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

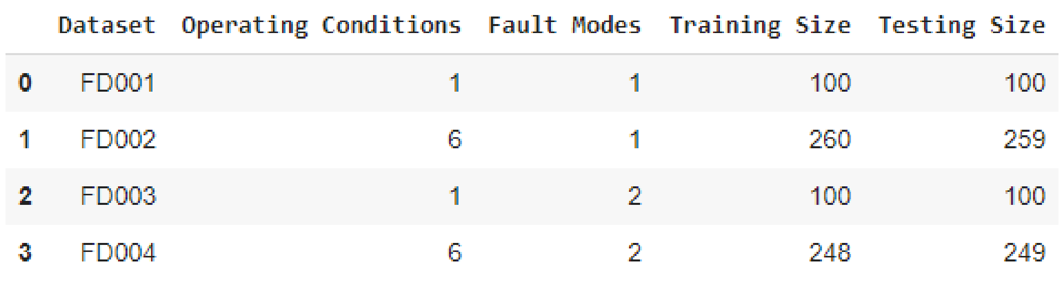
EDA

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.



	Dataset	Operating Conditions	Fault Modes	Training Size	Testing Size
0	FD001	1	1	100	100
1	FD002	6	1	260	259
2	FD003	1	2	100	100
3	FD004	6	2	248	249

Table 1: Datasets overview

Sensor Descriptions

Sensor #	Symbol	Description
1	T2	Total temperature at fan inlet
2	T24	Total temperature at LPC outlet
3	T30	Total temperature at HPC outlet
4	T50	Total temperature at LPT outlet
5	P2	Pressure at fan inlet
6	P15	Total pressure in bypass-duct
7	P30	Total pressure at HPC outlet
8	Nf	Physical fan speed
9	Nc	Physical core speed
10	epr	Engine pressure ratio (P50/P2)
11	Ps30	Static pressure at HPC outlet
12	phi	Ratio of fuel flow to Ps30
13	NRf	Corrected fan speed
14	NRc	Corrected core speed
15	BPR	Bypass ratio
16	farB	Burner fuel–air ratio
17	htBleed	Bleed enthalpy
18	Nf_dmd	Demanded fan speed
19	PCNfR_dmd	Demanded corrected fan speed
20	w31	HPT coolant bleed
21	w32	LPT coolant bleed

Table 2: Sensor Descriptions

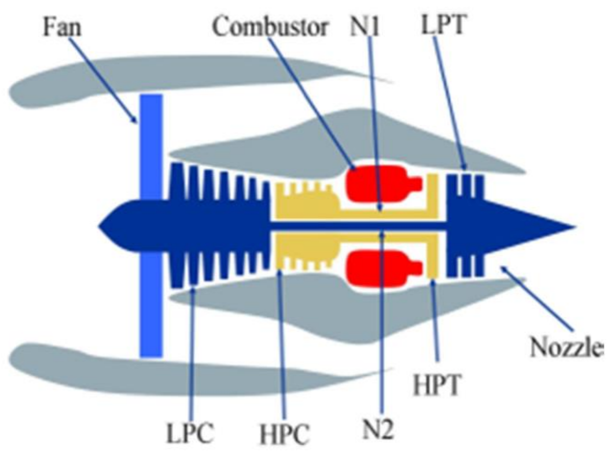


Figure 1: Turbofan diagram

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.

Unit	Time	Altitude	Mach Number	TRA	T2	T24	T30	T50	P2	P15	P30	Nf	Nc	epr	Ps30	phi	NRf	NRc	BPR	farB	htBleed	Nf_dmd	PCNfR_dmd	w31	w32	u1	u	RUL
1	1	-0.0007	-0.0004	100.0	518.67	641.82	1589.70	1400.60	14.62	21.61	554.36	2388.06	9046.19	1.3	47.47	521.66	2388.02	8138.62	8.4195	0.03	392	2388	100.0	39.06	23.4190	NaN	Na	191
1	2	0.0019	-0.0003	100.0	518.67	642.15	1591.82	1403.14	14.62	21.61	553.75	2388.04	9044.07	1.3	47.49	522.28	2388.07	8131.49	8.4318	0.03	392	2388	100.0	39.00	23.4236	NaN	Na	190
1	3	-0.0043	0.0003	100.0	518.67	642.35	1587.99	1404.20	14.62	21.61	554.26	2388.08	9052.94	1.3	47.27	522.42	2388.03	8133.23	8.4178	0.03	390	2388	100.0	38.95	23.3442	NaN	Na	189
1	4	0.0007	0.0000	100.0	518.67	642.35	1582.79	1401.87	14.62	21.61	554.45	2388.11	9049.48	1.3	47.13	522.86	2388.08	8133.83	8.3682	0.03	392	2388	100.0	38.88	23.3739	NaN	Na	188
1	5	-0.0019	-0.0002	100.0	518.67	642.37	1582.85	1406.22	14.62	21.61	554.00	2388.06	9055.15	1.3	47.28	522.19	2388.04	8133.80	8.4294	0.03	393	2388	100.0	38.90	23.4044	NaN	Na	187

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.

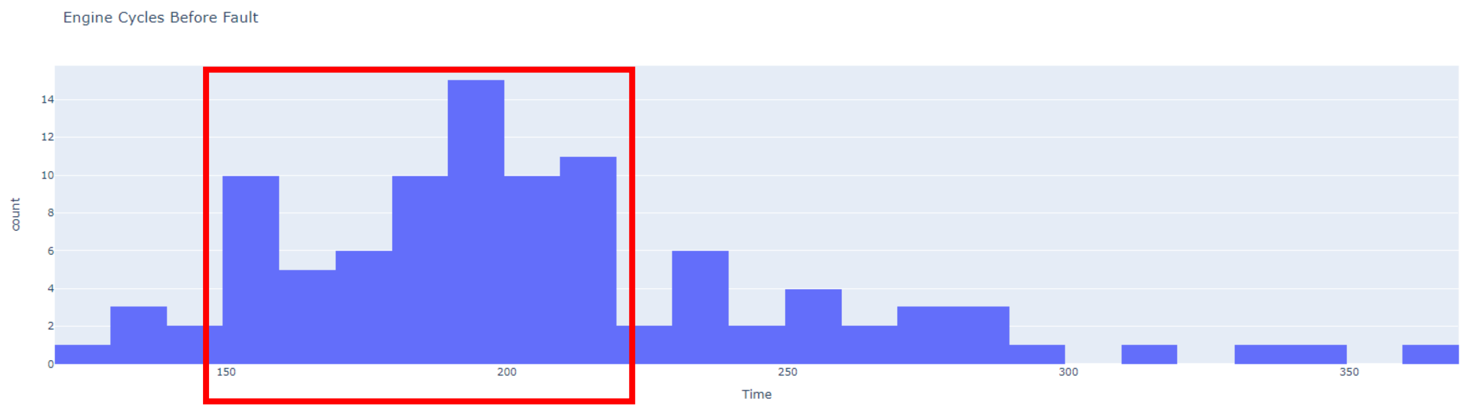


Figure 2: Engine cycles before fault for training data

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.

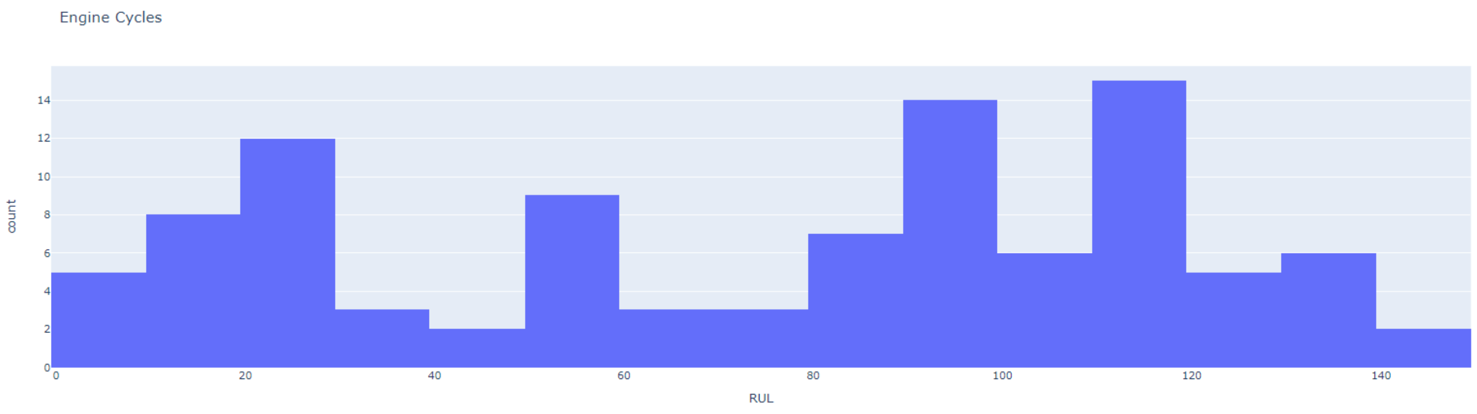
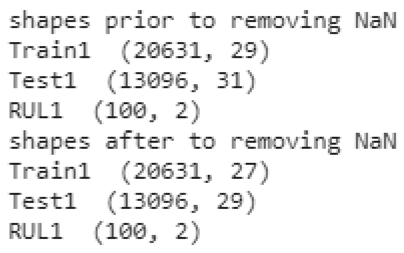


Figure 3: Engine cycles before fault for test data

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:



```
shapes prior to removing NaN
Train1  (20631, 29)
Test1  (13096, 31)
RUL1  (100, 2)
shapes after to removing NaN
Train1  (20631, 27)
Test1  (13096, 29)
RUL1  (100, 2)
```

Figure 4: Dataframe dimensions

Sensor Values vs Time

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

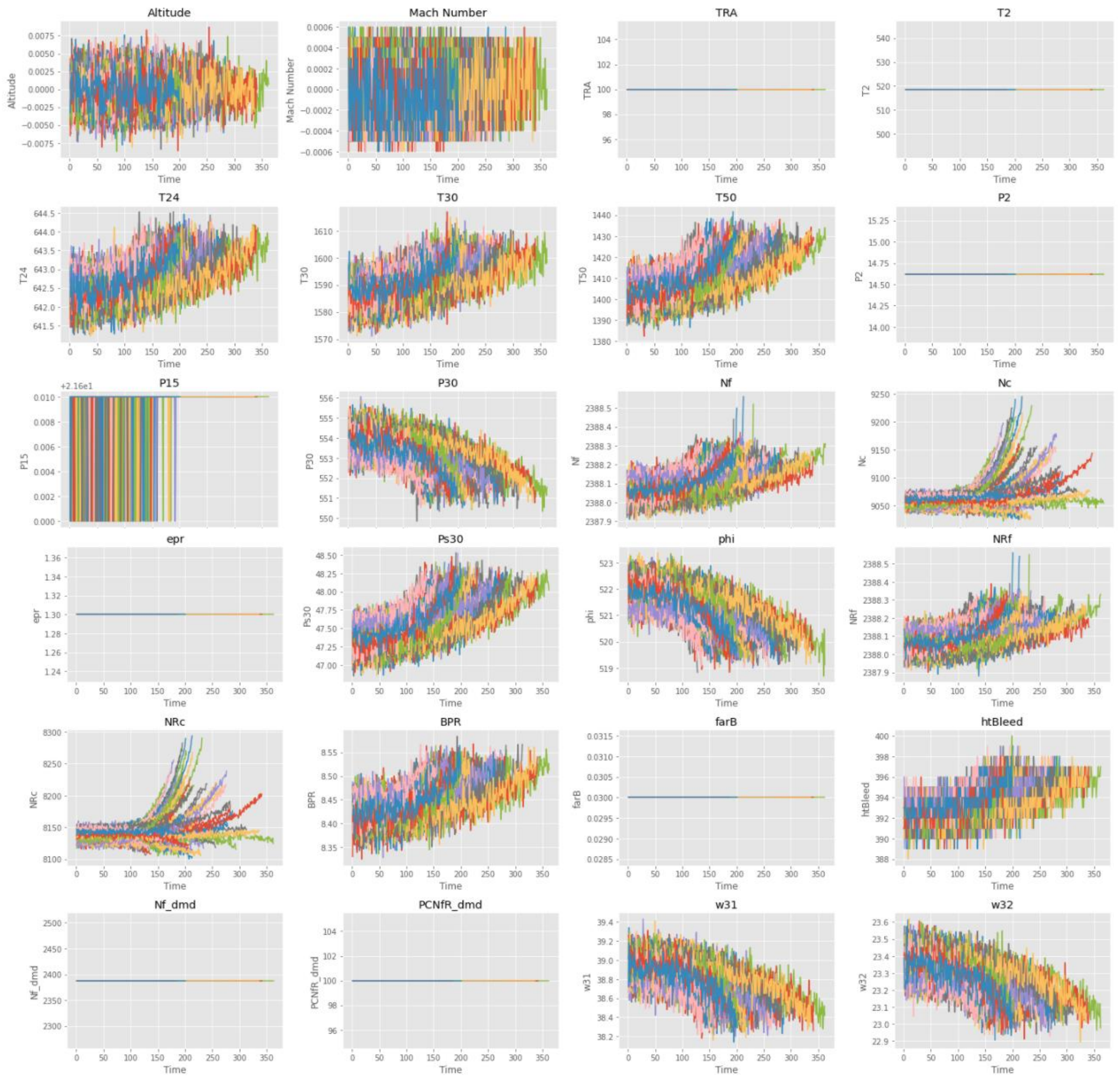
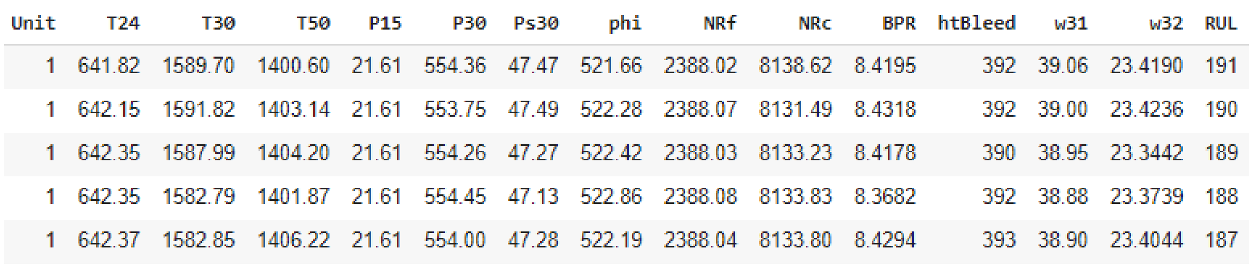


Figure 5: Sensor values vs time

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:



Unit	T24	T30	T50	P15	P30	Ps30	phi	NRf	NRc	BPR	htBleed	w31	w32	RUL
1	641.82	1589.70	1400.60	21.61	554.36	47.47	521.66	2388.02	8138.62	8.4195	392	39.06	23.4190	191
1	642.15	1591.82	1403.14	21.61	553.75	47.49	522.28	2388.07	8131.49	8.4318	392	39.00	23.4236	190
1	642.35	1587.99	1404.20	21.61	554.26	47.27	522.42	2388.03	8133.23	8.4178	390	38.95	23.3442	189
1	642.35	1582.79	1401.87	21.61	554.45	47.13	522.86	2388.08	8133.83	8.3682	392	38.88	23.3739	188
1	642.37	1582.85	1406.22	21.61	554.00	47.28	522.19	2388.04	8133.80	8.4294	393	38.90	23.4044	187

Table 4: Training dataframe

Bivariate Analysis

A correlation matrix was generated to identify related feature

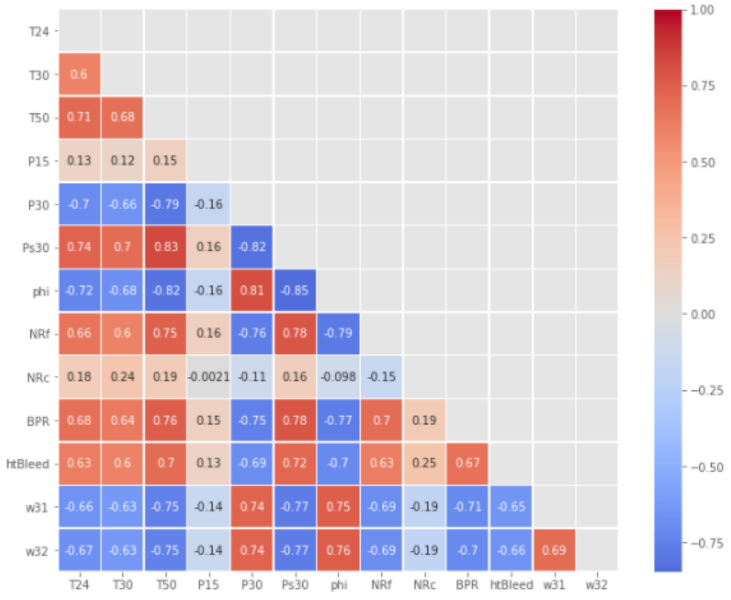


Figure 6: Correlation matrix to RUL

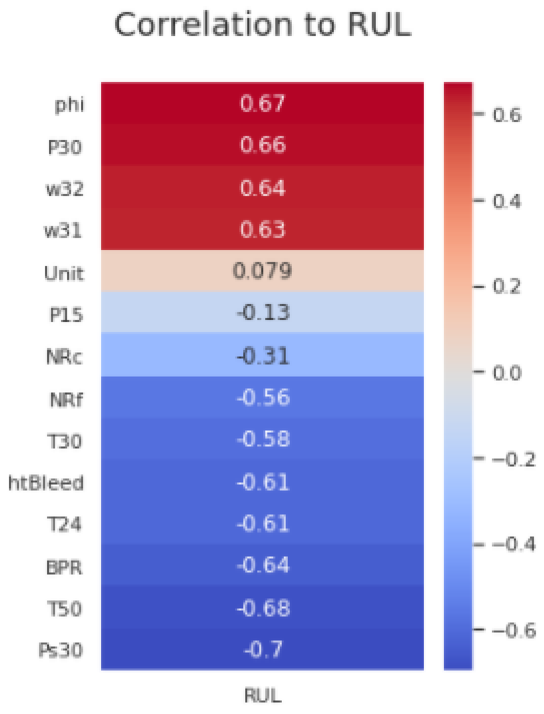


Figure 7: Correlation strength to RUL

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.

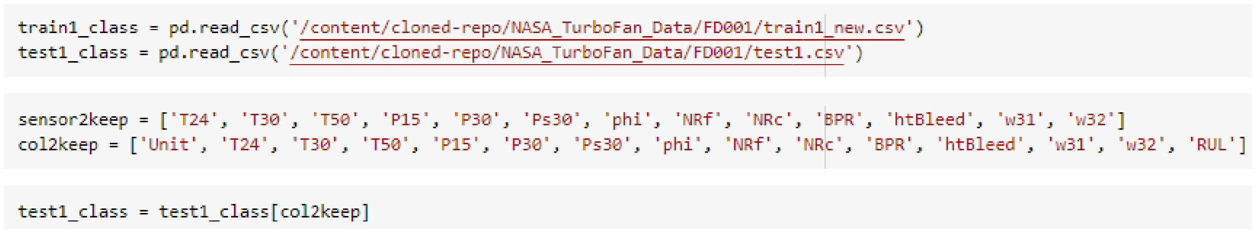
Decision Tree Classifier

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.

Data Visualization



```
train1_class = pd.read_csv('/content/cloned-repo/NASA_TurboFan_Data/FD001/train1_new.csv')
test1_class = pd.read_csv('/content/cloned-repo/NASA_TurboFan_Data/FD001/test1.csv')
```

```
sensor2keep = ['T24', 'T30', 'T50', 'P15', 'P30', 'Ps30', 'phi', 'NRf', 'NRc', 'BPR', 'htBleed', 'w31', 'w32']
col2keep = ['Unit', 'T24', 'T30', 'T50', 'P15', 'P30', 'Ps30', 'phi', 'NRf', 'NRc', 'BPR', 'htBleed', 'w31', 'w32', 'RUL']
```

```
test1_class = test1_class[col2keep]
```

Figure 8: Data Import

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

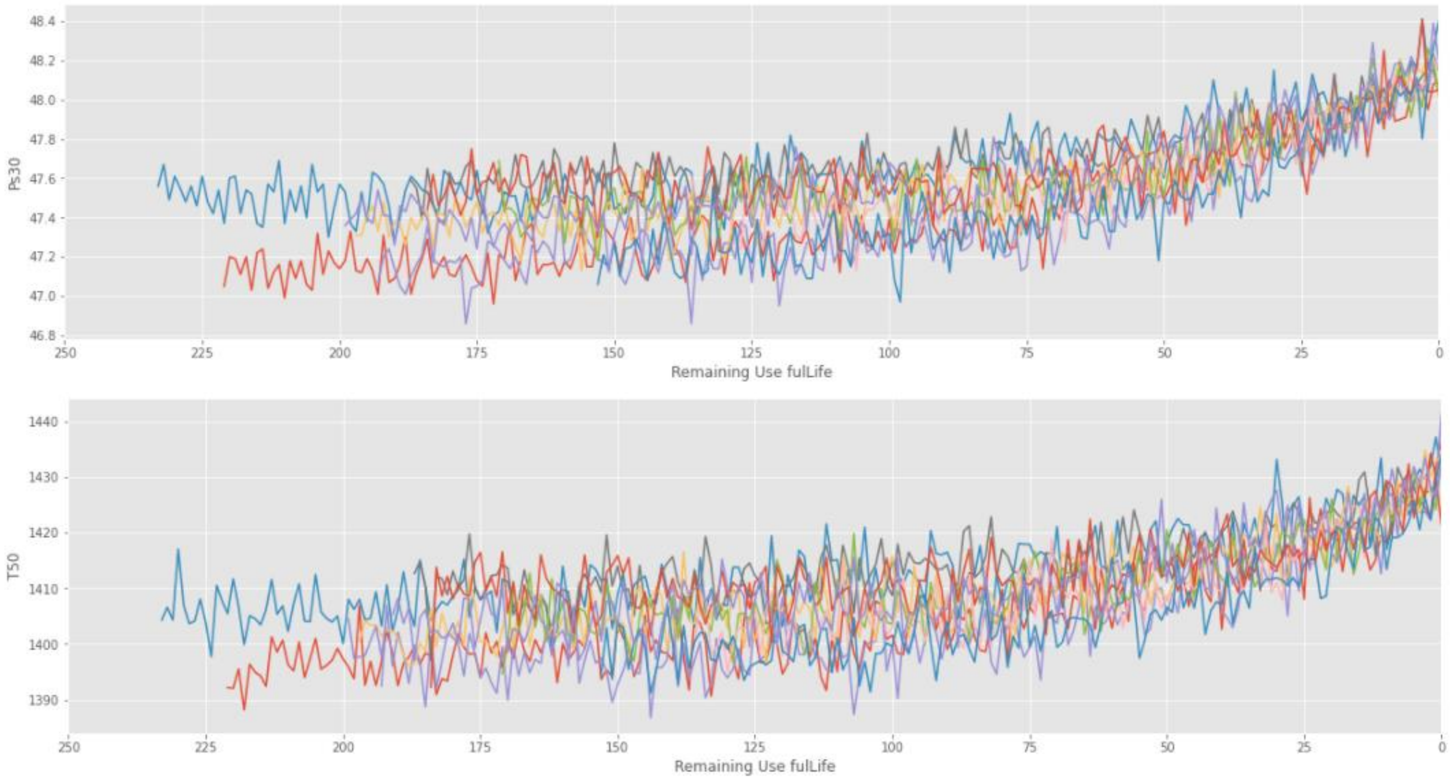


Figure 9: Sensor plot for every 10th engine unit

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

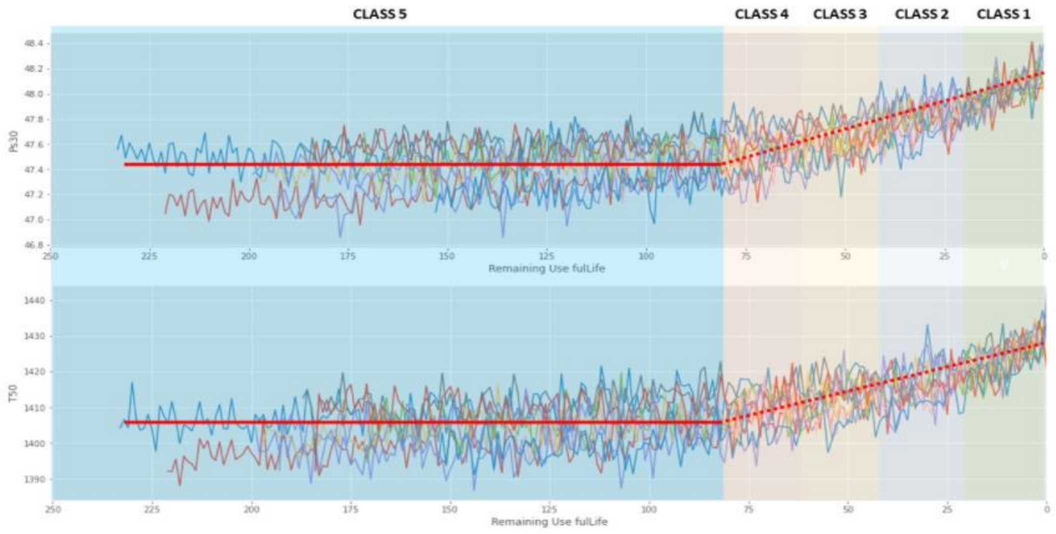


Figure 10: Sensor plot for every 10th engine unit with classes

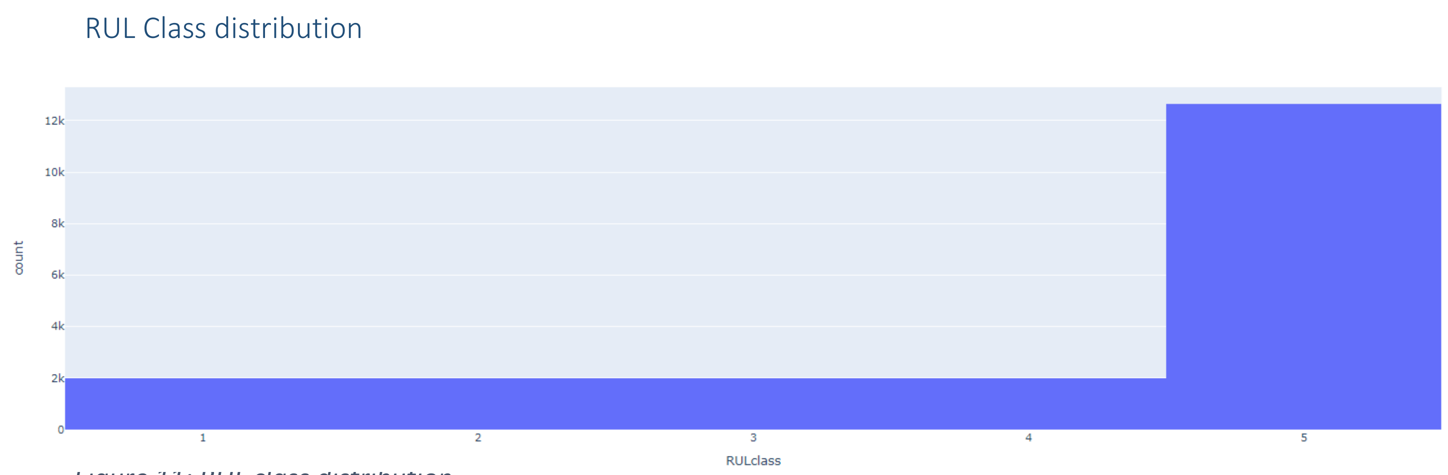
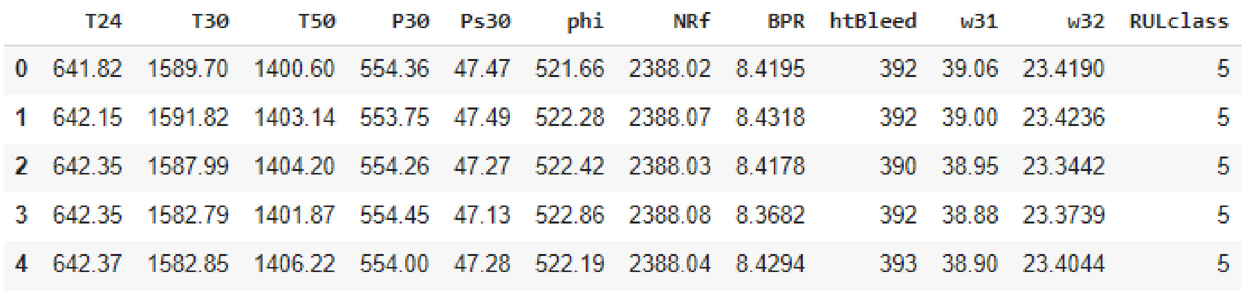


Figure 11: RUL class distribution

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.



	T24	T30	T50	P30	Ps30	phi	NRf	BPR	htBleed	w31	w32	RULclass
0	641.82	1589.70	1400.60	554.36	47.47	521.66	2388.02	8.4195	392	39.06	23.4190	5
1	642.15	1591.82	1403.14	553.75	47.49	522.28	2388.07	8.4318	392	39.00	23.4236	5
2	642.35	1587.99	1404.20	554.26	47.27	522.42	2388.03	8.4178	390	38.95	23.3442	5
3	642.35	1582.79	1401.87	554.45	47.13	522.86	2388.08	8.3682	392	38.88	23.3739	5
4	642.37	1582.85	1406.22	554.00	47.28	522.19	2388.04	8.4294	393	38.90	23.4044	5

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 r_i value. After this step, each minority example will be associated with a different neighborhood. The r_i value indicates the dominance of the majority class in each specific neighborhood. Higher r_i neighbourhoods contain more majority class examples and are more difficult to learn.
4. Normalize the r_i values so that the sum of all r_i values equals to 1.
5. Calculate the amount of synthetic examples to generate per neighbourhood. Because r_i 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 G_i data for each neighbourhood. First, take the minority example for the neighbourhood, x_i. Then, randomly select another minority example within that neighbourhood, xz_i.

```
5     12631                  1     12834
4      2000                  3     12720
3      2000                  5     12631
2      2000                  4     12547
1      2000                  2     12414
Name: RULclass, dtype: int64 Name: RULclass, dtype: int64
```

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.

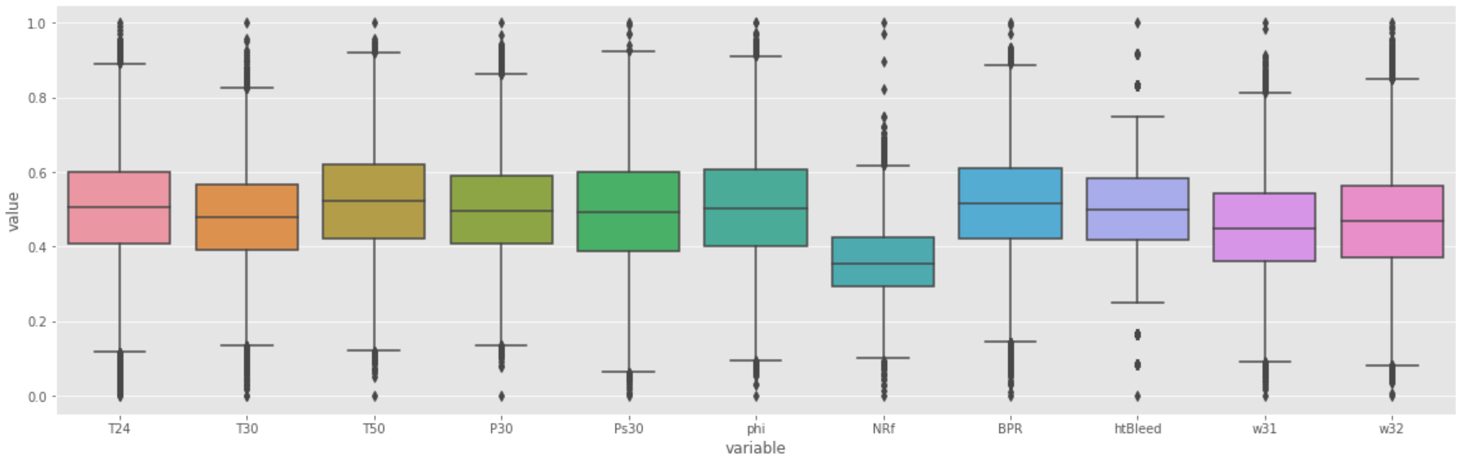
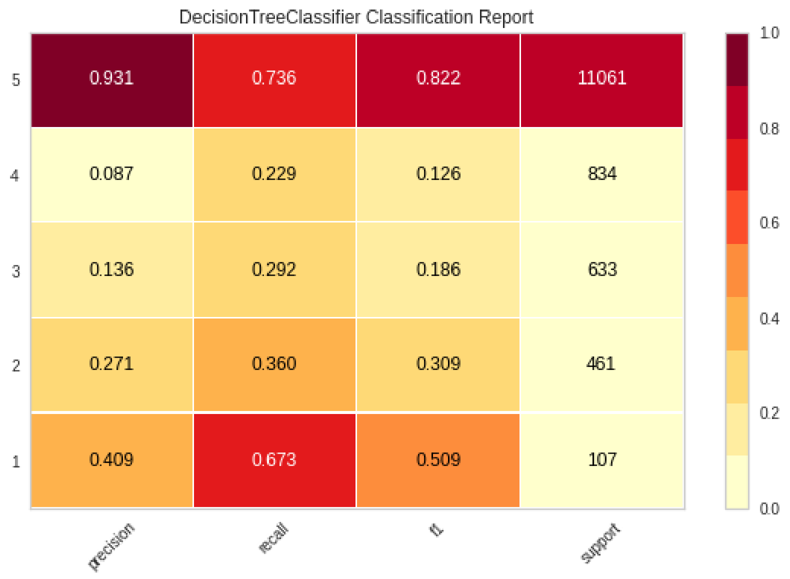


Figure 12: Training data after min-max scaling to 0-1

Train Baseline Classification Tree

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


DecisionTreeClassifier Classification Report

	precision	recall	f1	support
5	0.931	0.736	0.822	11061
4	0.087	0.229	0.126	834
3	0.136	0.292	0.186	633
2	0.271	0.360	0.309	461
1	0.409	0.673	0.509	107

1.0
0.8
0.6
0.4
0.2
0.0

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.

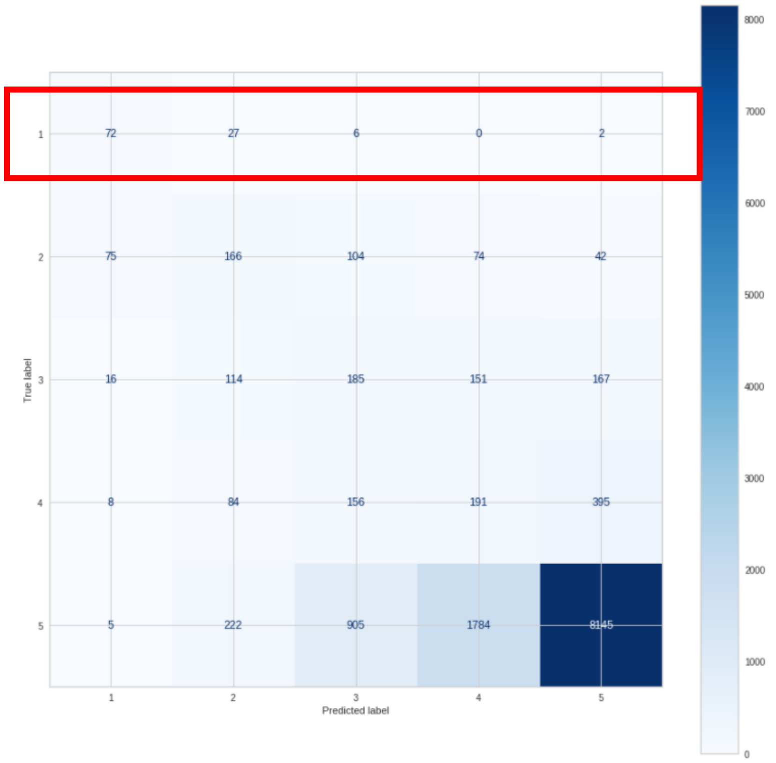


Figure 14: Initial model confusion matrix

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.

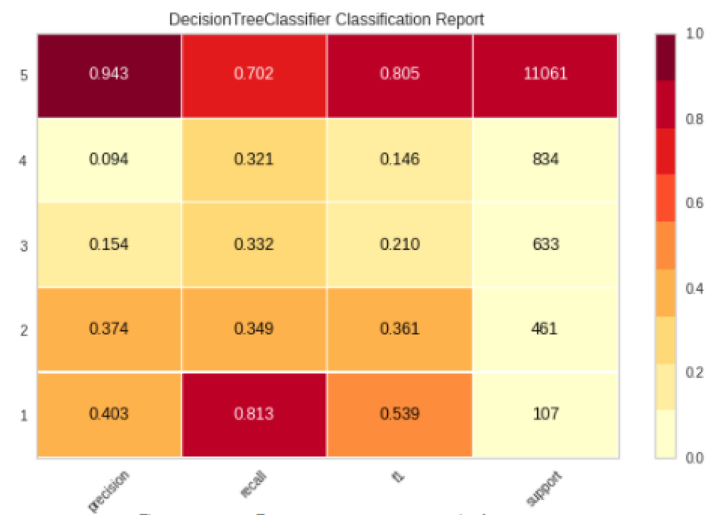


Figure 15: Model evaluation after GridSearch

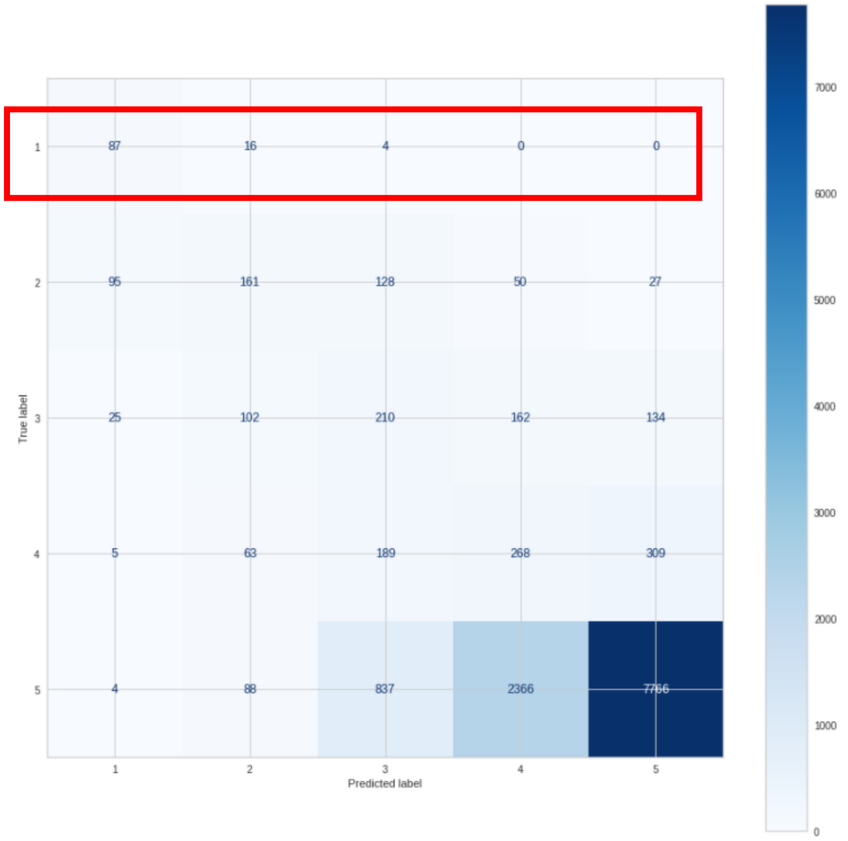
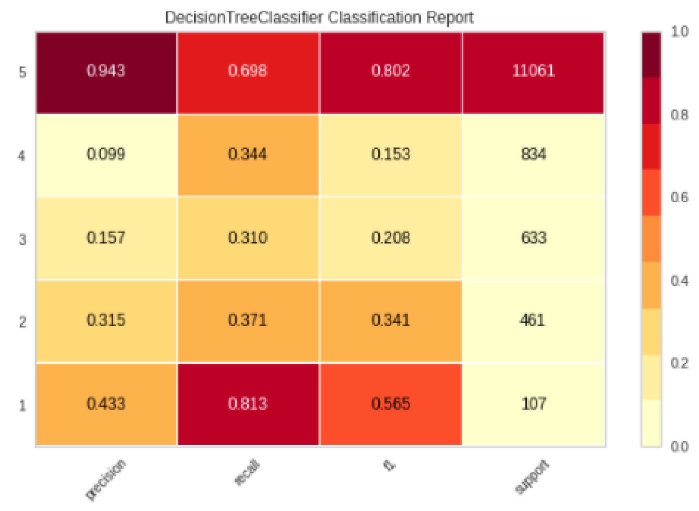


Figure 16: Confusion matrix after GridSearch

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

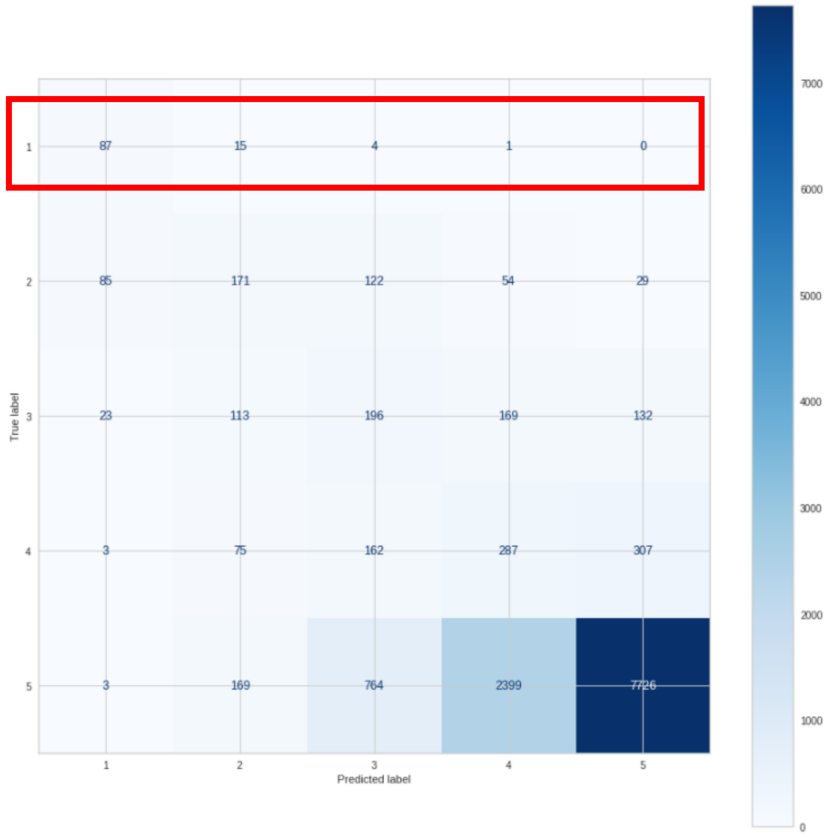
To tackle the imbalanced dataset we are first going to try Scikit-Learn's *class_weight:"balanced"* parameter. The "balanced" mode uses the values of y to automatically adjust weights so they are inversely proportional to the class frequencies in the input data. We will use the same parameters from our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



DecisionTreeClassifier Classification Report

	precision	recall	f1	support
5	0.943	0.698	0.802	11061
4	0.099	0.344	0.153	834
3	0.157	0.310	0.208	633
2	0.315	0.371	0.341	461
1	0.433	0.813	0.565	107

Figure 17: Model evaluation after tuning class weights



True label \ Predicted label	1	2	3	4	5
1	87	15	4	1	0
2	85	171	122	54	29
3	23	113	196	169	132
4	3	75	162	287	307
5	3	169	764	2399	7726

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.

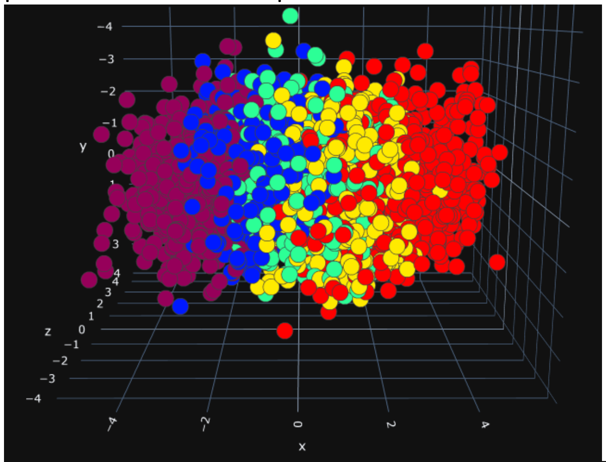


Figure 19: 3D LDA

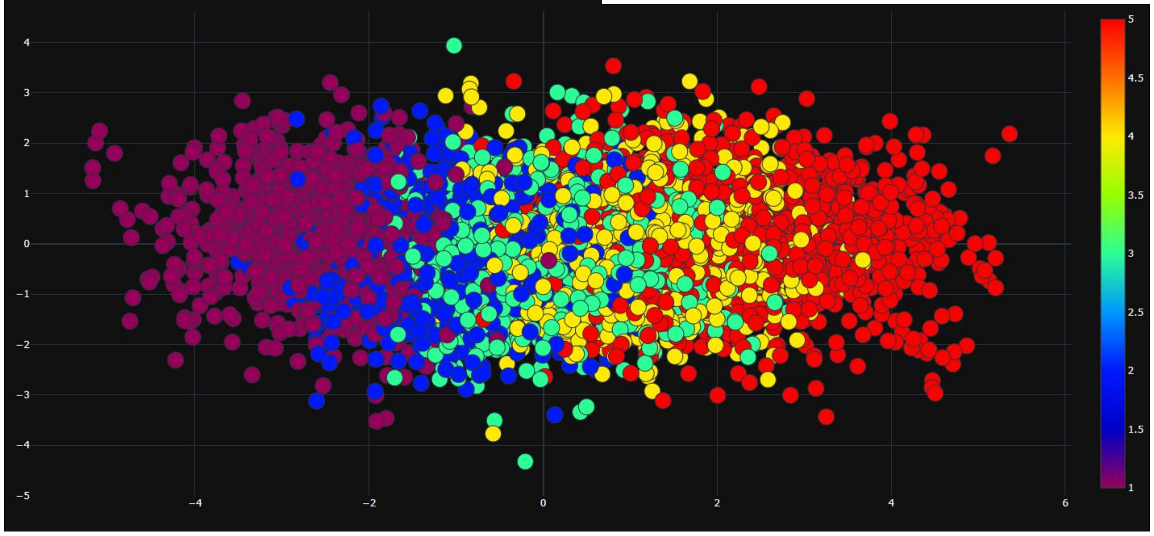


Figure 20: 2D LDA

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

- Criterion: Gini
- Max_depth: 9
- Min_samples_split: 1
- Min_samples_leaf: 2

The final tuning step we took was to use Scikit-Learn's *class_weight: "balanced"* parameter to account for the imbalanced data.

Model	Precision	Recall	F1
Initial	0.409	0.673	0.509
Final	0.433	0.813	0.565
Change	0.024	0.140	0.056

Table 7: Decision tree evaluation summary

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

Random Forest Classifier

A random forest classifier is a collection of individual decision trees (parameter: n_estimators) that operate together as an ensemble. Random forests are quite intuitive and the fundamental concept behind them is voting by committee – a large number of relatively uncorrelated models (trees) operating as a committee will outperform any of the individual constituent models. This produces a model that is more accurate than the sum of its parts. This is due to the fact that the trees protect each other from their individual errors.
We will use the cleaned and prepared data from the previous decision tree classifier model.

Train Baseline Tree

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

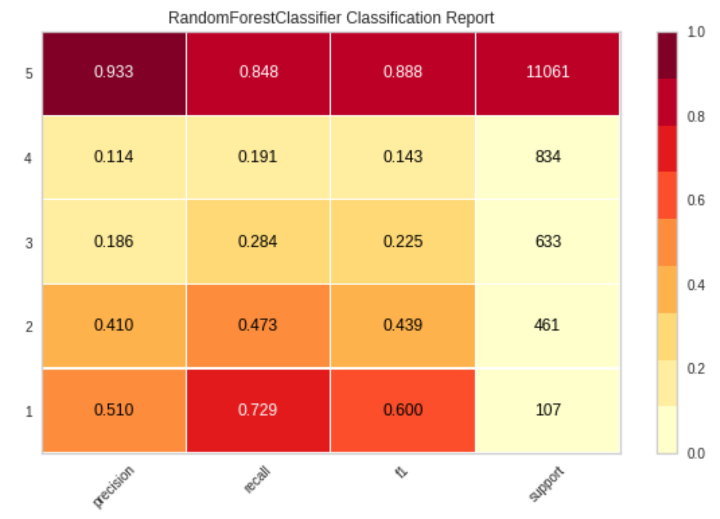


Figure 21: Baseline Random Forest evaluation

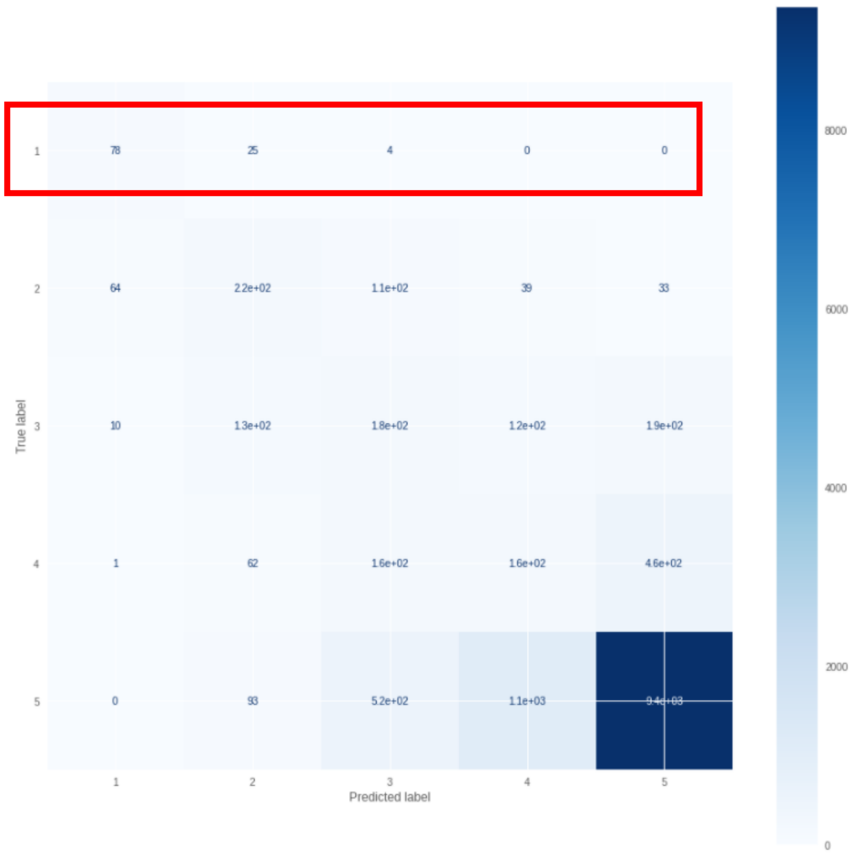


Figure 22: Baseline Random Forest confusion matrix

Hyperparameter Tuning

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

- n_estimators: 800
- class_weight: balanced
- min_samples_split: 2
- max_depth: 40
- max_features: sqrt

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

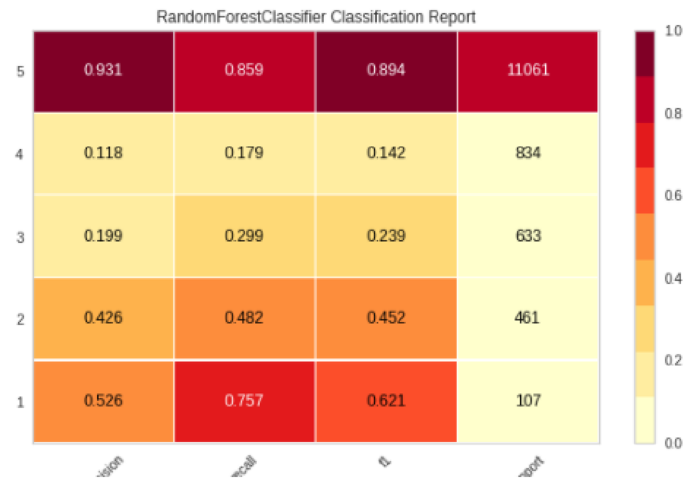


Figure 23: Tuned Random Forest evaluation

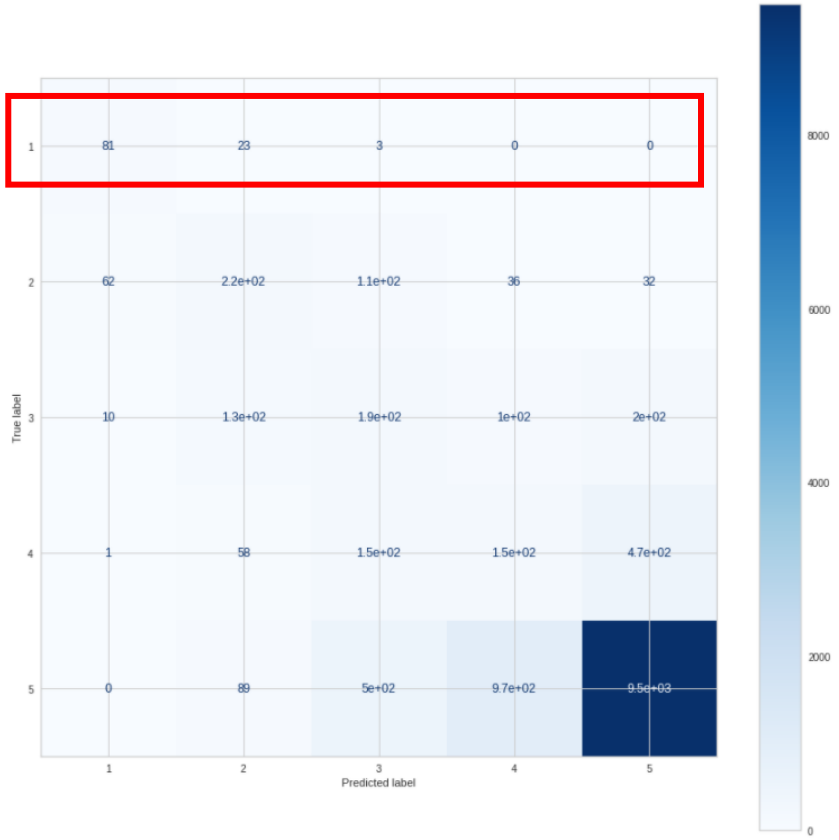
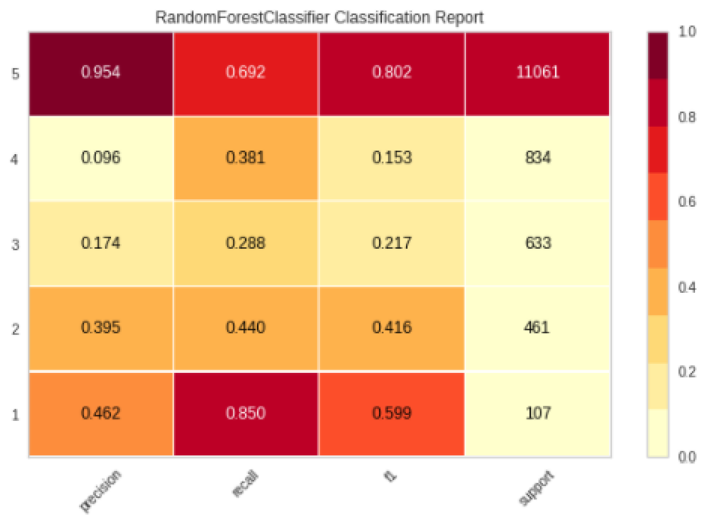


Figure 24: Tuned Random Forest confusion matrix

The model was severely overfitting the data so the parameters were tuned manually from experience to the following:

- n_estimators: 50
- class_weight: balanced
- min_samples_split: 2
- max_depth: 5
- max_features: sqrt


RandomForestClassifier Classification Report

	precision	recall	f1	support
5	0.954	0.692	0.802	11061
4	0.096	0.381	0.153	834
3	0.174	0.288	0.217	633
2	0.395	0.440	0.416	461
1	0.462	0.850	0.599	107

Figure 25: Manually Tuned Random Forest evaluation

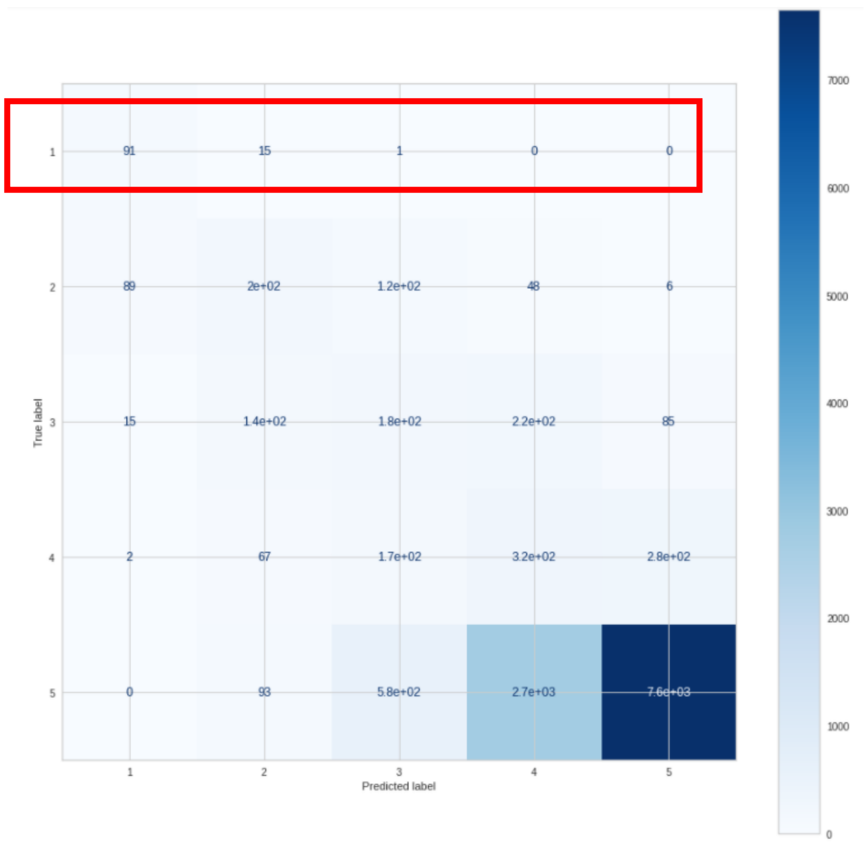

True label \ Predicted label	1	2	3	4	5
1	91	15	1	0	0
2	89	2e+02	1.2e+02	48	6
3	15	1.4e+02	1.8e+02	2.2e+02	85
4	2	67	1.7e+02	3.2e+02	2.8e+02
5	0	93	5.8e+02	2.7e+03	7.6e+03

Figure 26: Manually Tuned Random Forest confusion matrix

Random Forest 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. Following the initial model, we tuned the following parameters via GridSearch:

- n_estimators: 800
- class_weight: balanced
- min_samples_split: 2
- max_depth: 40
- max_features: sqrt

With these parameters, the model was severely overfitting the data. The n_estimators and max_depth were both reduced and the following model parameters were used:

- n_estimators: 50
- class_weight: balanced
- min_samples_split: 2
- max_depth: 5
- max_features: sqrt

Model	Precision	Recall	F1
Initial	0.510	0.729	0.600
GridSearch	0.526	0.757	0.621
Final	0.462	0.850	0.599
Change	-0.048	0.121	0.001

Table 8: Random forest evaluation summary

When comparing our Random Forest model to the previous Decision Tree model, we observed:

- 2.9% increase in precision
- 3.7% increase in recall
- 3.4% increase in F1

When looking at the confusion matrices of the two models, we observed:

- 16 late predictions for the Random Forest
- 20 late predictions for the Decision Tree

The model is overfitting the data, possibly due to the ADASYN implementation combined with the "weighted" average to account for class imbalance. Since the training data is already balanced with ADASYN the "weighted" average evaluation metric can not differentiate between classes.
The model parameters were adjusted from experience to reduce overfitting and improve generalization. The final manually adjusted model was significantly better than the initial model and uses far less memory and computational resources than the GridSearch model.

XGBoost Classifier

XGBoost is short for Extreme Gradient Boosting and is an efficient implementation of the stochastic gradient boosting machine learning algorithm. The stochastic gradient boosting algorithm, also called gradient boosting machines or tree boosting, is a powerful machine learning technique that performs well or even best on a wide range of challenging machine learning problems.
It is an ensemble of decision trees algorithm where new trees fix errors of those trees that are already part of the model. Trees are added until no further improvements can be made to the model. XGBoost is an effective machine learning model, even on datasets where the class distribution is skewed.
Before any modification or tuning is made to the XGBoost algorithm for imbalanced classification, it is essential to test the default XGBoost model and establish a performance baseline.

Train Baseline Model

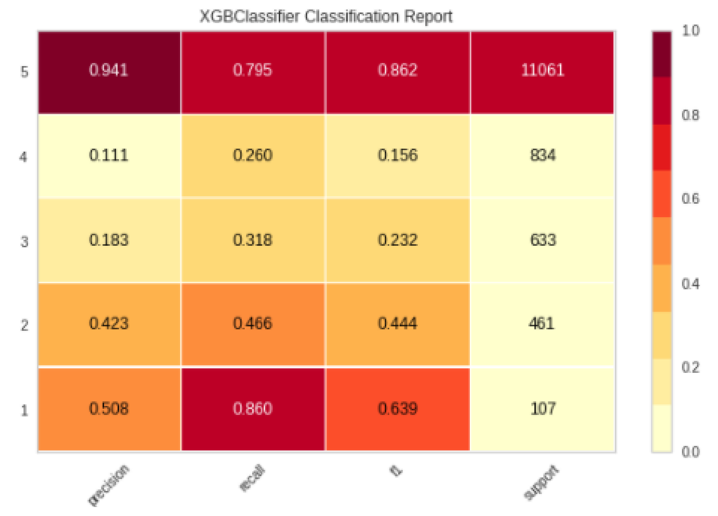


Figure 27: Initial XGBoost evaluation

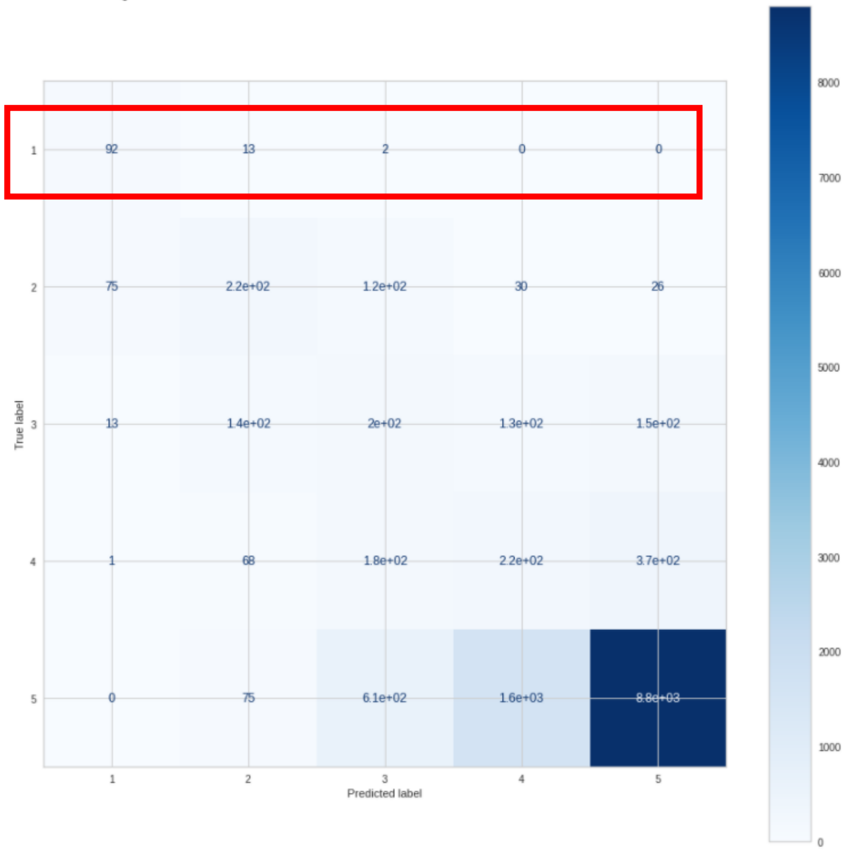


Figure 28: Initial XGBoost confusion matrix

From the results above, we can see that our initial XGBoost model is already performing better than either of the previous tuned models.

Hyperparaneter Tuning

We did not perform an exhaustive search, but the few parameters that we did search through did not significantly improve our model. The best performing model was xgb2, where the tree_method was set to "hist" and the scale_pos_weight was set to 10.
Tree_method was set to "hist" to speed up computations and scale_pos_weight was set to 10 to account for class imbalance. There are roughly 10 times as many "negative" class samples, (Classes 2-5) compared to "positive" Class 1 samples.
We retrained our model with the above parameter values with the following results.

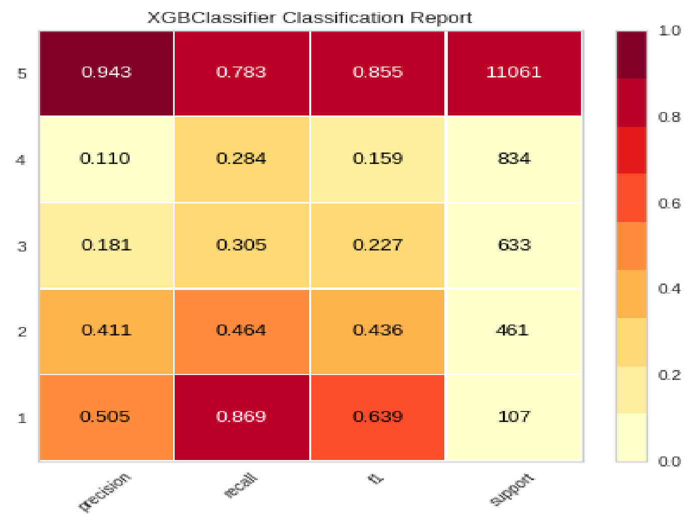


Figure 29: Final XGBoost evaluation

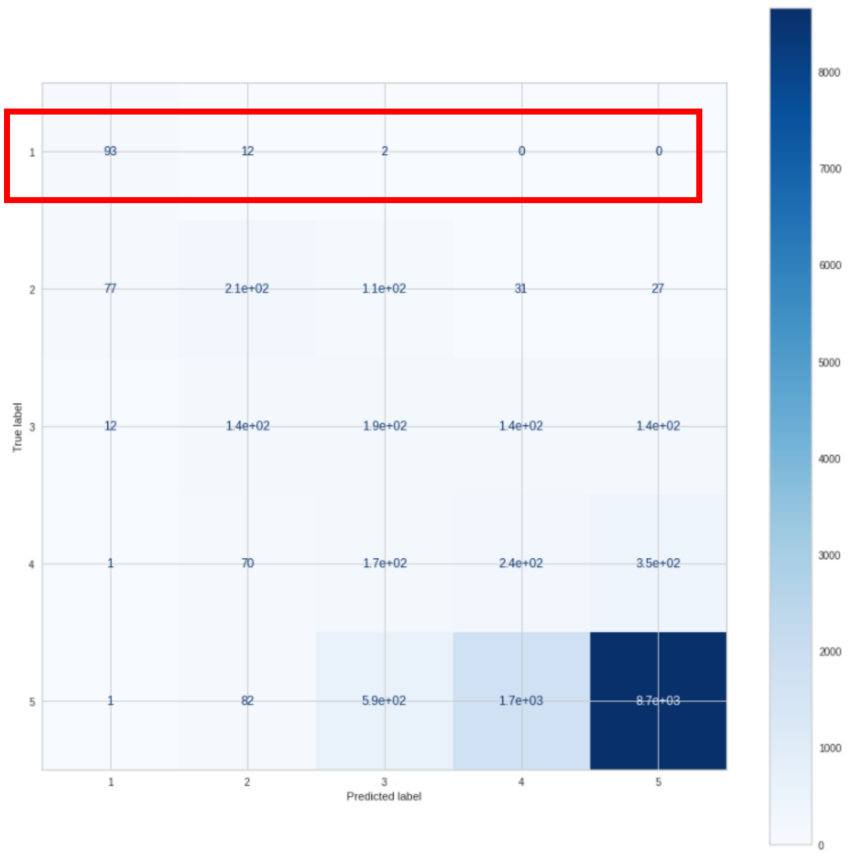


Figure 30: Final XGBoost confusion matrix

XGBoost Summary

Our initial model was very close in performance to our final model. A couple of the GridSeach parameters actually decreased model performance. The only differences between the initial model, xgb1, and our final model, xgb2, was the tree_method set to “hist” and the scale_pos_scale set to 10.

Model	Precision	Recall	F1
Initial	0.508	0.860	0.639
Final	0.505	0.869	0.639
Change	-0.003	0.009	0.000

Table 9: Random forest evaluation summary

When comparing our XGBoost model to the previous Random Forest model, we observed:

- 4.3% increase in precision
- 1.9% increase in recall
- 4.0% increase in F1

When looking at the confusion matrices of the two models, we observed:

- 14 late predictions for the XGBoost
- 16 late predictions for the Random Forest

The model is still overfitting the data, but not as bad as previous models.

Next Steps

Our best model, XGBoost, obtained a recall of 87% with 14 late predictions. The next steps to try and improve our model include:
feature selection
feature engineering - remove noise, sensor smoothing
deep learning with Tensorflow and Keras
apply asymmetric loss function to penalize late predictions more heavily
After applying these steps, we will have to revisit our business objective to see if our results align. A model with 90% recall may be adequate to put into production but depends on the costs associated with pulling engines out of service early to avoid late predictions.