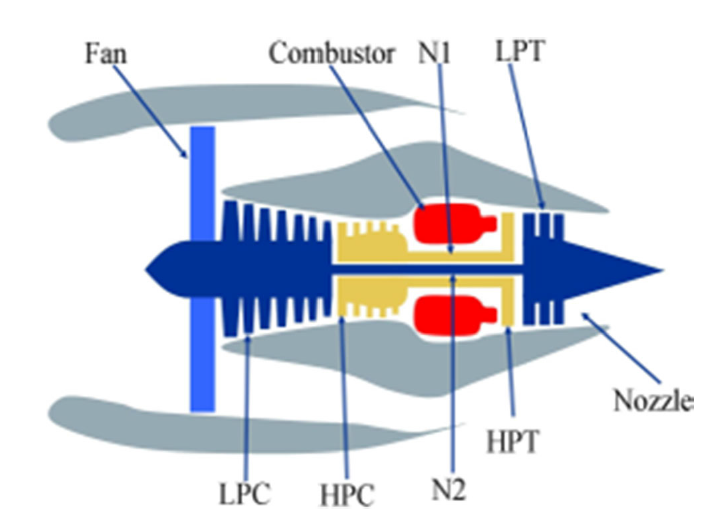


Figure 1: Turbofan diagram

Table 2: Sensor Descriptions

### Data Inspection

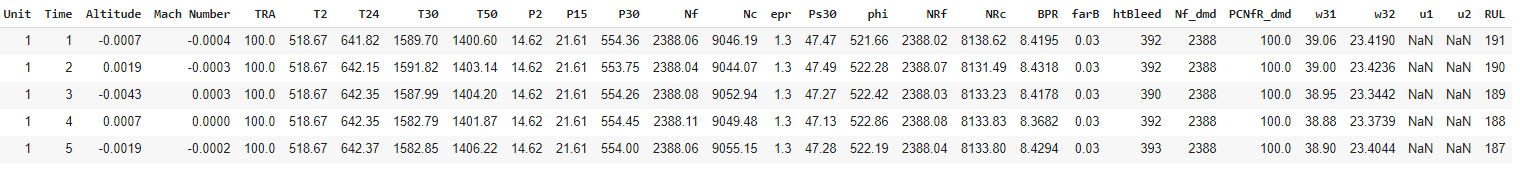
Our first step was to create a new column, ‘RUL’, that takes the max cycle time from each engine Unit # and counts down. The last ‘RUL’ value for each engine Unit # is the equivalent remaining cycles.

Table 3: Data Sample

Then we checked to see if there were any duplicate samples, which there were none.

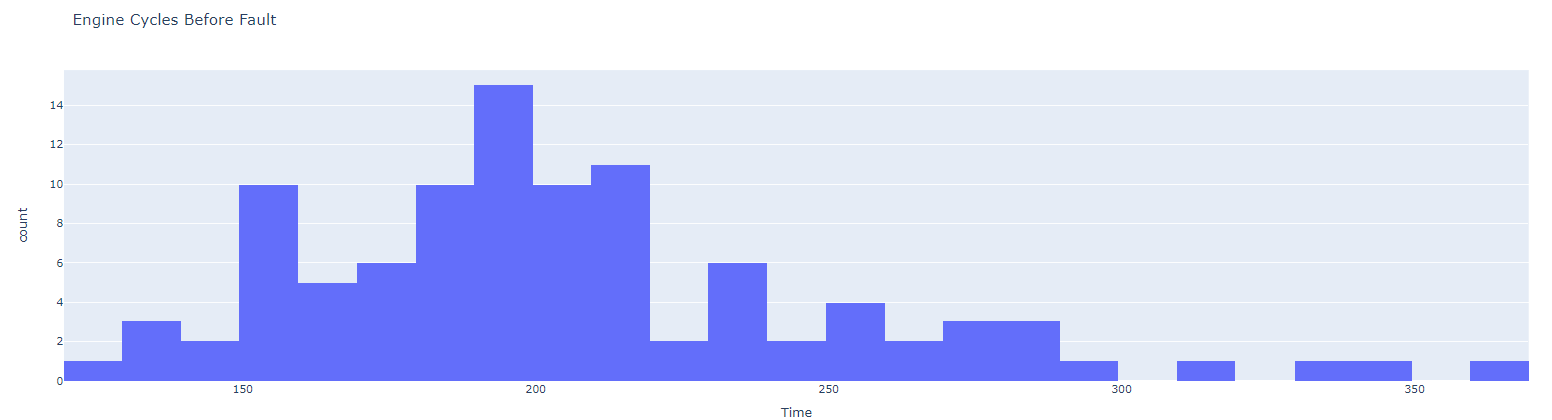
Next, we wanted to get a better idea of how many cycles the engines were running before generating a fault.

Figure 2: Engine cycles before fault for training data

We can see from Figure 2 that the majority of the engine units will run between 150-220 cycles before developing a fault.

Missing values can create some problems down the road so we addressed that early on by performing a quick check. There were two columns, u1 and u2, that had 100% missing values so those columns were dropped.

For our test data, we merged the test1 and RUL1 files. We also created the same ‘RUL’ column that we did for our training data earlier as well as a couple other columns:

* ‘Inv\_time: which took the difference between the min and max ‘RUL’ and counted down
* ‘RUL\_min: the minimum RUL for that specific engine unit ID.

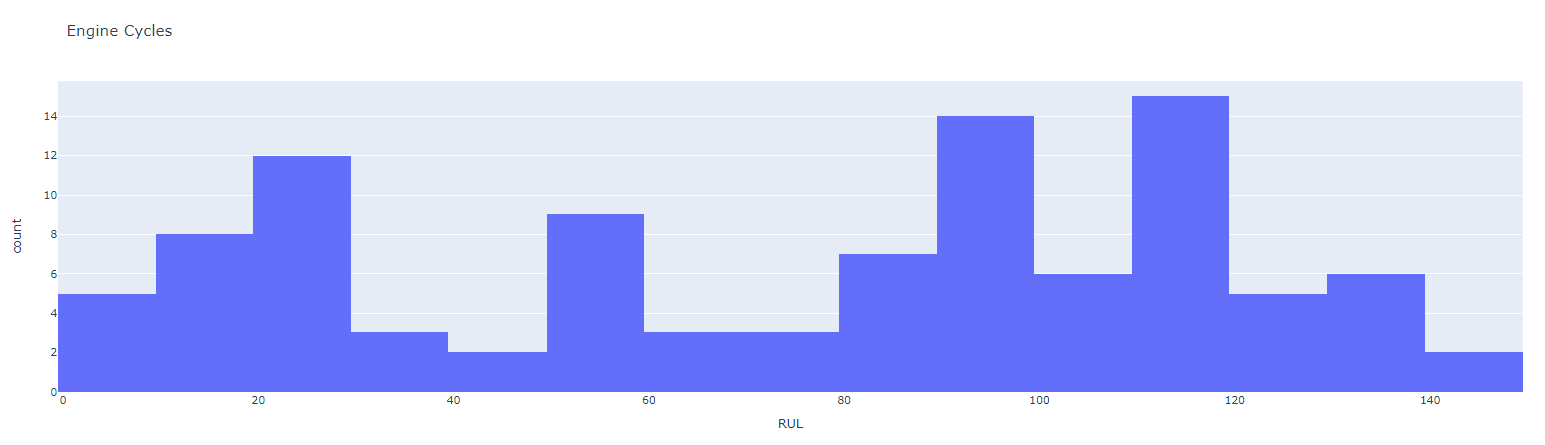
We plotted the engine cycles before fault for our test data as well.

Figure 3: Engine cycles before fault for test data

You can see that our test data has a significantly different distribution. This is because our test engines were not run until failure or fault. Instead, the engines were run for a number of predetermined cycles that we would then have to calculate an RUL for.

### Data Analysis

We dropped our missing value columns which left us with the following dataframe dimensions:

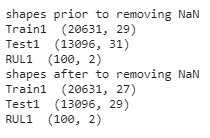


Figure 4: Dataframe dimensions

### Sensor Values vs Time

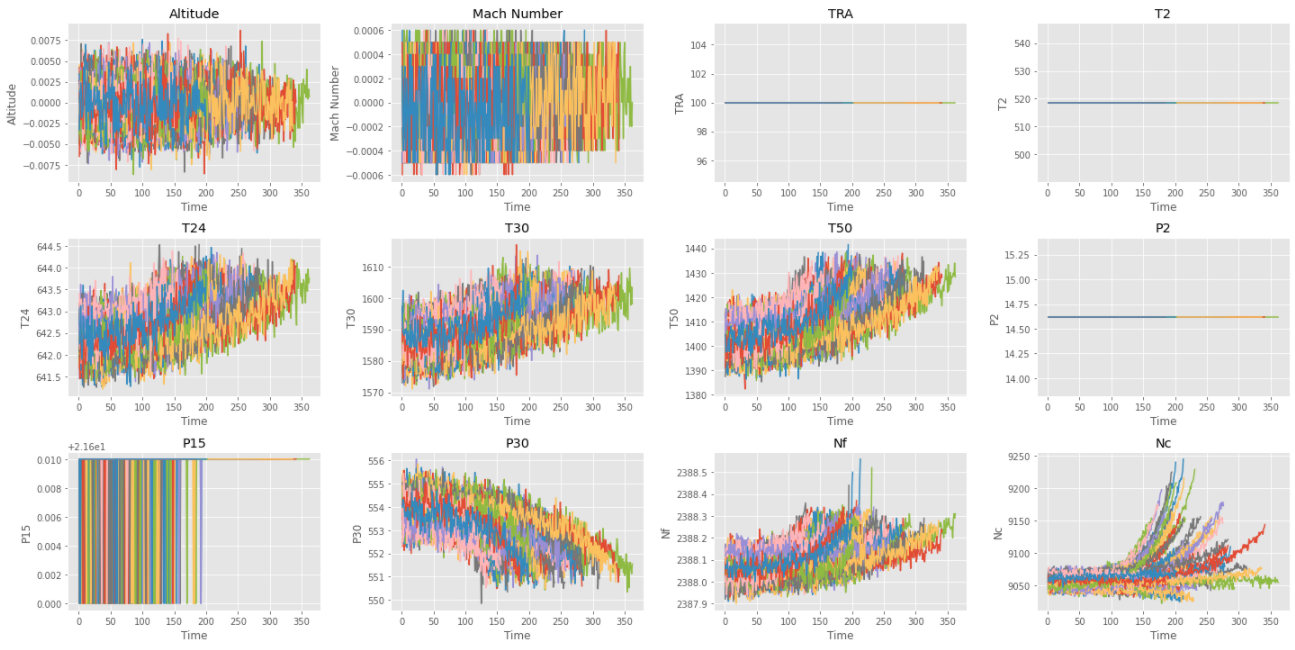
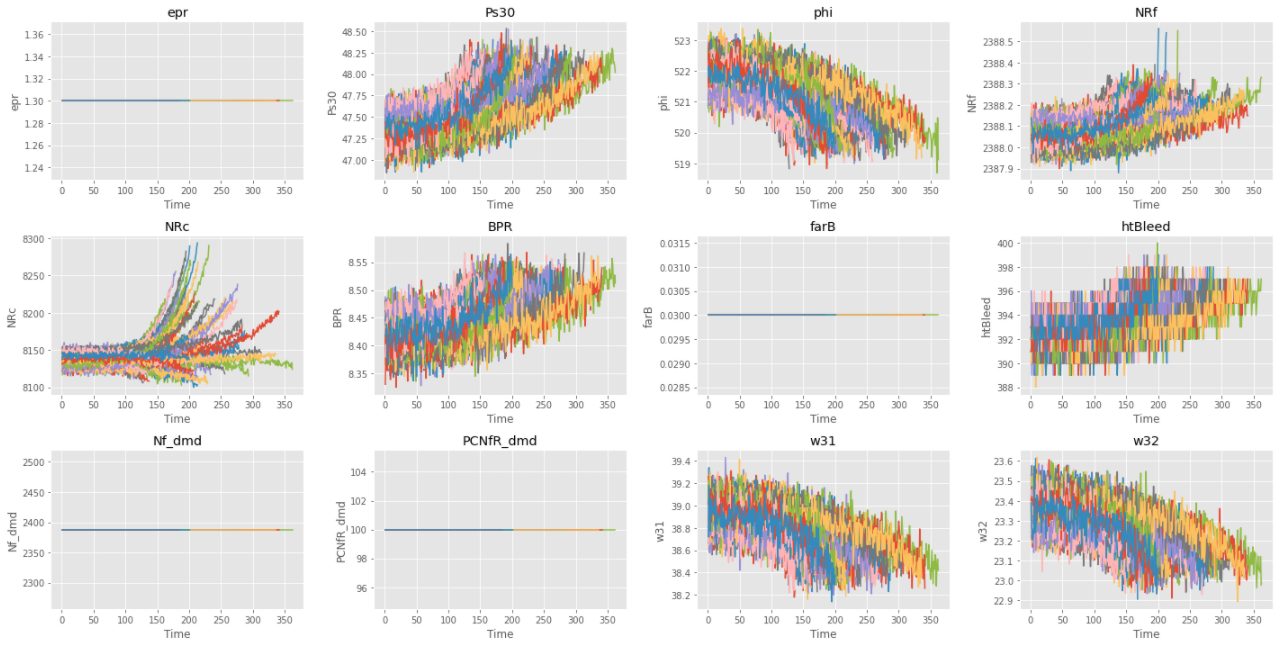
The following figures give us a better overview of our data and sensor distribution over time.

Figure 5: Sensor values vs time

Some of the sensors show a clear increase in decay rate towards an eventual fault. From the above plots, we can see several sensors that likely do not add any valuable insights. Also, the three operating conditions should be constant, as this is a lab simulation. As such, we will drop the following sensors:

* Altitude
* Mach Number
* TRA (throttle reverse angle)
* T2 - temperature at fan inlet
* P2 - pressure at fan inlet
* P15 - pressure in bypass-duct
* epr - engine pressure ratio (P50/P2)
* farB - burner fuel-air ratio
* Nf\_dmd -demanded fan speed
* PCNfR\_dmd - demanded corrected fan speed
* Nf - physical fan speed
* Nc - physical core speed

The controlled lab setting can explain most of the data observed from these sensors (ie: constant temperature, pressure, demanded speed, etc). The decision was made to drop Nf and Nc, and keep the corrected sensors NRf and Nrc.

As T30 (HPC outet temp) increases, P30 (HPC outlet pressure) drops, and Ps30 (HPC outlet static pressure) increases until a fault is triggered.

After dropping the columns listed above, our new training dataframe looks like this:

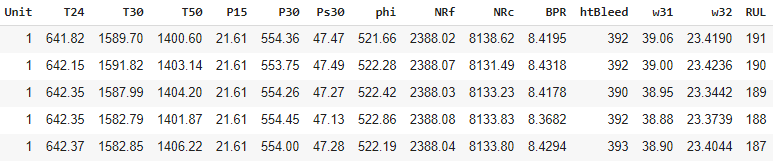


Table 4: Training dataframe

### Bivariate Analysis

A correlation matrix was generated to identify related features as well as multicollinearity.

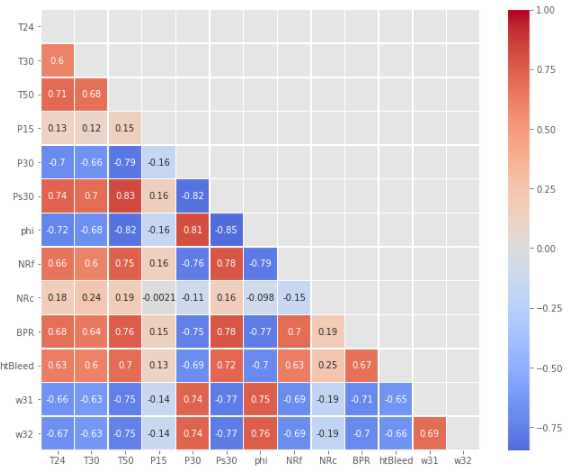
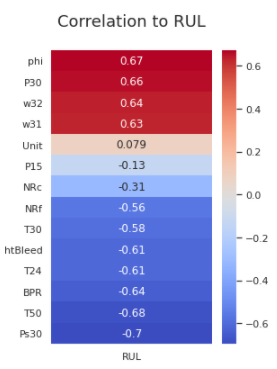


Figure 7: Correlation strength to RUL

Figure 6: Correlation matrix to RUL

It is important to note that the correlation coefficient only measures linear correlations. It may completeley miss out on nonlinear relationships. In the figure below, the plots along the bottom row all have a correlation coefficient equal to 0, despite the fact that their axes are clearly not independent. These are examples of nonlinear relationships.

## Decision Tree Classifier

Although decision trees may not be the fanciest algorithms out there, they are still a useful tool to visualize the decision-making process. They provide insights into which features are dominating the decisions and results.

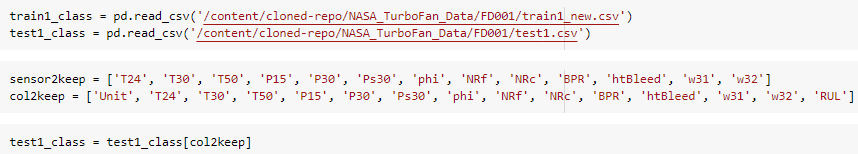
In this example, we will try to predict 5 different classes, with the emphasis on those that are closer to the actual RUL, as that aligns with our business objective.

### Import Data

We will import the dataframes that we cleaned up from the EDA section.

Figure 8: Data Import

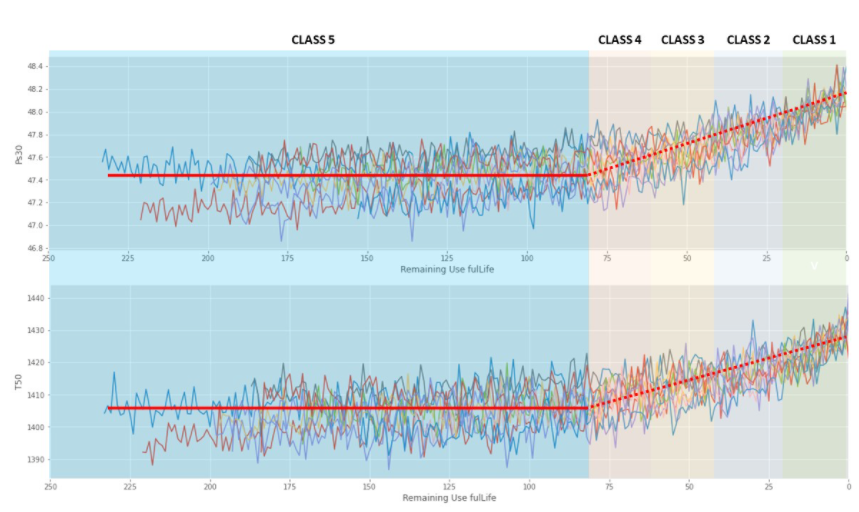
### Data Visualization



We will plot every 10th unit from the two most strongly correlated sensors, Ps30 and T50.

Figure 9: Sensor plot for every 10th engine unit

### Label Data Discretization - Creating RUL Classes for train1 and test1

The initial decision tree had 10 classes that were evenly distributed throughout the engine life cycle and, the model had a difficult time distinguishing between classes 5 to 10. We will try focusing on when the degradation accelerates.

* class 1 = RUL 0-20
* class 2 = RUL 20-40
* class 3 = RUL 40-60
* class 4 = RUL 60-80
* class 5 = RUL > 80

Figure 10: Sensor plot for every 10th engine unit with classes

### RUL class distribution

Figure 11: RUL class distribution

We can see from Figure 11 that the majority of our samples fall into class 5, six times more than any other class, resulting in an unbalanced dataset. Since we made a RULclass column, we will also have to drop the RUL column or the algorithm will simply use that to predict the RULclass. We will also drop the Unit column as that could lead to invalid predictions. We will also drop the columns that we discovered, from the EDA section, will likely provide little insight.

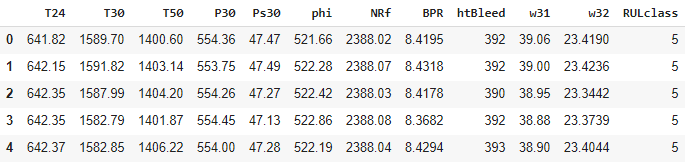


Table 5: train1\_class dataframe

### Imbalanced Dataset

To tackle the imbalanced dataset we used the Imbalanced-learn library (<https://imbalanced-learn.org/>), specifically the ADASYN algorithm. Here is a brief summary of how ADASYN, or adaptive synthetic sampling approach for imbalanced learning, is implemented:

1. calculate the ratio of minority to majority examples
2. calculate the total number of synthetic data to generate
3. find the k-Nearest neighbors of each minority example and calculate the rᵢ value. After this step, each minority example will be associated with a different neighborhood. The rᵢ value indicates the dominance of the majority class in each specific neighborhood. Higher rᵢ neighbourhoods contain more majority class examples and are more difficult to learn.
4. Normalize the rᵢ values so that the sum of all rᵢ values equals to 1.
5. Calculate the amount of synthetic examples to generate per neighbourhood. Because rᵢ is higher for neighbourhoods dominated by majority class examples, more synthetic minority class examples will be generated for those neighbourhoods. Hence, this gives the ADASYN algorithm its adaptive nature; more data is generated for “harder-to-learn” neighbourhoods.
6. Generate Gᵢ data for each neighbourhood. First, take the minority example for the neighbourhood, xᵢ. Then, randomly select another minority example within that neighbourhood, xzᵢ.

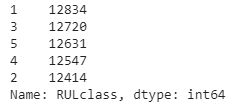
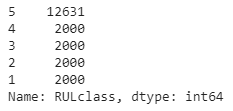


Table 6: Class distribution before (left) and after (right) ADASYN

### Feature Scaling

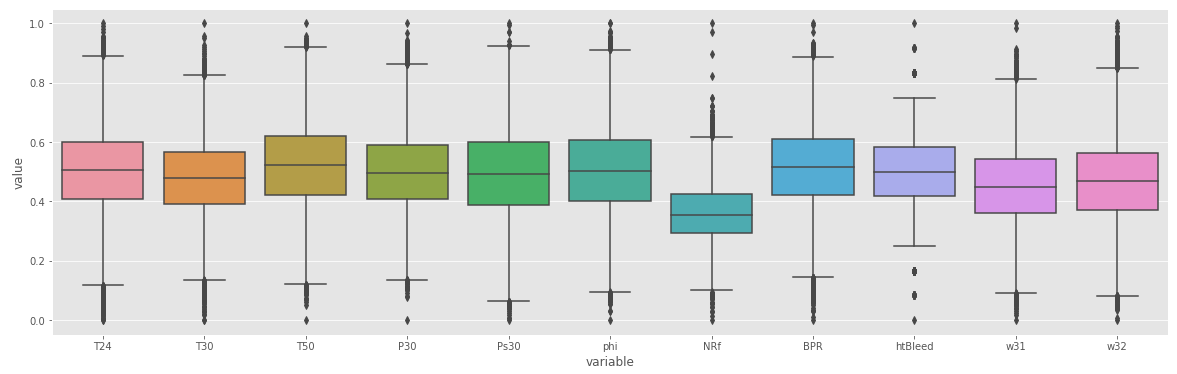
Since some of our sensor data does not follow a normal, or Gaussian distribution, we will apply min-max scaling.

Figure 12: Training data after min-max scaling to 0-1

### Train Initial Classification Tree

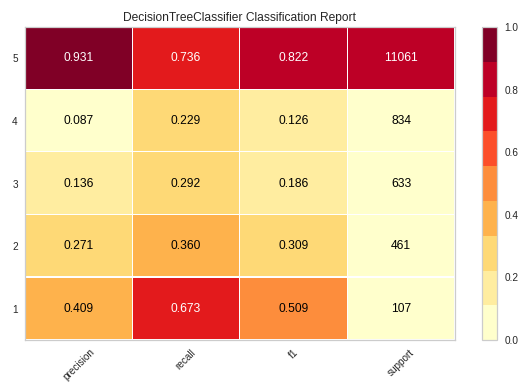
We will train an initial model that will serve as a baseline as we tune various parameters.

Figure 13: Initial model evaluation

Looking at Figure 13, the bottom row is of particular interest. For our business objective want to focus on Class 1, and in this case maximizing the recall score.

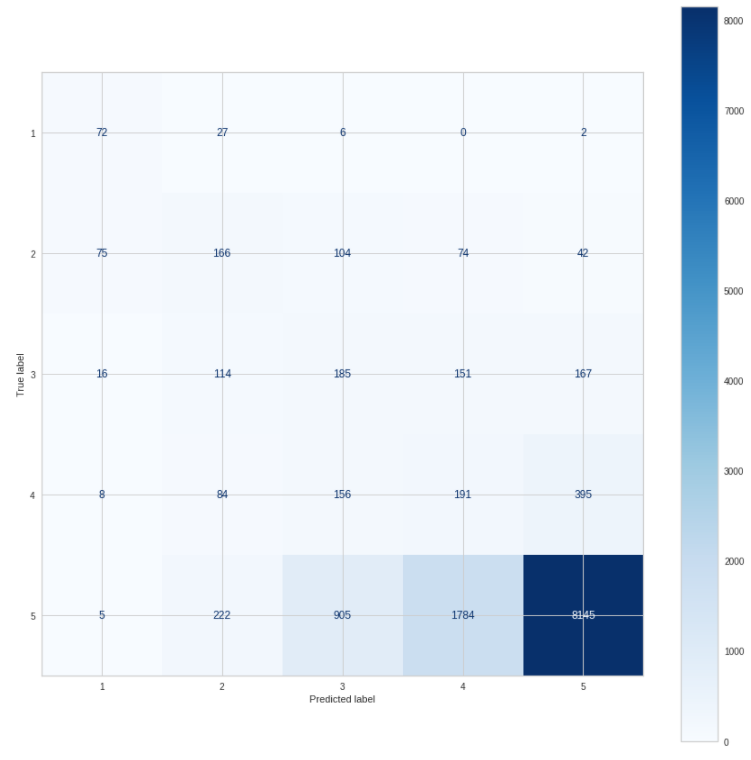


Figure 14: Initial model confusion matrix

The confusion matrix from our initial model shows 72 samples corrected classified as Class 1 with 35 late predictions of Class 2 or greater. We want to minimize the number of late predictions and will look at applying an asymmetric loss function that penalizes late predictions more severely later.

### Tune Initial Tree

To improve our initial model, we performed a GridSearch to optimize a few parameters with the following results:

* Criterion: Gini
* Max\_depth: 9
* Min\_samples\_split: 1
* Min\_samples\_leaf: 2

We retrained our model with the above parameter values with the following results.

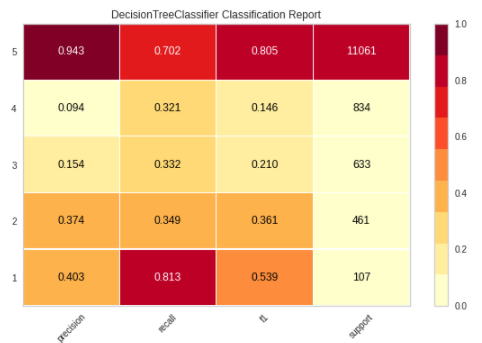


Figure 15: Model evaluation after GridSearch

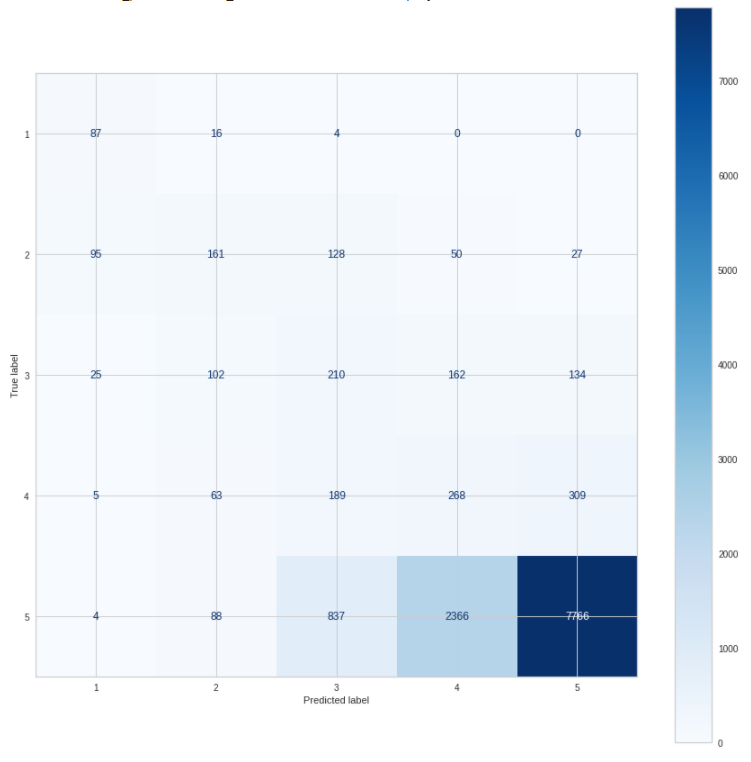


Figure 16: Confusion matrix after GridSearch

From the above metrics, we observe a 14% improvement in recall. This is also evident in the confusion matrix, where before the GridSearch we had 35 late predictions, now we have 20.

### Tune Class Weights

To tackle the imbalanced dataset we are first going to try Scikit-Learn's *class\_weight:"balanced"* parameter. The "balanced" mode uses the values of y to automatically adjust weights so they are inversely proportional to the class frequencies in the input data. We will use the same parameters from our previous GridSearch.

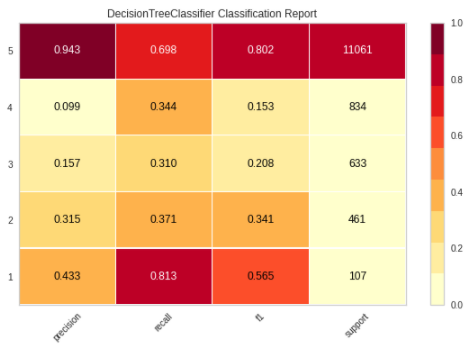


Figure 17: Model evaluation after tuning class weights

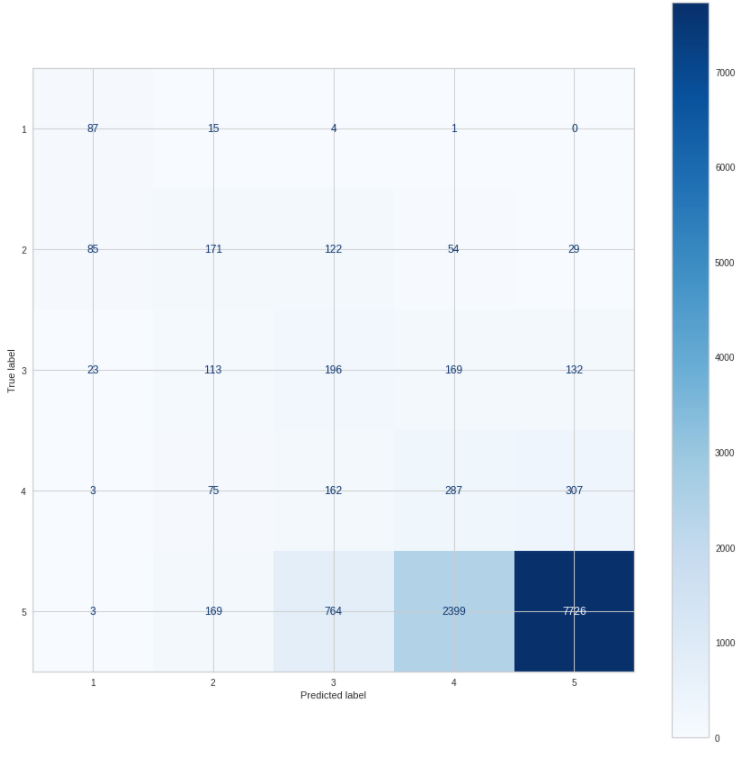


Figure 18: Confusion matrix after tuning class weights

After applying the “balanced” weight class to our model we saw a slight improvement to our precision while our recall remained unchanged. The confusion matric is also very similar, with the exception of 1 very late, Class 4, prediction that we did not have before.

### Dimensionality Reduction

We will explore four different dimensionality reduction techniques:

* PCA – Principal Component Analysis:
  + PCA works by identifying the hyperplane that lies closest to the data and then projects the data on that hyperplane while retaining most of the dataset variation.
* t-SNE – T-distributed stochastic neighbor embedding:
  + (t-SNE) takes a high dimensional data set and reduces it to a low dimensional graph that retains a lot of the original information. It does so by giving each data point a location in a two or three-dimensional map. This technique finds clusters in data, thereby ensuring that an embedding preserves the meaning in the data. t-SNE reduces dimensionality while keeping similar instances close and dissimilar instances apart.
* LDA – Linear Discriminant Analysis:
  + Linear Discriminant Analysis (LDA) is most commonly used as a dimensionality reduction technique in the pre-processing step for pattern-classification. The goal is to project a dataset onto a lower-dimensional space with good class-separability to avoid overfitting and reduce computational costs.
* UMAP – Uniform Manifold Approximation and Projection:
  + UMAP is a nonlinear dimensionality reduction method; it helps visualize clusters or groups of data points and their relative proximities.
  + UMAP is similar to t-SNE but with probably higher processing speed, therefore, faster and better visualization.

First, we created a new dataframe from the original train1\_class dataframe. Then, we took a representative sample for visualization purposes using Scikit\_Learn stratified sample split, 10%.

There are a lot of interesting visuals in this section and I recommend you view them in the notebook. The dimensionality reduction techniques were performed on the previous 5 Class models and compared to a 10 Class models that was created earlier (see notebook).

When comparing the PCA between the 10 and 5 class trees, the 10 class PCA shows a better cluster separation for class 1. When we look at the 3D plot of the 5 class PCA, some planes run along the y-axis at a 45-degree angle. These planes don't appear to reflect the data; they are likely from the sub-sampling we did. Overall, PCA did not create well-defined clusters for our classes.

t-SNE (T-distributed stochastic neighbour embedding) finds clusters in data to reduce dimensionality while keeping similar instances close and dissimilar instances apart. As with PCA, t-SNE failed to generate any well-defined clusters for our classes. In our example, the 5 class t-SNE plot appeared to have a more uniform distribution than the 10 class plot. The 10 class plots had more of a wave function shape, whereas the 5 class plots were more of a flattened bowl.

LDA (Linear Discriminant Analysis) is similar to PCA, except that rather than finding the component axes that maximize the variance of our data, we are also interested in the axes that maximize the separation between our classes. There becomes evident when we look at our plot. The 10 and 5 class plots show better class separation than the previous reduction techniques, especially the 5 class plots. The PCA plots showed some separation on the extremities, but the rest of the classes were all mixed, something that our class prediction error plot also showed. Again, this makes sense when we look at our sensor data and see the decay rate increases as the RUL decreases. Our model has a tough time distinguishing classes between the extremities because the data does not have a lot of variation. The values between class 1 and class 5, for example, are easy to distinguish between. Still, the values between classes 2-4 are more similar, and it becomes difficult to find a clear definition or separation between the class boundaries.

UMAP (Uniform Manifold Approximation and Projection) is a nonlinear dimensionality reduction technique that effectively visualizes clusters and their relative proximities. UMAP is similar to t-SNE but with higher processing speeds. When we look at the 10 and 5 class plots for UMAP we see a similar outcome to the t-SNE: nonlinear shapes with minimal class separation. Again, the extremity classes are better defined than those in between.

For our dataset, LDA performed the best, and we will use it going forward as we adjust our model parameters and assumptions.

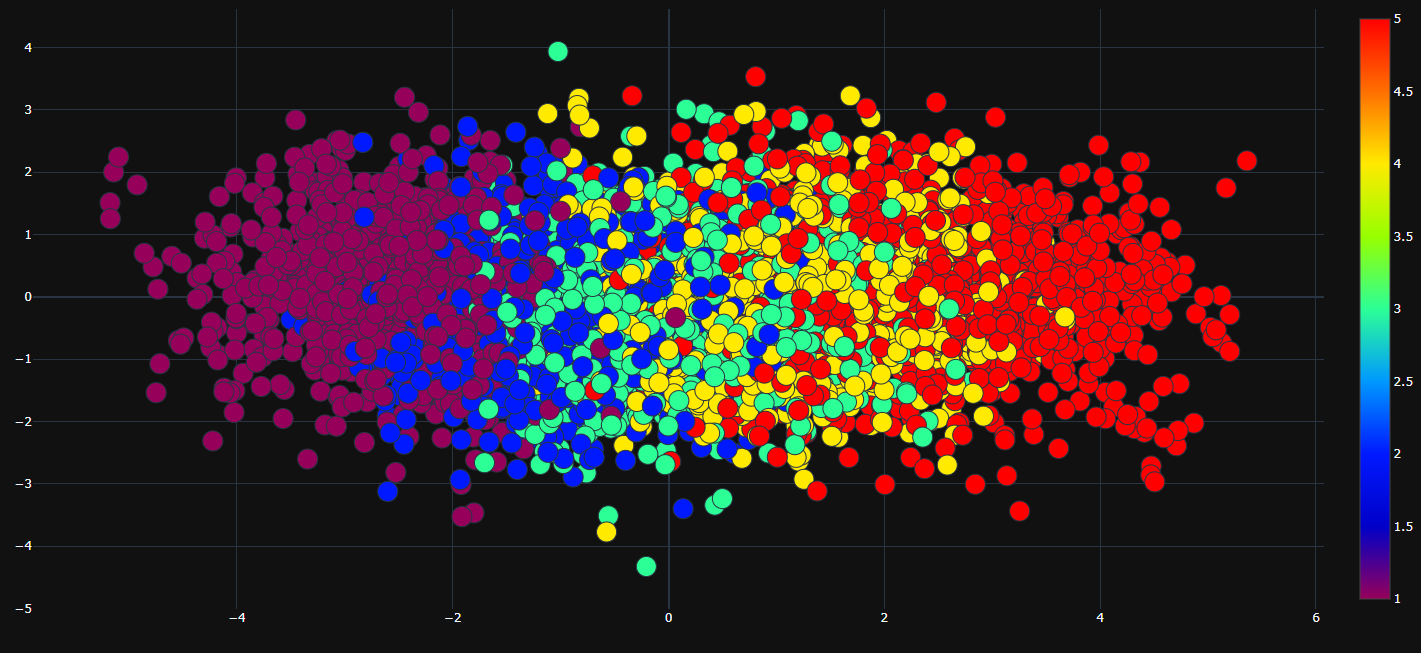
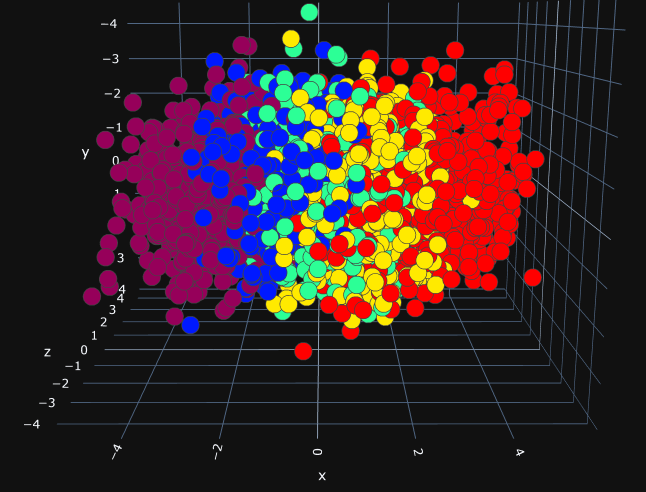


Figure 19: 3D LDA

Figure 20: 2D LDA

### Decision Tree Summary

With the imbalanced nature of our dataset, in combination with our business objective of preventing engine failures, Class 1 recall was the most suitable metric to evaluate our model.

Our initial baseline model had the following scores:

* precision: 0.409
* recall: 0.673
* f1: 0.509

Following the initial model, we tuned the following parameters via a GridSearch:

* Criterion: Gini
* Max\_depth: 9
* Min\_samples\_split: 1
* Min\_samples\_leaf: 2

Which resulted in a tree with the following scores:

* precision: 0.403
* recall: 0.813
* f1: 0.539

The final tuning step we took was to use Scikit-Learn's *class\_weight: "balanced"*parameter to account for the imbalanced data, which resulted in the following scores:

* precision: 0.433
* recall: 0.813
* f1: 0.565

Our recall improved by 14% from our original baseline model.