MARMARA UNIVERSITY
FACULTY OF ENGINEERING

DIAGNOSTICS WITH ARTIFICIAL INTELLIGENCE ON A CNC MACHINE

Cem Karahan

GRADUATION PROJECT REPORT
Department of Mechanical Engineering

Supervisor

Assoc. Prof. İBRAHİM SİNA KUSEYRİ

ISTANBUL, 2025

MARMARA UNIVERSITY
FACULTY OF ENGINEERING

Diagnostics with Artificial Intelligence on a CNC Machine
by

Cem Karahan
June 30, 2025, Istanbul

SUBMITTED TO THE DEPARTMENT OF MECHANICAL ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

OF

BACHELOR OF SCIENCE

AT

MARMARA UNIVERSITY

Signature of Author(s) .....

Department of Mechanical Engineering

Certified By .....

Project Supervisor, Department of Mechanical Engineering

Accepted By .....

Head of the Department of Mechanical Engineering

ACKNOWLEDGEMENT

First of all, I would like to thank my supervisor Assoc. Prof. İbrahim Sina Kuseyri for the valuable guidance and advice on preparing this thesis and giving me moral and material support.

January 2025 Cem Karahan

Contents

Abstract

The increasing demand for efficiency in modern manufacturing has driven the need for innovative maintenance strategies. This thesis investigates predictive maintenance leveraging decision tree-based algorithms on CNC machining data to prevent failures, reduce downtime, improve maintenance scheduling and extend tool life in the production process. Using a dataset containing five distinct failure types and operational parameter- such as power, torque, tool wear, rotational speed and material type-the study applies decision tree-based methods, including decision trees, random forests and XGBoost. Feature engineering is employed to enhance model performance and extract actionable insights. By leveraging existing machine learning models, this research aims to demonstrate the practical viability of predictive maintenance solutions not only for factories have CNC machinery but also any industrial environment capable of real time equipment monitoring in corelation with Industry 4.0.

SYMBOLS

G_t: Gini Impurity Index for Node t

$P_{i,k}$: Probability of Class i occurring in the Dataset

H_t: Entropy (impurity measure) at Node t

D_i: Dataset at the i^{th} Child Node

D: Dataset at the Parent Node

ξ_i: Split Chosen at the i^{th} Node

λ: Regularization Coefficent, also L2 Regularization in XGBoost

mr(X,Y): Margin Function

h(X, θ): Predicted Class by a single tree

S: Strength of a Random Forest (Expected Margin)

PE: Generalization Error Bound

G_L, G_R: Gradient Sums

H_L, H_R: Hessian Sums for Left and Right Child Nodes

α: L1 Regularization in XGBoost

γ: Minimum Loss Reduction to Split a Node in XGBoost

ABBREVIATIONS

AI: Artificial Intelligence

AUC: Area Under the Curve

DT: Decision Tree

FP: False Positive

FN: False Negative

kNN: K-Nearest Neighbours

ML: Machine Learning

PCA: Principal Component Analysis

PdM: Predictive Maintenance

PM: Preventive Maintenance

RF: Random Forest

RM: Reactive Maintenance

ROC: Receiver Operation Characteristic

RUL: Remaining Useful Life

SVM: Support Vector Machines

TP: True Positive

TN: True Negative

XGBoost: Extreme Gradient Boosting

LIST OF FIGURES

LIST OF TABLES

1. Introduction

In manufacturing, maintenance is and has been a very important aspect of the workflow of factories. Regardless of the sector, heavy or light machining, all equipment is subject to maintenance either because of regular check-up or repair. In large factories, regular maintenance can take up to weeks and during these intervals all the production may seize in some circumstances. These pauses cause companies to lose large amounts of money and increase their operation and maintenance costs thus affect profit margins. While companies start to embed smart manufacturing tools with the emerging technological advancements and reducing costs of these technologies, it is still not industry-standard. Production issues led to significant losses in sales, estimated at up to €400 million per week [1]. An example from another sector, not a production one, Amazon experienced only 49 minutes of downtime, and this cost the company $4 million in lost sales in 2013 [2]. Predictive maintenance is not an old subject but the implementation of it has its own hardships involving aligning the existing production line to industry 4.0. The advancement of industry 4.0, involving industrial internet of things (IIoT), artificial intelligence (AI) or machine learning (ML), big data, data warehouse, automation and cyber-physical systems, is making easier for companies to embrace predictive maintenance. There are three maintenance types in total: Reactive, preventative and predictive maintenance. Reactive maintenance, as the name states, is only done if a need occurs. This need may arise from a fault or equipment failure and there is no way to know when will a failure or a fault will happen which makes this approach a risky one. Preventative maintenance on the other hand is carried out on a planned schedule either in a fixed interval of time or cycle. This have its own disadvantages such as an unnecessary maintenance being carried out. Predictive maintenance, however, offers a more balanced approach by timely maintenance with initial high installation price.

CNC machines are highly effective and expensive tools that are essential for manufacturing. The performance and reliability of CNC determines the quality of products it has processed. However, CNC may incur faults unpredictably, which will cause low precision, production stagnation or even serious loss and casualties if troubleshooting is not timely. Therefore, predictive maintenance is necessary to avoid faults and improve the reliability of CNC [3]. Predictive maintenance has two fundamental components to work. 1-) System architecture: Technology to enable live, meaningful data to be stored and integrated with predictive

maintenance algorithm. 2-) Predictive maintenance algorithm: There are various methods to construct the algorithms such as artificial neural network, deep learning methods, decision trees, and other machine learning methods. Approach to the problems and the selection of methods may differ according to the specific need of the field and must be assessed case by case. This paper will focus on the algorithm side of predictive maintenance and the method is chosen to be decision trees. Using a highly imbalanced synthetic data, the goal of this study is to establish a solid accuracy on classifying failures while having minimum false-positive alarms. The rest of this thesis is organized as follows: the Theoretical Background section provides brief information about maintenance types, relevant machine learning algorithms, and the working logic of decision trees, random forests and XGBoost. The Methodology section introduces the dataset and the modelling process, starting with a default decision tree then advancing to more complex methods including random forests and XGBoost, with application of techniques such as SMOTE, Bayesian optimization and PCA anomaly detection, and providing insight and justification in various steps. The Results and Discussion section presents the performance of each method, providing details by comparing the results and explaining the behaviour of the algorithms. Finally, the Conclusion section summarizes the key findings of the thesis.

2. Theoretical Background

2.1. Predictive Maintenance

Maintenance is the backbone of any industry or a manufacturing plant. It is associated with operating costs, efficiency and runtime of a business. There are three types of maintenance: Reactive, preventative, and predictive. Reactive maintenance (RM) is done after a failure has happened. It follows a 'run-to-failure' approach which means no preventive action is taken beforehand which dates back to the early stages of industrialization. This method may be cost-effective in the short term but has disadvantages because of the possibility of long-term expenses due to emergency repairs or secondary damage. In contrast, preventive maintenance (PM) aims to maintain, replace or repair equipment in a scheduled, regular intervals. PM aims to lessen the likelihood of failures and achieve optimum system reliability and safety while using the least amount of maintenance resources possible. PM requires the machines to have a known lifespan. It is assumed that the failure behaviour of the equipment can be expressed with a bathtub curve. While PM approach helps extend equipment lifespan, minimize unplanned

downtime and improve operational efficiency, it can also lead to unnecessary maintenance actions which in turn decrease operational efficiency.[3], [5]

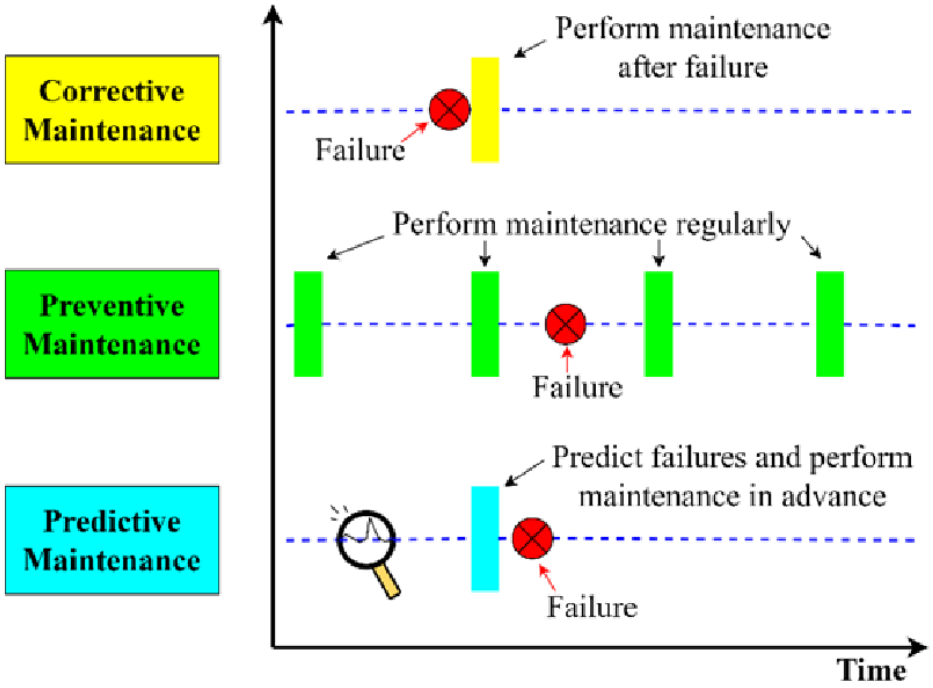

Figure 1: *Maintenance Schedules by Type [6]*

The evolution and development of maintenance, along with industrial revolution steps, carried manufacturing and efficiency and amount of production forward. Predictive maintenance (PdM) is a balancing approach between reactive and preventive maintenance, as portraited in Figure 2. PdM is a data-driven strategy that aims to "predict" a failure beforehand. It uses real-time data, sensor data and detect anomalies or patterns lead to potential failures by leveraging analytical models, often machine learning. This approach is part of Industry 4.0 and plays an important role of intelligent decision making based on data. PdM comes with high initial adaptation costs since it requires a suitable system architecture, and this can be expressed as a trade-off between high initial costs with less maintenance in the long term. This trade-off can be expressed with a graph as shown in the figure 3.

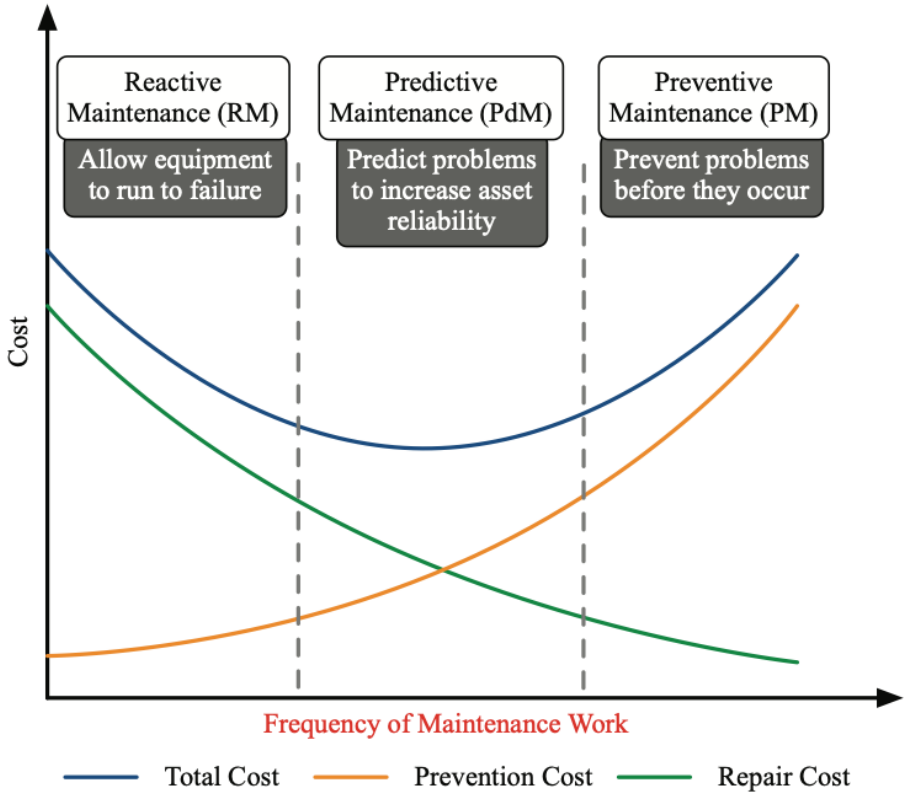

Figure 1. *Comparison of RM, PM and PdM on the cost and frequency of maintenance [2]*

2.2. Machine Learning Methods

2.2.1. Introduction to Machine Learning

There is no doubt that one of the pillars of PdM is machine learning. Machine learning (ML) is a subfield of artificial intelligence (AI), and it is, in simple terms, constructing an algorithm to learn patterns from the data and make predictions based on them. The information is the past data, and the prediction will be made using future data. The idea of constructing an algorithm from data emerged when traditional, statistical modelling approaches - which typically has a form y = f (predictor variables, random noise, parameters) - started to be unable to capture complex, high dimensional and nonlinear patterns in data. The algorithmic approach is to find a function f(x) which initially considered as a black box that learns from strong predictors, converging if recurring patterns exist in the data and ultimately achieving a good predictive accuracy [7]. In the scope of machine learning, there are three types of learning: supervised, unsupervised and reinforcement learning. In supervised learning, the model learns from labelled data. Unsupervised learning is done by capturing the structure of unlabelled data, in other words it is done by clustering similar instances. Reinforcement learning is a type where the algorithm is trained with rewards and penalties, eventually optimizing its outcome. Deep learning is a subset of machine learning where the model uses layers of artificial neural

networks to understand patterns. Some common ML methods can be illustrated below as an example.

- Linear Models
    - Linear Regression, Logistic regression
- Instance-Based Methods
    - K-Nearaest Neighbors
- Support Vector Machines (SVM)
- Probabilistic Models
    - Naive Bayes
- Tree Based Methods
    - Decision Tree, Random Forest, Gradient Boosting
- Neural Networks

2.2.2. Reasoning of Selecting Tree-Based Algorithms

There are many methods of machine learning techniques to choose to be applied in PdM. All these models have their own advantages and disadvantages. Among these algorithms decision tree and decision tree-based algorithms are selected because they can capture non-linearities, suitable for wide variety of data and especially for decision tree, the relative easiness to interpret and understand. Linear models like regression, SVM may not capture the noise, and the imbalanced nature of the data may be challenging for these methods. K-NN is a relatively easy method but for large data it may not scale well enough compared to tree-based methods. Probabilistic methods like Naïve Bayes need independent feature space which is not the case for multiple sensor machine data. SVM is a powerful method, yet similar to K-NN its scalability to data size may not be appropriate for a manufacturing environment. Lastly, deep learning methods are more suitable when there is an abundance of data for leveraging the power of these methods. The dataset chosen for this thesis is not big enough for deep learning methods to surpass decision tree-based algorithms. It is also necessary to mention that although tree ensemble methods are not as transparent as a single decision tree, they are more interpretable than deep learning methods which are often labelled as ‘black box’ methods. This may not be an issue if explainability or interpretability is not an issue and result of the algorithm is accepted without question. Accuracy versus interpretability trade-off is a popular term in machine learning, symbolizing this dilemma. A very effective, complex, accurate model may please with its predictions, but the learning steps will be abstract. On the other hand, less accurate but more interpretable algorithm will give insights of the problem itself and may be a foundation

work for future, more complex solutions. The aim of this paper is to illuminate and when possible, interpret the steps the algorithm takes. For instance, visualising the depth of the tree and the steps of pruning the tree, or demonstrating how data is partitioned after splitting and how the data or parameters alter the results.

2.2.3. Brief Information on Python and Libraries

Python is a very popular and widely used programming language. Moreover, most recent and deep learning libraries are Python based and have been for many years [18]. Like many other machine learning tools, decision tree is also included as a library that could be used within Python. Scikit-learn is an open-source library that has many features for machine learning, including decision tree-based algortihms, performance metrics and feature engineering tools. Other than Scikit-learn, tools like Bayesian optimization, reading the dataset, performing various alterations to data and creating visualisations are made easier with the help of libraries in Python. Some major libraries that are used in this thesis other than Scikit-learn are ‘pandas’, ‘numpy’, ‘matplotlib.pyplot’, ‘imblearn’ , ‘skopt’, ‘xgboost’ and ‘pyplot’-‘seaborn’ for visualisation.

2.2.4. The Fundamentals and Maths of Decision Trees

Decision trees are versatile algorithms, meaning they can be used in both regression and classification. They are non-parametric which is another way of saying they do not make explicit assumptions of the functional form of f(x), that they are algorithmic models. Not just decision trees but all non-parametric methods are easier to adapt into various functions as long as they are fed with large enough number of observations. However, algorithmic methods are more prone to overfit, which is a term when the model tries to fit the training data too much, instead of generalizing. The name tree is very intuitive, as decision tree have a flowchart scheme. The tree begins with a node, a junction where a binary question is asked, and this is called a root node. Then this node splits into two, creating its child nodes containing two, non-overlapping subsets of the data. A node is called parent node if it has successor nodes, which are the child nodes of that parent node. Then if the necessary conditions are met and splitting continues, the child nodes become parent nodes. Once a child node does not split further, then this node is named as a leaf node. Leaf node means that no node will follow this node, that it has no child nodes. There are many criteria to decide whether the node should split or not. All these criteria share one goal, and that is finding the best split possible. The aim is to have the best predictor, the best estimate of the test or a new data. Now comes the first question, how does a split occur, what is the criteria for a node to split further to become a parent or a node

to become a leaf? These are perhaps the fundamental questions as for the working principle of decision trees.

To showcase a very simple decision tree, a small classic dataset of plant is used. This dataset is from 1936, Fisher. It is one of the earliest data presented for evaluating classification techniques [9]. In the Figure 4 below, a basic decision tree is constructed to show the node types, splitting logic and the visual representation of a decision tree. The starting node is called the root node. Here the question is "Is petal length (cm) smaller than or equal to 2.45?". The samples that have petal lengths higher than 2.45 cm are shown in the red node, with all samples belonging to the class "setosa", making it a leaf node. The others subset of the original data is then further split with the same dimension, "petal width (cm)" but with a different value proposition. After this, the child nodes of this node become leaf nodes and ends the decision tree.

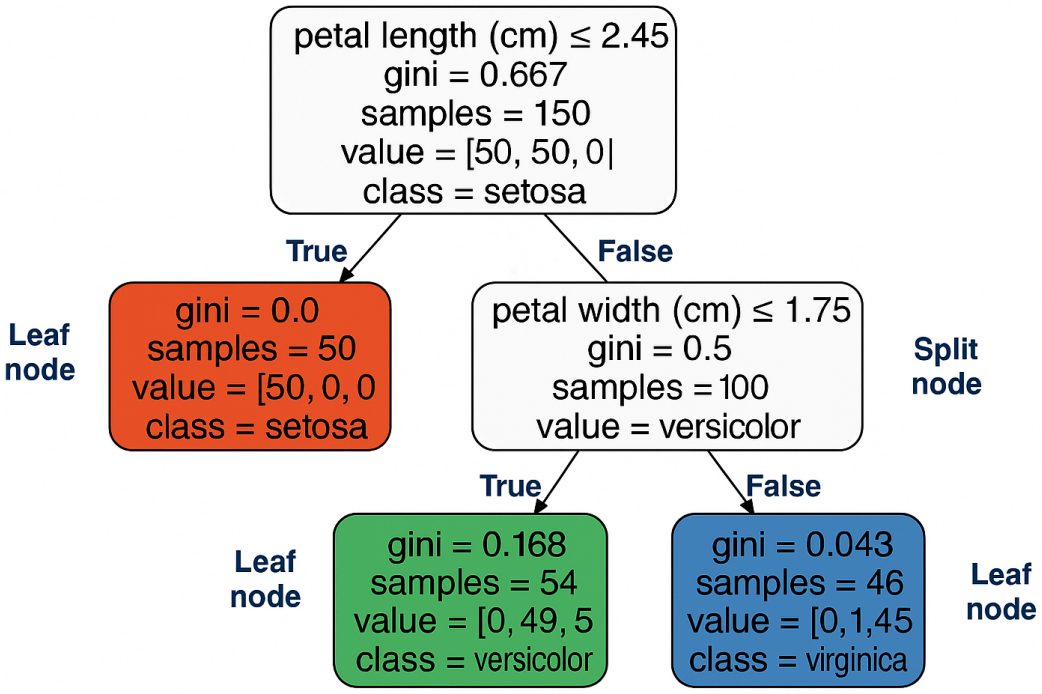

Figure 2: Basic Decision Tree [10]

Here, "samples" represent how many rows of data present in the node. "Value" shows the class count in that sample dataset. For example, at the root node it can be seen "setosa", "versicolor" and "virginica" all have 50 samples. Gini is an impurity measure. Impurity is a very key term

that describes the homogeneity of the data, more impure meaning heterogeneous and purer meaning homogeneity. In theory the decision tree algorithm splits to make the nodes pure, or in other words, gini index equal to zero. The mathematics is basic as well.

$$G_t = 1 - \sum_{k=1}^{n} p_{i,k}^2 \qquad (2-1)$$

Where G_t is the gini index, $p_{i,k}$ is the probability i^{th} class to occur in the sample. For example, in red coloured node, notice the gini index equals to 0 where:

$$p_{setosa} = 1, \qquad p_{versicolor} = 0, \qquad p_{virginica} = 0 \qquad (2-2)$$

$$G_t = 1 - \left({p_{setosa}}^2 + {p_{versicolor}}^2 + {p_{virginica}}^2\right) = 1 - (1 + 0 + 0) = 0 \qquad (2-3)$$

Whereas in the green coloured node the gini index is equal to 0.168, where the calculation is as follows:

$$p_{setosa} = 0, \qquad p_{versicolor} = \frac{49}{54} = 0{,}907, \qquad p_{virginica} = \frac{5}{54} = 0{,}093 \qquad (2-4)$$

$$G_t = 1 - (0 + 0{,}907^2 + 0{,}093^2) = 0{,}168 \qquad (2-5)$$

There is another method of calculating impurity than gini, which is entropy. Entropy, like in thermodynamics where it approaches zero when the molecules are stable, it is zero when the sample dataset contains only one class. It measures the uncertainty in the node.

$$H_t = -\sum_{p_{i,k} \neq 0}^{n} p_{i,k} log_2\left(p_{i,k}\right) \qquad (2-6)$$

The concept is similar to the gini but it introduces logarithmic scale. Both impurity calculation methods are similar, but entropy penalizes mixed nodes more, making it a better choice for highly imbalanced datasets. A basic decision tree algorithm favours the split that maximizes the information gain, or in other words split is done at a feature in a parent node that minimizes the sum of the child node impurities. Information gain or impurity drop equation combines equation (2-1) or (2-6) depending on the choice of impurity calculation. In this paper, entropy is chosen, and the justification will be made.

$$Information\ Gain = H_t - \sum_{i=1}^{k} \frac{D_i}{D} * H(D_i) \qquad (2-7)$$

Referring to Figure 4, the gini impurity in the parent node in the second level on the right is 0,5. The child nodes gini impurities are 0,168 and 0,043, leading to an information gain of 0,3337. On the other hand, since entropy is very high in uniform data distribution and in the second level of the tree this uniformness changes, the information gain is calculated as 0,9183. Even though the choice of impurity measure does not have a huge impact on model performance, entropy will be used because of its nature to notice small changes in class distribution. To summarize, it can be said that the best split is the one that maximizes the information gain.

$$\xi_i = argmax(Information\ Gain)_i \qquad (2-8)$$

Where ξ_i denotes the split in the ith node. Now that perhaps the most important concepts are covered, it is time to mention the hyperparameters in decision tree algorithm. Like in many machine learning algorithms, decision tree has its own parameters to control the construction of the tree. One may tweak with the parameters and construct more complex or more balanced trees. Some major parameters are:

- Criterion: Selects the impurity measure between gini and entropy, the default is gini.
- Max_features: Maximum number of features that are evaluated for splitting at each node.
- Max_leaf_nodes: maximum number of leaf nodes
- Min_samples_split: Minimum number of samples a leaf node must have to be created.
- Min_samples_leaf: Minimum number of samples a leaf node must have to be created.

These parameters and some others not mentioned here, represent regularization techniques applied in other machine learning algorithms. Regularization is a penalizing element introduced to an algorithm to avoid overfitting. It may be suitable to give an example of regularization: When dealing with linear regression, the aim is to draw the regression line as close as possible to the data points. In the process of constructing the line, error is measured to know how far the regression line is away from the data points. If the line would try to be close to each data point in the dataset, it may suffer from overfitting and fail to generalize the overall

pattern. So here comes the regularization part, where the regularization term is subtracted from the error value.

$$Total\ Error = (Averaged\ Square\ Error) + \lambda * (Slope^2) \qquad (2-9)$$

The parameters above are specified before the tree is constructed, so they are called "pre-pruning". Pruning is a convenient and intuitive word since real trees are pruned in order to maintain a healthy growth and eliminate ill or growth-blocking branches/leaves. In the methodology section, the information given will be put in the action in detail.

There are three main considerations in decision tree algorithms. The first one being the selection of the splits, the second one being when to decide a node is a leaf or parent and thirdly assigning the leaf node to a class [11]. To have an insight and outlook how well a decision tree is performing, there are some evaluation metrics. The main ones for decision tree-based algorithms are accuracy, precision, recall, f1-score, confusion matrix and ROC curve. Accuracy measures the percentage of correctly classified samples.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \qquad (2-9)$$

F1 Score is the primary metric in this paper for capturing the performance of dealing with an imbalanced dataset which the details will be covered in Methodology section.

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall} \qquad (2-10)$$

Recall is the measure of correctly identified actual positives.

$$Recall = \frac{TP}{TP + FN} \qquad (2-11)$$

F1-Score in the equation (2-10), is the harmonic mean of precision and recall. ROC curve and area under curve shows the trade-off between true positive rate and false positive rate where AUC being equal to 1 is the ideal scenario.

Confusion matrix shows a summary of the performance of a classifier by comparing the predictions with the actual, test data. It is a two-dimensional matrix where training (predicted) and test (actual) classes are placed. The true positive (TP), false positive (FP), false negative (FN) and true negative (TN) terms are just for illustration as they are applicable only to binary class environment. As it can be seen in the table, the value 45 at the top left side shows the algorithm labelled 45 instances of data to be class 1 that is also class 1 in test data. The value 0 means the algorithm didn't make any mistake by wrongly labelling the data instance as class 1 even in reality that instance belongs to class 2. Whereas the value 5 at the top right corner tells that the algorithm classified 5 data points as class 2 while they were class 1in test data.

Table 1: Confusion Matrix

		Predicted	
		Class 1	Class 2
Actual	Class 1	45 (TP)	5 (FN)
	Class 2	0 (FP)	10 (TN)

2.2.5. Ensemble Method – Random Forest

Decision tree, as the name suggests, is made up of a single tree. A single decision tree may be prone to overfitting as it tries to address every data instance. Although they are relatively easy to interpret, visualize and present decision trees tend to stay in the 'interpretability' side of the accuracy vs interpretability trade-off. Ensemble methods of decision trees on the other hand, tackles this limitation by combining multiple decision trees to form a predictor. Two ensemble methods will be explored in this paper which are Random Forest and XGBoost, which is a method of gradient boosted trees. Random forests by building many decision trees on different bootstrap samples of the data and then combining their outputs. This process is very important as a concept and is known as bagging, i.e. bootstrap aggregating. In other words, while a tree is growing, a subset of input variables is selected randomly to increase the randomness and robustness in its evolving decision nodes. Being an ensemble of decision trees, random forests, by nature, enable the model to learn the non-linear relations better and tend not to overfit. Trees in the ensemble votes for the most popular class, and with the growth of each tree, random forests form. The reason that the random forests do not tend to overfit comes from the concept of law of large numbers. This can be explained in a rather simple way. A single decision tree may have overfit the training data or likewise five or ten

decision trees may have common errors of fitting to the noise, but when 100 or 200 decision trees are added together, the overfitting disappears as the majority votes gained from meaningful data take over the noisy decisions. The accuracy of random forests depends on the individual tree performances and the dependence between them. If the correlation, dependence between them is not strong, then each valuable and unique information of the trees construct a healthier random forest.

$$mr(X,Y) = P_\theta(h(X,\theta) = Y) - max_{j \neq Y} P_\theta(h(X,\theta) = j) \qquad (2-12)$$

Where h(X, θ) denote the class predicted by one tree grown with randomness θ. $mr(X,Y)$ represents the vote fraction for the true class subtracted by the largest fraction for any wrong class. A more positive margin means more confident prediction, whereas a negative margin implies a misclassification. Averaging the margin values, taking the expectance of them results in strength which is shown in below equation (2-13)

$$S = E_{X,Y}[mr(X,Y)] \qquad (2-13)$$

Also, correlation in random forest means how similar the prediction of the trees are. Combining correlation and strength, a measurement called generalisation-error bound emerges.

$$PE \leq \frac{p * (1 - S^2)}{S^2} \qquad (2-14)$$

This error bound translates as follows: Increasing the strength, the information gain in each tree, lowers the error by increasing the denominator. Whereas decreasing the correlation, which is done by bootstrap sampling and random feature selection of features, decreases the generalisation-error bound lower [20].

2.2.6. Ensemble Method - XGBoost

While a decision tree is simply a single tree made of varying number of branches, and random forest is an ensemble of decision trees that convolute to a single classifier, boosting is a method in which shallow trees are sequentially trained on top of each other. After the first tree is trained, the residuals (gradients) affect how the next tree is trained, and this process

continues until the pre-specified number of trees are trained. In the figure below, the flow is illustrated where each tree is formed using the residual of the previous tree, resulting in a final prediction.

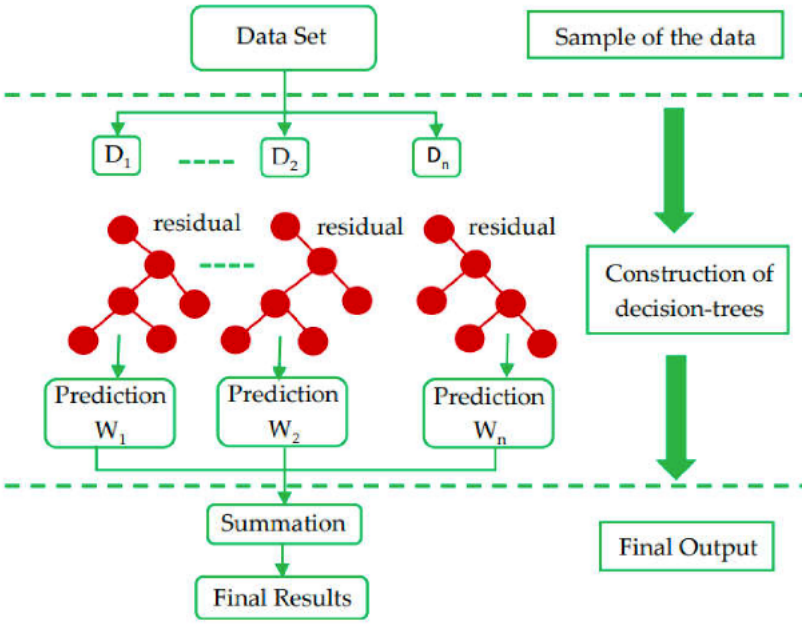

Figure 3: XGBoost Process [21]

XGBoost evaluates splits by checking how much they reduce the current loss. The equation of reduction of loss, which is obtained by splitting a node, and the additive nature can be illustrated as the below equations [22].

$$\widehat{y_i^{(t)}} = \widehat{y_i^{(t-1)}} + \eta f_t(x_i) \qquad (2-15)$$

$$Gain = \frac{1}{2}\left[\frac{(|G_L| - \alpha)^2}{H_L + \lambda} + \frac{(|G_R| - \alpha)^2}{H_R + \lambda} - \frac{(|G_L + G_R| - \alpha)^2}{H_L + H_R + \lambda}\right] - \gamma \qquad (2-16)$$

XGBoost also stands out with its regularization methods to prevent overfitting to the noise. Each split occurs if the gain is positive after subtracting γ (gamma) parameter. This parameter also contributes to prevent overfitting as it reduces the complexity. The α (alpha) parameter introduces the L1 regularization, where the gradient should be higher than α, if specified, to contribute to the gain. Also, min_child_weight parameter in the classifier controls whether the split should be considered by setting a threshold for H_L and H_R, the Hessian sum, so that if they are below a specified value, the split is not to be considered.

Lastly, λ (lambda) parameter is L2 regularization term that penalizes the value of large leaf weights therefore preventing overfitting. These parameters work together to find ways of training the model by the magnitude of reduction in loss. In summary, some major and used parameters of XGBoost classifier are:

- Learning_rate: Comes into play after the calculation of gain, affecting the leaf weights for the next tree.
- Gamma: Minimum loss reduction required to make a further partition on a leaf node of the tree.
- Max_depth: Maximum depth of a tree.
- Min_child_weight: Minimum sum of instance weight (hessian) needen in a child.
- Lambda: L2 regularization
- Alpha: L1 legularization

3. Methodology

Not only finding real world data is hard but also finding one that is easy to interpret and publishable one is very challenging. Dealing with raw data have its own challenges about cleaning and turning them in to meaningful datasets. Moreover, the goal is to evaluate and explore constructing accurate decision-tree based algorithms. Due to these reasons the synthetic dataset that is found in Institute of Electrical and Electronics Engineers (IEEE) is used for CNC machining process. It is necessary to mention that in this section, the aim is to take the reader from a fresh start to the end regarding data and machine learning aspect. To do a thorough analysis, the methodology section from now on will have this outline:

- Introduction of the data
- Feature engineering and “vanilla” decision tree
- Further feature engineering
- Parameter investigation for decision tree
- Finding the optimal setting for decision tree
- Applying random forest to the current setting of decision tree
- Investigating random forest capabilities
- Exploring XGBoost

While exploring what can be done with the data and machine learning models, the paper will provide insights and explanations why some actions are useful, and some are not. Let’s

start with knowing the data. It has 10.000 readings of which there are 8 features and 5 failure types. The features are:

1- UDI: Specifies the index, starts from 1 and ends at 10.000
2- Product ID: It's combination of a unique product number and the type of material denoted by 'L', 'M' or 'H'.
3- Type: Has three distinct letters which are L, M and H. They denote the quality of the material and low materials make up 50 % of all products, medium materials make up for 30 % and high-quality materials make up 20 % of all the materials.
4- Air Temperature (K): Generated using a random walk process later normalized to a standard deviation of 2 K around 300 K
5- Process Temperature (K): Generated using a random walk process normalized to a standard deviation of 1 K, added to the air temperature plus 10 K.
6- Rotational Speed (RPM): Calculated from a power of 2860 W, overlaid with a normally distributed noise
7- Torque (Nm): Values are normally distributed around 40 Nm with σ = 10 Nm and no negative values.
8- Tool Wear (min): The quality variants H, M, L add 5, 3, 2 minutes of tool wear to the used tool in the process.

One should mention that the failures are independent of each other. The failure types are:

1- Tool Wear Failure (TWF): The tool is either replaced or fail at a randomly selected tool wear time between 200-240 mins, which happens 120 times in the dataset.
2- Heat Dissipation Failure (HDF): When the temperature difference between air and process temperature is lower than 8,6 K and tool's rotational speed is below 1380 rpm, heat dissipation causes a failure. This happens in 115 samples.
3- Power Failure (PWF): It is known that power is equal to the product of torque and rotational speed. When the power is below 3500 W or above 900 W, the process fails. This is the case for 95 samples in the dataset.
4- Overstrain Failure (OSF): Whenever the product of tool wear and torque exceeds the thresholds—11.000 minNm for low quality, 12.000 minNm for medium quality, and 13.000 minNm for high quality—the process fails due to overstrain. This condition occurs 98 times in the dataset.
5- Random Failures (RNF): Every case in the dataset has a 0,1 % chance to fail regardless of any other parameters. This failure happens 19 times.

So, in total, 339 datapoints have some kind of failure type and it is apparent that the data is highly imbalanced. This fact will be in consideration throughout the paper and some measures will be taken to overcome the discrepancies caused by imbalance, along with exploring decision tree-based algorithms and their techniques. A sample of data is provided in the table below.

Table 2: Sample of the Data

UDI	Product ID	Type	Air temperature [K]	Process temperature [K]	Rotational speed [rpm]	Torque [Nm]	Tool wear [min]	Machine failure	TWF	HDF	PWF	OSF	RNF
1	M14860	M	298.1	308.6	1551	42.8	0	0	0	0	0	0	0
2	L47181	L	298.2	308.7	1408	46.3	3	0	0	0	0	0	0
3	L47182	L	298.1	308.5	1498	49.4	5	0	0	0	0	0	0
4	L47183	L	298.2	308.6	1433	39.5	7	0	0	0	0	0	0
5	L47184	L	298.2	308.7	1408	40.0	9	0	0	0	0	0	0
6	M14865	M	298.1	308.6	1425	41.9	11	0	0	0	0	0	0
7	L47186	L	298.1	308.6	1558	42.4	14	0	0	0	0	0	0
8	L47187	L	298.1	308.6	1527	40.2	16	0	0	0	0	0	0
9	M14868	M	298.3	308.7	1667	28.6	18	0	0	0	0	0	0
10	M14869	M	298.5	309.0	1741	28.0	21	0	0	0	0	0	0
11	H29424	H	298.4	308.9	1782	23.9	24	0	0	0	0	0	0
12	H29425	H	298.6	309.1	1423	44.3	29	0	0	0	0	0	0
13	M14872	M	298.6	309.1	1339	51.1	34	0	0	0	0	0	0
14	M14873	M	298.6	309.2	1742	30.0	37	0	0	0	0	0	0
15	L47194	L	298.6	309.2	2035	19.6	40	0	0	0	0	0	0
16	L47195	L	298.6	309.2	1542	48.4	42	0	0	0	0	0	0
17	M14876	M	298.6	309.2	1311	46.6	44	0	0	0	0	0	0
18	M14877	M	298.7	309.2	1410	45.6	47	0	0	0	0	0	0
19	H29432	H	298.8	309.2	1306	54.5	50	0	0	0	0	0	0
20	M14879	M	298.9	309.3	1632	32.5	55	0	0	0	0	0	0

3.1. Feature Engineering and Train Test Split

Now that the data is introduced, it is time to start with a small feature engineering and explore the algorithm. Decision tree classifier, like other machine learning algorithms, require input data to be numeric for processing. Decision trees make binary splits based on numerical comparisons or categorical encodings. Therefore, string values must be converted to numerical values to allow the model to learn. This conversion of data is considered as feature engineering, and this will be one of several feature engineering that will be done in this paper. First, related libraries are imported, and the data is loaded in the figure below.

```
ning_dt.py > ...
  import pandas as pd
  import numpy as np
  from sklearn.tree import DecisionTreeClassifier, plot_tree
  from sklearn.model_selection import train_test_split, GridSearchCV, cross_val_score
  from sklearn.preprocessing import OneHotEncoder
  from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, f1_score, make_scorer
  import helper_dt as h
  import matplotlib.pyplot as plt
  from imblearn.over_sampling import SMOTE

  folder = '/Users/cemkarahan/Desktop/proje/data/'
  df = pd.read_feather(folder + 'thesis_data.fth') # Reading and loading the data
```

Figure 4: Importing All the Libraries and Loading Data

Now that the data is loaded, it's time to select the feature and response variables and do the necessary conversions. Recall the "Type" feature in the dataset; it has "L", "M", "H" values in it. First, these values should turn into numeric. Then, to allow the tree to consider each category independently, one-hot encoding will be applied. This will create Type_L, Type_M and Type_H, each defined as 1s and 0s according to the original "Type" dimension. Also, 'UDI' and 'Product ID' are not required and have no meaningful information for the model as per this paper's focus. So, these two dimensions are dropped out in this part. It is essential and mandatory that the feature values should be in a single column since a multiclassifier (a single instance corresponding to multiple failure types) is not used. To achieve this, 'Safe' value will be introduced to this field, representing normal operation and all the other types of failure such as "HDF" and "OSF" will be written as field values of the dimension "Failure Type". As a final step, to enhance feature-response behaviour, the absolute difference between air and process temperature is computed, leading to more direct relation between heat related failures. The corresponding code block is in the below figure.

```
###### One Hot Encoding - Type ######
ohe = OneHotEncoder(sparse_output = False)
encoded_features = ohe.fit_transform(df[['Type']])
df_encoded = pd.DataFrame(encoded_features, columns = ohe.get_feature_names_out(['Type']))
df_encoded['Product ID'] = df['Product ID'].values
df = df.merge(df_encoded, how = 'left', on = 'Product ID')

df.drop(columns = {'UDI', 'Product ID'}, inplace = True) # Dropping unnecessary fields

df['Dumy'] = df['TWF'] + df['HDF'] + df['PWF'] + df['OSF'] + df['RNF']
df['Safe'] = np.where(df['Dumy'] > 0, 0, 0.5)
df['Failure Type'] = df[['Safe', 'HDF', 'OSF', 'PWF', 'RNF', 'TWF']].idxmax(axis = 1) # Combining the failure types into one column
df['Temperature Delta'] = abs(df['Process temperature [K]'] - df['Air temperature [K]'])
```

Figure 5: Feature Engineering, Preparing the Data

Now it's time to get the data ready to be used in the model. First, the feature and the response variables are separated as x and y, continued by partitioning x and y as train and test sets to be readied for training the model. The randomized or cross-validated split of training and testing sets has been adopted as the standard of machine learning for decades. The establishment of these split protocols are based on two assumptions: (i)-fixing the dataset to

be eternally static so we could evaluate different machine learning algorithms or models; (ii)- there is a complete set of annotated data available to researchers or industrial practitioners. [1] The reason for splitting data into training and test subsets is to ensure the subset used in training the model is separate from the data use to evaluate its performance. If the same data points were included in both training and testing phases – known as data leakage- the model's evaluation metrics (e.g. accuracy, precision, recall) could become artificially optimistic. This happens because the model may overfit the training data, memorizing specific examples rather than learning general patterns. So, to overcome these unwanted circumstances and analyse different models with the same train and test subset, it is the best to split the data.

To perform the split, although scikit-learn's "train_test_split" function could be used, a more realistic evaluation could be achieved using nested cross-validation. First, using "StratifiedKFold" from scikit-learn, 5-fold of train and test data, i.e. outer split, was generated. Then, in a for-loop, each of the folds were used one at a time, ensuring the test data was never leaked into the training. This way, instead of evaluating just one test split, all results of the test data could be concatenated to have the whole dataset as an evaluation. The code written is in the Figure 5, below.

```
y_true_list = []
y_pred_list = []
dt_params = dt_params or {}
smote_params = smote_params or {}
outer_cv = StratifiedKFold(n_splits = 5, shuffle = True, random_state = 42)

for fold, (train_idx,test_idx) in enumerate(outer_cv.split(x, y), 1):
    x_train, y_train, x_test, y_test = split_data_cv(x, y, train_idx, test_idx)
    dt = DecisionTreeClassifier(**dt_params)
    dt.fit(x_train, y_train)
    y_pred = dt.predict(x_test)

    y_true_list.append(y_test)
    y_pred_list.append(y_pred)

y_true_all = np.concatenate(y_true_list)
y_pred_all = np.concatenate(y_pred_list)
print("Classification Report Test:\n", classification_report(y_true_all, y_pred_all, zero_division = 0))
labels = ['Safe', 'HDF', 'OSF', 'PWF', 'TWF']
test_cm = confusion_matrix(y_true_all, y_pred_all, labels = labels) # Confusion matrix preperation
h.print_multiple_class_confusion_matrix(test_cm) # Confusion matrix, layout
```

Figure 6: Splitting and Evaluation of Dataset

The parameters of "StratifedKFold" is selected so that of the data is allocated for training and the rest 30% is used for training. The "random_state" parameter is used to control the randomness of the data split. In this paper, "random_state" is specified as 42 to ensure

reproducibility. Each time the code is run, the dataset will be split into the same training and testing subsets. Without setting this parameter, the split would differ on each execution which would make the results difficult to replicate.

```
x = df[['Type_H', 'Type_M', 'Type_L', 'Air temperature [K]', 'Process temperature [K]', 'Rotational speed [rpm]',
        'Torque [Nm]', 'Tool wear [min]']]
y = df['Failure Type']

x_train, x_test, y_train, y_test = train_test_split(x, y, train_size = 0.7, random_state = 42)

model = DecisionTreeClassifier(random_state = 42)
model.fit(x_train, y_train)
```

Figure 7: Identifying X, Y and Splitting Train - Test Data

3.2. Default Decision Tree

Decision tree classifier is a tool that can be found in scikit-learn library. After simply importing the tool, decision tree classifier can directly be used by providing correctly formatted “x” and “y”, i.e. the features and classes, to the model. A vanilla decision tree means the most basic implementation of the decision tree algorithm, without tuning hyperparameters or applying pruning. The reason this is emphasized in this paper is because this classification method is the foundation of all the tree-based algorithms such as ensemble and boosting.

```
model = DecisionTreeClassifier(random_state = 42)
model.fit(x_train, y_train)

print(model.get_params()) # Prints the parameters of the model

y_test_pred = model.predict(x_test) # Model prediction based on the unseen, test data

print("Accuracy Test:", f1_score(y_test, y_test_pred, average = 'macro'))
print("Classification Report Test:\n", classification_report(y_test, y_test_pred))

labels = ['Safe', 'HDF', 'OSF', 'PWF', 'RNF', 'TWF']
test_cm = confusion_matrix(y_test, y_test_pred, labels = labels) # Confusion matrix preperation

h.print_multiple_class_confusion_matrix(test_cm) # Confusion matrix, layout
```

Figure 8: Default Decision Tree, Evaluation Metrics and Confusion Matrix

This model will give a baseline performance of the training data. After that, evaluation and future steps will be taken. The only parameter that will be set is “random_state” because this parameter will ensure randomness will be the same throughout the code and the results, in the same fundamentals with train-test-split. Once the model is trained, the code will print the parameters that are processed in this model. To grasp the performance of the model, a library called Pretty Table will be used to visualise confusion matrix, and sklearn’s classification report layout will include precision, recall, accuracy and most importantly f1-score. The last observatory steps are constant throughout the paper and sometimes tree figure will be added.

Table 3: Evaluation Metrics - Summary

label	precision	recall	f1-score	support
HDF	0,96	0,92	0,94	25
OSF	0,83	0,68	0,75	28
PWF	0,78	0,58	0,67	24
RNF	0	0	0	7
Safe	0,99	0,99	0,99	2903
TWF	0	0	0	13
Accuracy			**0,97**	3000
Macro Avg	0,59	0,53	0,56	3000
Weighted Avg	0,98	0i97	0,97	3000

Table 3 includes both averages of the classes and individual class performances. First of all, because the dataset is highly imbalanced, weighted average is a poor metric. Macro averages of the metrics, however, are an important indicator for the performance of the algorithm.

Table 4: Confusion Matrix of Vanilla Decision Tree

Confusion Matrix	**Pred: Safe**	**Pred: HDF**	**Pred: OSF**	**Pred: PWF**	**Pred: RNF**	**Pred: TWF**
Actual: Safe	2864	1	4	3	12	19
Actual: HDF	1	23	0	1	0	0
Actual: OSF	9	0	19	0	0	0
Actual: PWF	10	0	0	14	0	0
Actual: RNF	7	0	0	0	0	0
Actual: TWF	13	0	0	0	0	0

In the Table-3, under the f1-score and to the right of the accuracy, accuracy value seen as 0,97. Accuracy value being so high is understandable since, looking at the confusion matrix

out of 3.000 samples, 2864 + 23 + 19 + 14 = 2920 values are correctly classified of which 2.864 belong to the class safe. In the meantime, recall came out to be low because there are not a lot of failure instances in the data and nearly half of them were wrongly classified. This can also be seen from the precision values of each class. “Safe” and “HDF” classes were nearly successfully labelled. “PWF” and “OSF” are mediocre, but the rest are relatively low. When this observation is enhanced with confusion matrix, it can be seen that TWF (tool wear) and RNF (random) are the weak spots of the algorithm. With exploration of tools, the classification performance of OWF and PWF can be improved, while HDF looks promising.

This analysis translates as the algorithm isn’t very effective when labelling instances as failures. With no meaningful feature engineering and without any pruning, decision tree still looks promising for further development.
Before any improvement on algorithm or feature engineering, it is wise to look at the correlations of the features with responses and see the top contributors.

Table 5: Top 5 Correlations

Class	Feature	Correlation
HDF	Temperature Delta	0,191
OSF	Torque [Nm]	0,183
OSF	Tool wear [min]	0,156
HDF	Torque [Nm]	0,143
PWF	Rotational speed [rpm]	0,123

Based on Table 3-4-5, RNF appears to behave randomly, as the base algorithm couldn’t label any of it successfully and find meaningful correlation with any of the features. On the other hand, looking at correlations, further feature engineering could be applied to improve the labelling of classes. On a positive note, creation of the feature “Temperature Delta” is justified as it showed the highest correlation among others. Notably, highest correlated features are associated with “OSF”, “HDF” and “PWF”, and this is inspiration for engineering some new parameters. These parameters are Overstrain parameter, which is the product of torque, tool wear and type; Power parameter that is the multiplication of torque, rotational speed and type; the Heat parameter consisting of difference between process and air temperature and multiplication with type; Tool parameter that is the product of tool wear

and type. These parameters might capture the compound effect of each other on some classes. In fact, when the algorithm in Figure-8 is computed again with only changing the feature list, it resulted with improvement of the average F1-score from 0,57 to 0,65. Having introduced the vanilla decision tree and applied feature engineering with unbiased algorithm, the features and the evaluations have been established for the remainder of the paper.

```
types = ['L', 'M', 'H']
for i in types: # Only for Overstrain Parameter
    df[f'Overstrain Parameter_{i}'] = df['Torque [Nm]'] * df['Tool wear [min]'] * df[f'Type_{i}']

for i in types: # Only for Power Parameter
    df[f'Power Parameter_{i}'] = df['Torque [Nm]'] * df['Rotational speed [rpm]'] * df[f'Type_{i}']

for i in types: # Only for Heat Parameter
    df[f'Heat Parameter_{i}'] = (df['Process temperature [K]'] - df['Air temperature [K]']) * df[f'Type_{i}']

for i in types: # Only for Tool Parameter
    df[f'Tool Parameter_{i}'] = df['Tool wear [min]'] * df[f'Type_{i}']
```

Figure 9: Creation of the New Parameters

3.3. Decision Tree – Tuning Parameters

As mentioned before, entropy would be the chosen criterion for impurity measurement. With all parameters and features same, only changing criterion to ‘entropy’, which is simply done by adding “criterion = ‘entropy’” argument to the DecisionTreeClassifier function in Figure-6, increased average F1-score from 0,65 to 0,68.

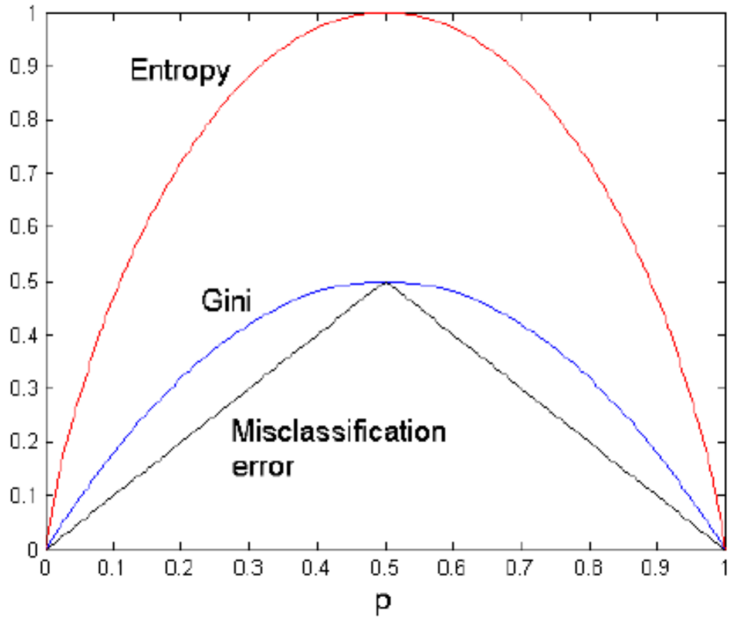

Figure 10: Comparison of Impurity Measures vs Class Probability

The slight improvement in the F1-score is the main justification for using entropy. This is because entropy is more sensitive to slight imbalances in the data than gini, allowing it to initiate splits as a result. This setup will be called as the base decision tree in following parts.

Another way to address the imbalanced dataset would be oversampling. Oversampling means synthetically producing samples, with a purpose of increasing instances the minority classes, balancing the dataset and in turn helping the model to understand the patterns. There is one more way of altering the models learning without adding or removing data which is assigning class weights. This can be done via specifying the 'class_weight' hyperparameter in scikit's decision tree classifier function. It enables the user to influence the model by emphasizing the classes that have higher valued weights. Finding the optimal setting for oversampling or class weights can be very hard to find manually. Instead, methods like grid search or Gaussian optimization may be used. There is a tool called 'GridSearchCV' which performs an exhaustive search over specified parameter values, allowing multiple parameter values to be checked over many iterations. However, this method can be computationally expensive as checked parameters and the values searched increases. For example, in the case of the dataset in this thesis, there are 11 values for 5 parameters meaning 11^5 iterations that equals 161.051, which is a challenge for a single laptop or computer that is presently available. Instead of such a greedy algorithm, Bayesian optimizations was used. The working logic of Bayesian optimization can be outlined like this: A surrogate function (prior) maps the relationship between class-weights and F1 score as a probability distribution. Then acquisition function iteratively selects the most promising weight combination and with these values, the model is trained and the resulting F1 scores update corresponding values in the first assumption. This creates the posterior and then acquisition function again chooses some combinations, the same dataset is trained with chosen parameters, the posterior is updated, and the iterations go on like this. In each iteration, the posterior functions' distribution comes closer to finding the best weight combinations by creating global maximum in its distribution. This process can be seen in Figure 10 and 11. In result, the algorithm makes a couple of dozen iterations instead of tens of thousands made in grid search method. [17]

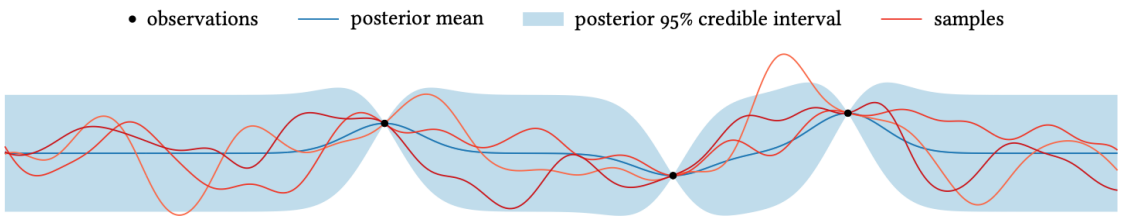

Figure 11: The Posterior Process

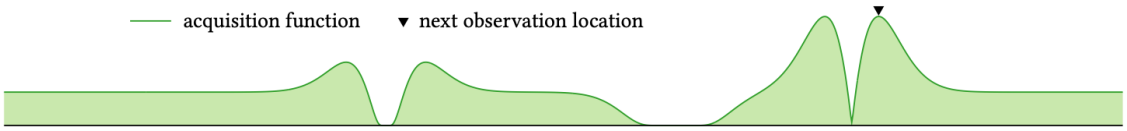

Figure 12: Acquisition Function Picking New Sample Points

Regardless of which method is used for finding the correct parameters, cross-validation should be used as it creates randomness in evaluating during exploration. In this paper, n_fold and scoring values are chosen to be 5 and F1-score, respectively. This will mean the data is split into 5 equal sizes and in each iteration, 4/5 of these splits are used as training and the rest 1/5 is used as test. Number of folds also means the number of iterations, so during 5 iterations all the splits will be used as train/test this way, ensuring randomness. To determine whether using or not using SMOTE (oversampling method), class weights and post-pruning, the following method was tried, using Bayesian optimization with cross-validation:

1. Parameters of SMOTE were explored and optimal values were tested with the original train/test dataset.
2. With the established setting of SMOTE, class weight combination was found and tested with the original train/test data.
3. Class weight combination was found without applying SMOTE and tested with the original train/test data.
4. Then two setups were chosen, one being the base decision tree where only entropy and random state are specified, and the other being the combination of class weights and oversampled data. After applying post-pruning two these two setups, the ultimate choice became the post-pruned tree with the base decision tree The reasoning during these steps is explained below.

At the first step, it was observed that SMOTE resulted in a decrease in F1-score, relative to the base decision tree. This may have been caused by increasing the instances of noise and misleading the algorithm to pay attention to non-existing relation between feature and failures. After oversampling, tuning class weights was explored which does not alter the training data but instead is tuned inside the algorithm settings. By default, class weights are set to 1, meaning each class is treated equally during training. If class weights of some or all classes were to change, then the model emphasizes the higher weighted classes more, pays more attention to the attributes of them.

```
results = []
for fold, (train_idx, test_idx) in enumerate(outer_cv.split(x, y), 1):
    x_train, y_train, x_test, y_test = ft.split_data_cv(x, y, train_idx, test_idx)
    x_train = x_train.reindex(columns=x.columns, fill_value=0)
    x_test  = x_test .reindex(columns=x.columns, fill_value=0)

    # 3) run BayesSearchCV on a DecisionTreeClassifier
    bayes_cv = BayesSearchCV(WeightedDecisionTreeClassifier(random_state=42),
        search_spaces = param_space,
        scoring = minority_auc_scorer,   # your custom AUC scorer
        cv = inner_cv,
        n_iter = 80,
        n_jobs = 4,
        refit = True,
        random_state = 42,
        verbose = 1
    )
    bayes_cv.fit(x_train, y_train)

    # 4) evaluate on the held-out fold
    best = bayes_cv.best_estimator_
    y_pred = best.predict(x_test)
    f1_macro = f1_score(y_test, y_pred, average='macro', zero_division=0)
    auc_score = bayes_cv.best_score_

    # 5) append exactly the same way you did before
    results.append({
        'fold':      fold,
        **bayes_cv.best_params_,
        'f1_macro':  f1_macro,
        'auc':       auc_score,
    })
df_results = pd.DataFrame(results)
print(df_results)
```

Figure 13: The Code for Searching Class Weights Using Bayesian Optimization

Here and in the next Bayesian search functions, for the goal of focusing on the performance of minority classes, a custom area-under-curve (AUC) is created. AUC is a metric that give intuitions about the performance of the algorithm for each class, ranging from 0.5 to 1.0. The function is shown in the Figure 14, and the weight of “Safe” was deliberately reduced to better asses the minority classes performance.

```python
def minority_auc_scorer(estimator, x, y_true):
    # Predict class probabilities
    y_prob = estimator.predict_proba(x)

    # Build sample weights: Safe=0.1, failures=1.0
    w = np.where(y_true == 'Safe', 0.1, 1.0)

    # Compute weighted AUC
    return roc_auc_score(
        y_true,
        y_prob,
        average="weighted",
        multi_class="ovr",
        sample_weight=w
    )
```

Figure 14: Custom AUC Scorer Function

Table 6: Example of Class Weight Distribution Based on Bayesian Optimization

Fold	hdf_w	osf_w	pwf_w	safe_w	twf_w	f1_macro	auc
1	9,05	20	20	5	1	0,78	0,93
2	13,54	10,15	20	4,05	20	0,76	0,93
3	1	5,91	1	4,51	1	0,8	0,93
4	20	20	13,61	0,95	1	0,8	0,94
5	11,9	1,55	20	4,9	1	0,77	0,93

The “f1_macro” in the Table 6 represents the result of the each 5-fold cross-validated test, trained on 5-fold cross-validated inner training, not the actual value of the decision tree classifier that is trained on a single train data, closer to a real application. However, cross validation does indeed provide a close estimation of the performance of parameters. The parameters in Table 6 are obtained using the same method, doing oversampling and not being the only difference. When the parameters in Table 6 were used, F1 score increased with weight class alteration for both oversampled and the original training dataset, with the oversampled one being slightly higher. At this point, the combination of class weights and oversampling seemed to present promising outcome. After these results, the final comparison was to be done by applying post-pruning to the last (oversampled & weighted and only weighted) versions applied to original and oversampled training dataset. But first, it is necessary to explain what post-pruning is and what motives lead to apply post-pruning.

Exploring the parameters of the decision tree classifier before training the data lies in the category of pre-pruning. This may result in fitting the data better but also it may not. Moreover, the trade-off between bias and variance should be kept in a sensible range. Bias means the algorithm tends to give a constant prediction without affected by the data, whereas variance means the algorithm tries to address all the data in the training set. It is important to keep a balance of this trade off to minimize the error in the prediction, as the equation of error is shown in below.

$$Error(X) = Variance + Bias^2 + Irreducible\ Error \qquad (2-12)$$

In decision trees, the variance translates to increasing tree depth or not limiting the growth of tree. The bias refers to a relatively smaller, simpler tree where the tree does not address every data in the training set. Perhaps the safest option to keep bias-variance trade-off is to use the post-pruning tool. This important trade-off lies in the mathematic nature of post-pruning process. Pruning allows a large tree to grow without stopping it and then prune it upward. By calculating cost complexity of each subtree, it selects the parent nodes whose cost complexity is less than the nodes that precede and prunes them so that the parent node is now a leaf node. Cost complexity is given by:

$$R_\alpha(T_i) = R(T_i) + \alpha * |T| \qquad (2-13)$$

Where R(T) is the total impurity of the tree/node. α is the complexity factor that controls how much to penalize complexity. |T| is the number of leaf nodes in the tree. Here, increasing or decreasing the complexity factor affects the bias-variance trade-off since it penalizes the number of leaves. Increasing alpha pushes the algorithm to have simpler, smaller trees thus having bias over variance. It was observed that as the tree is pruned upward, the entropy of leaf nodes initially starts from 0 and then increases due to loss of information. Usually, as a tree is pruned upward, the overfitting decreases but then the model's impurity increases as pruning continues meaning bias overwhelms variance. However, there is a good balance where some unnecessary branches get pruned and the model doesn't overfit eanymore.

The figure below shows entropy vs pruning level, comparing the three decision tree settings. The blue line refers the model trained on original data, with parameters obtained from cross validation on oversampled data. The orange line represents the model trained on oversampled data and whose parameters obtained from cross validation on oversampled data. The green line represents the model trained on original data, with parameters tuned using cross validation on original data.

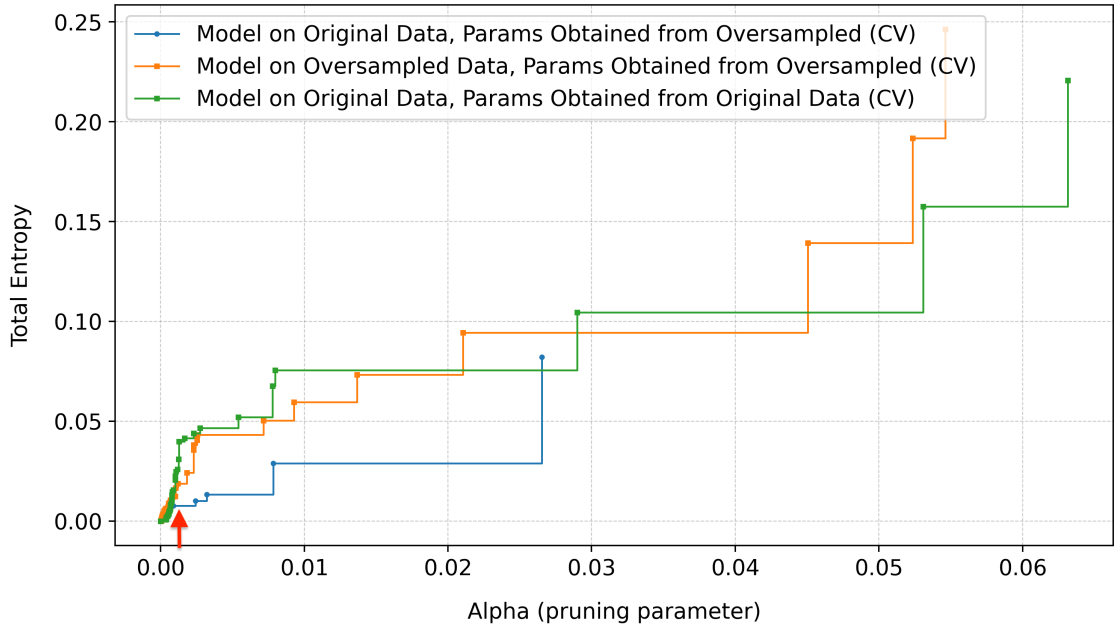

Figure 15: Entropy vs Cost-Complexity Factor

Alpha values between 0 and 0. After the red arrow that is marked in the plot, the three models differentiate from each other. The entropy in green and orange lines continue to increase after the arrow and elimination of sub-trees during this process caused valuable information loss. The bottom branches of the tree were effectively pruned at around alpha values corresponding to a range of 0.00001 and 0.001. It is visible that the green line significantly had a higher entropy during pruning. Although the orange line's entropy values seemed to be lower than the green line's, the model represented by blue line is far off better than these two. Also, the plateau after the first, and also frequent, pruning expresses the model does not lose valuable information even the alpha value increases. These insights are parallel to the classification performances and confusion matrix of these models. The green and the orange models show decreasing classification performance when post pruning starts, leading to a conclusion that these models may have overfitted the training data. The poor performance may be related to the nature of decision trees' likelihood of overfitting to noise. On the other hand, the blue model's performance and false positives decreased when post pruning was

applied. In the light of this information, the model that is represented by blue line was concluded as the chosen decision tree. This plot is a limited section of the whole pruning phase, where in the full plot the alpha value increases the total entropy comes closer to 1 as can be seen in the Figure 13. For informational purposes though, it is valuable and enough to see where the entropy starts to increase with a higher slope. In the means of a final check for the right value of alpha that was to be used in the final decision tree algorithm, F1 score vs alpha parameter plot was created. The plot reinforces alpha values between 0.001 and 0.005 for an optimal F1 score. The F1 scores were obtained from using cross-validation and the scoring was based on ‘PWF’, ‘OSF’ and ‘HDF’, same methodology used during Bayesian optimization. The aim is not to select the settings corresponding to highest F1 score but to find a good balance of reducing the tree size with a solid F1 score. In the case of the decision tree algorithm in question, the F1 score slightly decreased when tested on the original test data. This however was not a negative outcome since it decreased the false positives in expense of a slight decrease of F1 score. The red line in the plot corresponds to alpha values around 0.003. Hence, 0.003 was used in the settings of decision tree algorithm.

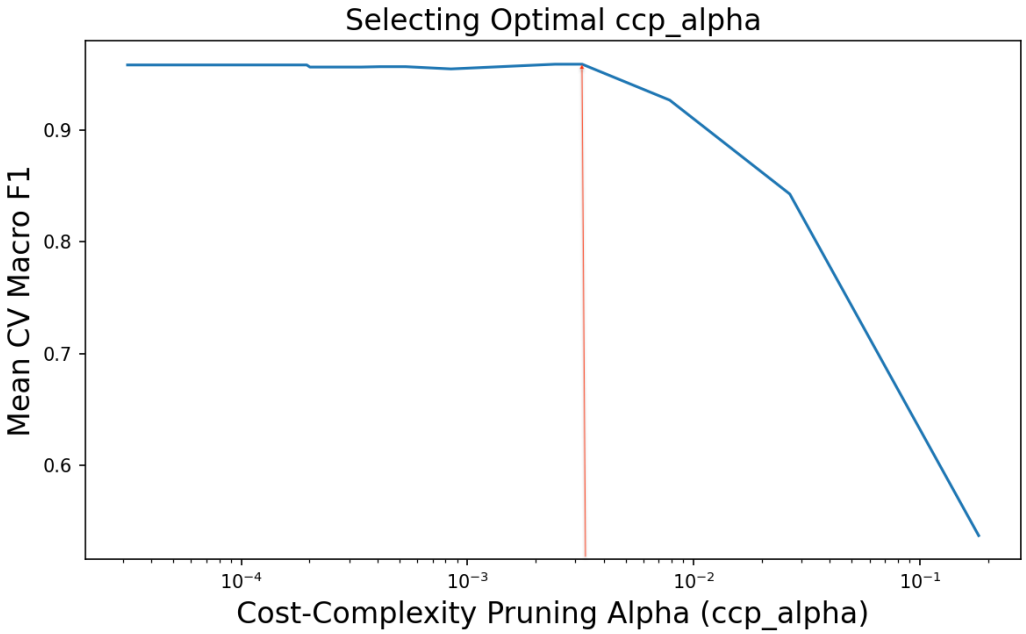

Figure 16: Macro F1 Scores vs Alpha Parameters

The findings and intermediate steps of establishing a Decision Tree is shown in more detail compared to Random Forest and XGBoost. This is due to Decision Tree being the fundamental of this concept and also the methodology being similar in Random Forest and XGBoost. The findings and the reasoning of the output of the processes will be explained in Results and Discussion section.

3.4. Random Forest

The next method that is used after the decision tree is random forests. They are ensemble of decision trees and their ability to overcome overfitting that a decision tree may be prone to. In the previous section, the results of the decision tree were promising but couldn't learn the hard-to-capture instances of the data, in the favour of avoiding overfitting. The reason to avoid overfitting was the cost of having too many false positives which may not be acceptable in manufacturing setting, as it would disrupt the processes unnecessarily. These issues are in the subject of bias-variance trade off and random forest is one possible method to tackle it. The methodology used was similar to the decision tree section. First, a baseline random forest was tested to have an insight of the algorithm behaviour. Then, class weights and sample weights options were explored. After that, oversampling and undersampling methods were explored to increase the chance of capturing the hard, borderline instances of failures. Finally, anomaly detection methods were investigated to tackle false positives and the inability to classify TWF failure.

The code structure of random forest is very similar to the decision tree. Here, as a difference, calibrator was added to increase reliability of predicted probabilities. CalibratedClassifierCV is a tool provided by scikit-learn. It generated five clones of the random forest classifier with each having different training splits, and fitted the test data five times, enabled to have more realistic alignment to a real application. The baseline, vanilla random forest showed better results in terms of eliminating false positives, especially caused by 'TWF' misclassifications. Also, this evaluation showed that bagging mechanism alone, without adding any complex tuning or methods, proved to be successful of tackling variance. The only setting that was tuned was the "n_estimators" parameter. With a quick grid-search, using 200 decision trees in the forest provided a stable F1-macro score.

```
base_rf = RandomForestClassifier(n_estimators= 200, random_state= 42, n_jobs= 8)
calibrator = CalibratedClassifierCV(base_rf, cv =5)
calibrator.fit(x_train, y_train)
```

Figure 17: Setting of Base-line Random Forest

After the base model, class weights options and then sample weights were explored. These explorations were done in the same structure that was used previously in decision trees. It is necessary to explain the difference between class weight and sample weights. Class weights are assigned in a higher level, meaning the model treats the same class weight all throughout the training process. Sample weights are the same except the user can alter the weight values in the row scale, they are constructed as vectors in the size of the training data. This could be beneficial for the cases where case specific information is known. Also, the averaging nature of random forest may assist to prevent overfitting to the data in this case. For example, it is stated by the author of the dataset used in this thesis that power failure occurs when power is below 3500 W or above 9000 W. By using sample weights, the rows that contain this information could be labelled with higher weight value to tell the model to emphasize it more. For the construction of sample weights, the code below was written.

```
wear = x_train['Tool wear'].values
heat = x_train['Temperature Delta'].values
power = (2 * np.pi / 60) *  x_train['Rotational speed'].values * x_train['Torque'].values
overstrain = wear * x_train['Torque']
rot_speed = x_train['Rotational speed'].values

mask_twf = (y_train == 4) & (wear >= 200) & (wear <= 240)
mask_heat = ((heat < 8.59) & (rot_speed < 1279))
mask_power = ((power < 3500 ) | (power > 9000))
mask_overstrain = (overstrain > 11000)

sample_weight = np.ones(len(y_train))
sample_weight[mask_heat] = hdf_weight
sample_weight[mask_power] = pwf_weight
sample_weight[mask_overstrain] = osf_weight
sample_weight[mask_twf] = twf_weight
sample_weight[y_train == 0] = safe_weight
```

Figure 18: Sample Weight Generation

The next step was to implement over-sampling and/or under-sampling methods with/without sample weights setting. First, “BorderlineSMOTE” tool was used to increase the instances that were in boundary of being “Safe” and faulty, which presumably were the hard instances for the tree to classify. Then, under-sampling method, “TomekLinks”, was introduced to eliminate the noisy and unnecessary instances which over-sampling may have introduced. After exploring over-sampling and under-sampling combinations, sample weights were introduced, by generating sample weights respecting to the method used. An example of this combination is given in the Figure 18 below.

```
twf_weights = [i for i in np.round(np.linspace(0.85, 0.95, 4), 2)]
all_weights = [i for i in np.round(np.linspace(18, 21, 4), 2)]
safe_weights = [i for i in np.round(np.linspace(3.6, 3.9, 3), 2)]
k_neighbors_list = [3, 4, 5, 6, 7, 8]

smote_values = [i for i in np.linspace(300, 2000, 3, dtype = int)]
smote_strategies = [{'TWF': r1, 'OSF': r2, 'PWF': r3, 'HDF': r4}
                    for r1 in smote_values
                    for r2 in smote_values
                    for r3 in smote_values
                    for r4 in smote_values]

all_params = []
for fold,(ti,vi) in enumerate(outer_cv.split(x,y),1):
    x_train, y_train, x_test, y_test = ft.split_data_cv(x,y,ti,vi)
    for ss in smote_strategies:
      for k in k_neighbors_list:
        for twf_w in twf_weights:
            for aw_w in all_weights:
                for safe_w in safe_weights:
                    all_params.append((fold, x_train, y_train, x_test, y_test, twf_w, aw_w, safe_w, ss, k))

n_jobs_outer = 5

results = Parallel(n_jobs = n_jobs_outer, verbose = 10)(delayed(run_experiment)(p) for p in all_params)
```

Figure 19: Sample Search of Sample Weights and Oversampling Parameters

With false positives of “TWF” in focus, anomaly detection method was applied. In highly imbalanced datasets, classification may be particularly hard since the algorithm may not be able to learn the underrepresented instances and sometimes even increase the false positives whether oversampling method is included or not. Anomaly detection, when used to identify the minority class as the anomaly, may provide valuable information. In this paper’s case, anomaly detection method was used to decrease the false positive which inevitably increased after the introduction of oversampling [19]. PCA, principal component analysis, rotates the axis of the features and by doing this the outlying points stand out, having large reconstruction error. As a result, those instances which were the FP predictions of the algorithm were removed. The corresponding code block for PCA is shown in Figure 20 and implementation in Figure 21.

```
def anomaly_detect_pca_xgb(x_train, x_test, n_c, th, w):
    #Preparing data
    binary = ['Type_H', 'Type_M', 'Type_L']
    num = [col for col in x_train.columns if col not in binary]
    preprocessor = ColumnTransformer(transformers = [('num', StandardScaler(), num), ('bin', 'passthrough', binary)])
    scaler = preprocessor.fit(x_train)
    x_train_scaled = scaler.transform(x_train)
    x_test_scaled = scaler.transform(x_test)

    #Actual Steps of PCA
    pca = PCA(n_components = n_c, svd_solver = 'full', whiten = w)
    x_train_pca = pca.fit_transform(x_train_scaled)
    x_recon = pca.inverse_transform(x_train_pca)
    reconstruction_error = ((x_train_scaled - x_recon)**2).mean(axis = 1)
    threshold = np.percentile(reconstruction_error, th)
    anomalies = reconstruction_error > threshold

    x_test_pca = pca.transform(x_test_scaled)
    x_test_recon = pca.inverse_transform(x_test_pca)
    reconstruction_error_test = ((x_test_scaled - x_test_recon)**2).mean(axis=1)
    anomalies_test = reconstruction_error_test > threshold
    return anomalies_test
```

Figure 20: PCA Anomaly Detection Function

3.5. XGBoost

Having explored decision tree and random forest, it is time to explore and see if XGBoost will provide benefits to the classification performance of this highly imbalanced dataset. XGBoost was selected because it allows to manage bias variance trade off using its regularization parameters. Random forest showed promising balancing to false positive and true positive balance. However, it wasn’t very efficient on the false positives of TWF. By using XGBoost, it is aimed to reduce FP of TWF while capturing a good performance throughout the dataset. XGBoost is a gradient-boosting method, meaning it fits decision trees sequentially, each tree learning from the previous tree. With parameters, the algorithm’s learning pace, weight of each individual leaf and the threshold of desired error reduction can be changed. This approach is what makes XGBoost a promising candidate, the probability of

identifying complex patterns in the data without increasing false positives. The methodology used was similar to the ones used in decision tree and random forest. First, the default algorithm was used to see the baseline performance. Then, main parameters such as “max_depth”, “learning_rate”, “min_child_weight”, “gamma” were explored using Bayesian search. After that the regularization tools were introduced which are “reg_alpha” and “reg_lambda”. Then, similar to previous sections, oversampling/undersampling and sample weights were explored, finally concluding with PCA anomaly detection for reducing false positives. As an inclusive search grid using Bayesian optimization, the related code is shown in the Figure 22 below.

```
### Bayesian Search Parameters ###
param_space = {
    'n_estimators': Integer(1000, 1200),
    'learning_rate': Real(0.1, 0.26, prior = 'log-uniform'),
    'max_depth': Integer(4, 6),
    'gamma': Real(0.15, 0.35, prior = 'log-uniform'),
    'min_child_weight': Real(2, 5),
    'reg_alpha': Real(0, 2),
    'reg_lambda': Real(0.01, 10, prior = 'log-uniform'),
    # class weights #
    'twf_w': Real(1.0, 10.0),
    'osf_w': Real(2.5, 5.0),
    'pwf_w': Real(2.5, 5.0),
    'hdf_w': Real(2.5, 5.0),
    'safe_w': Real(0.2, 3.0)
}
```

Figure 21: Search Parameter for XGBoost

After getting the initial parameter values from Bayesian search, more detailed grid search was done in order to ensure using parameters from a tighter range, resulting in more precision. For illustration, the exhaustive search over parameters was done and executed using the code in Figure 23.

```
### XGBoost Parameters ###
learning_rate = np.logspace(np.log10(0.125), np.log10(0.23), 3)
max_depth =  [i for i in np.linspace(4, 6, 3).astype(int)]
min_child_weight = np.linspace(4.0, 5.0, 2)
gamma = np.logspace(np.log10(0.25), np.log10(0.35), 3)
reg_alpha = np.linspace(0.6, 1.65, 3)

all_params = []
class_map = {'Safe': 0, 'HDF': 1, 'OSF': 2, 'PWF': 3, 'TWF': 4}
y = y.map(class_map)

### Fold for sample weights for each class ###
for fold, (train_idx, test_idx) in enumerate(outer_cv.split(x, y), 1):
    x_train, y_train, x_test, y_test = ft.split_data_cv(x, y, train_idx, test_idx)
    for lr in learning_rate:
        for md in max_depth:
            for mcw in min_child_weight:
                for g in gamma:
                    for twf_w in twf_weights:
                        for osf_w in osf_weights:
                            for pwf_w in pwf_weight:
                                for hdf_w in hdf_weight:
                                    for safe_w in safe_weights:
                                        all_params.append((fold, x_train, y_train, x_test, y_test, lr, md, mcw, g, twf_w, osf_w, pwf_w, hdf_w, safe_w))

n_jobs_outer = 4
results = Parallel(n_jobs = n_jobs_outer, verbose = 10)(delayed(run_experiment_xgb)(p) for p in all_params)
df_results = pd.DataFrame(results)
```

Figure 22: Grid Search of XGBoost

The definition of the function used is shown in Figure 24 below.

```
## outer 5-fold split ##
outer_cv = StratifiedKFold(n_splits = 5, shuffle = True, random_state = 42)
def run_experiment_xgb(params):

    fold, x_train, y_train, x_test, y_test, lr, md, mcw, g, twf_w, osf_w, pwf_w, hdf_w, safe_w= params # unpack params

    xgb = XGBClassifier(objective = 'multi:softprob', n_estimators = 3000, max_depth = md, learning_rate = lr, min_child_weight = mcw, gamma = g, random_state=42, n_jobs = 1)

    sw = ft.sample_weights_gen_xgb_combined(x_train, y_train, twf_w,  osf_w, pwf_w, hdf_w, safe_w)

    xgb.fit(x_train, y_train, sample_weight = sw)

    # predict
    y_predict = xgb.predict(x_test)
    f1_macro = f1_score(y_test, y_predict, average='macro', zero_division=0)

    return {
      'fold': fold,
      'max_depth': md, 'learning_rate': lr, 'min_child_weight': mcw, 'gamma': g,
       'twf_weight': twf_w, 'osf_weight': osf_w, 'pwf_weight': pwf_w, 'hdf_w': hdf_w, 'safe_weight': safe_w,
       'f1_macro': f1_macro
    }
```

Figure 23: Grid Search Function of XGBoost

Last but not least, when the search grid became large, different ways of plotting were applied. For Random Forest and XGBoost, due to higher numbers of parameters than Decision Tree, pyplot was used to map every combination of parameters with F1 macro score. Figure 25 is shown as an example of how this plot looks like. In this specific example, F1 macro values bigger than or equal to 0.805 are selected at the right-hand side of the plot.

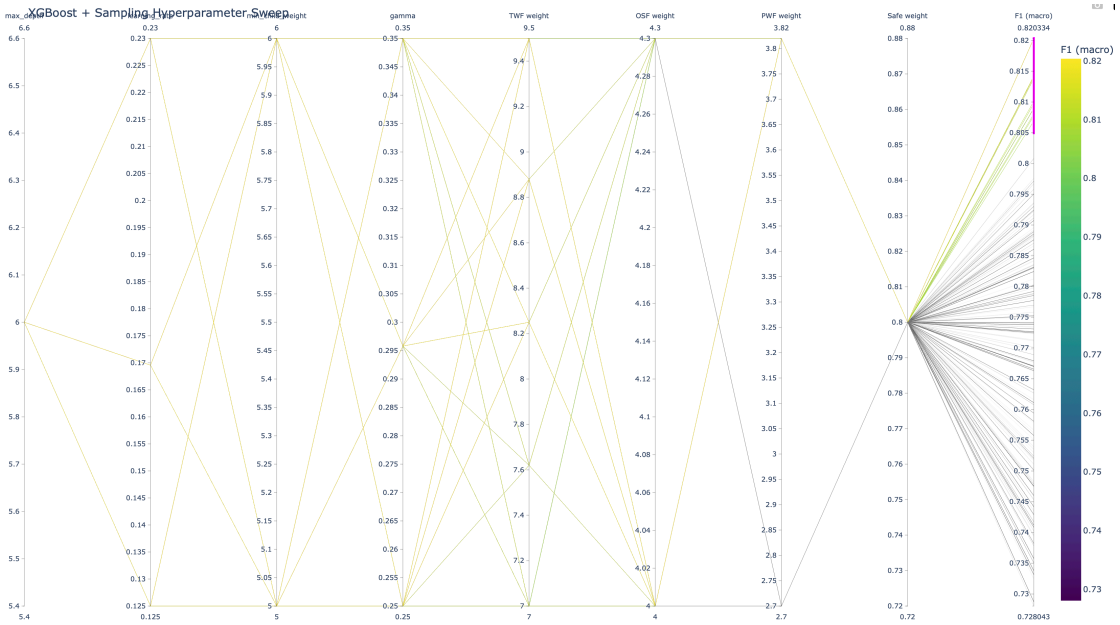

Figure 24: Combination of Parameters and F1 Score

4. Results and Discussion

The results of the progression of the algorithms are presented in this section, including performance of each machine learning techniques – Decision Trees, Random Forest, and XGBoost models. Firstly, the default settings of each model will be analysed by their confusion matrices and macro-averaged F1 scores. In every step of the explorations, the imbalanced nature of the data, false positives and difficulty of capturing 'TWF' incidents were the decisive facts. Also, the results of Decision Tree were shared relatively more since tree-based models are in discussion and Decision Tree is the foundation of the rest, giving leaner insights. Cross model comparisons of the results may offer insight into trade-offs between the models and the possible strengths and weaknesses of the models. In the tables below, each of the model specifications is displayed.

Table 7: Per Class Scores - Default Decision Tree

Class	Precision	Recall	F1 - Score	Support
HDF	0,927	0,878	0,902	115
OSF	0,989	0,978	0,984	92
PWF	0,963	0,975	0,969	80
TWF	0	0	0	42
Safe	0,994	0,999	0,997	9670
Accuracy			0,993	9999
Macro Avg.	0,775	0,766	0,770	9999
Weighted Avg.	0,989	0,993	0,991	9999

Table 8: Confusion Matrix - Default Decision Tree

	Pred: Safe	Pred: HDF	Pred: OSF	Pred: PWF	Pred: TWF
Actual: Safe	9663	7	0	0	0
Actual: HDF	10	101	1	3	0
Actual: OSF	1	1	90	0	0
Actual: PWF	2	0	0	75	0
Actual: TWF	42	0	0	0	0

Table 9: Per Class Scores - Default Random Forest

Class	Precision	Recall	F1 - Score	Support
HDF	0,980	0,852	0,902	115
OSF	0,947	0,978	0,984	92
PWF	0,987	0,975	0,969	80
TWF	0	0	0	42
Safe	0,994	1,000	0,997	9670
Accuracy			0,993	9999
Macro Avg.	0,782	0,761	0,770	9999
Weighted Avg.	0,989	0,993	0,991	9999

Table 10: Confusion Matrix - Default Random Forest

	Pred: Safe	Pred: HDF	Pred: OSF	Pred: PWF	Pred: TWF
Actual: Safe	9667	2	0	0	1
Actual: HDF	12	98	4	1	0
Actual: OSF	2	0	90	0	0
Actual: PWF	1	0	1	78	0
Actual: TWF	42	0	0	0	0

Table 11: Per Class Scores - Default XGboost

Class	Precision	Recall	F1 - Score	Support
HDF	0,964	0,939	0,952	115
OSF	0,967	0,957	0,962	92
PWF	0,844	0,812	0,828	80
TWF	0,167	0,024	0,042	42
Safe	0,993	0,998	0,995	9670
Accuracy			0,991	9999
Macro Avg.	0,787	0,746	0,756	9999
Weighted Avg.	0,988	0,991	0,989	9999

Table 12: Confusion Matrix - Default XGBoost

	Pred: Safe	Pred: HDF	Pred: OSF	Pred: PWF	Pred: TWF
Actual: Safe	9647	4	3	12	4
Actual: HDF	6	108	0	0	1
Actual: OSF	4	0	88	0	0
Actual: PWF	15	0	0	65	0
Actual: TWF	41	0	0	0	1

The macro F1 score of default DT, with ± standard deviation, of 5-fold CV is 0,770 ± 0,07. When looked at default RF, the macro F1 score o is 0,770 ± 0,08. Thirdly, the macro F1 score of default XGBoost is 0,754 ± 0,016. The “Safe” class is classified with a minimum F1-score of 0,995, which is a strong sign that even with default configuration, these models can capture the pattern in the data well. All three results have one significant weakness, which is the misclassification of the class “TWF” and nearly all of them are classified as “Safe”. The

default Random Forest performed very similar to Decision Tree even though it is an ensemble method. This suggests that misclassification of the minority classes remain a challenge without tuning techniques. Default XGBoost algorithm performed weaker, underlying tuning or oversampling. Default XGBoost algorithm consists of weak trees in its training, and this result shows that shallow trees may not be very effective for this dataset. Recall equation 2-15,

$$\widehat{y_i^{(t)}} = \widehat{y_i^{(t-1)}} + \eta f_t(x_i)$$

Here, label noise exposed the vulnerability of gradient boosting, where the variance is increased due to the trees keep trying to fit to the inflated residuals caused by class imbalance and noisy label like “TWF”.

All the models struggled when dealing with boundary – instances. The performances are promising, but the high numbers of FP and inability to correctly classify the class “TWF” are poised to be the major challenge regarding this dataset. Next, the progression of DT is observed. This involves exploring the configurations of class weights, hyper-parameter tuning and oversampling methods. First, oversampling was tested aiming feeding more data into the algorithm would improve its classification performance. In the Table 13, the classification report, per class, is presented. The settings for oversampling were found to be 866 sampling for “TWF, 750 for “OSF”, 200 for “PWF” and 1000 for “HDF”.

Table 13: Per Class Scores - DT with Oversampling

Class	Precision	Recall	F1 - Score	Support
HDF	0,917	0,870	0,893	115
OSF	0,918	0,978	0,947	92
PWF	0,974	0,950	0,962	80
TWF	0,082	0,143	0,104	42
Safe	0,995	0,992	0,994	9670
Accuracy			0,987	9999
Macro Avg.	0,778	0,787	0,780	9999
Weighted Avg.	0,990	0,987	0,988	9999

With F1 – macro score being 0,781 ± 0,008, the results improved slightly from the default settings. However, this due to correctly classified 6 instances of the class “TWF”, and in the expense of increasing the FP predictions of this class. This suggests that creating more

synthetic data to the already challenging boundary level instances of the data, the model's ability to classify correctly decreased. This result suggests that the model learned the weak, noisy spots of the dataset more and therefore failed to perform well on the test data. This is seen more evident when looked at the confusion matrix.

Table 14: Confusion Matrix - DT with Oversampling

	Pred: Safe	Pred: HDF	Pred: OSF	Pred: PWF	Pred: TWF
Actual: Safe	9596	8	1	2	63
Actual: HDF	6	100	7	0	2
Actual: OSF	1	0	90	0	1
Actual: PWF	3	0	0	76	1
Actual: TWF	35	1	0	0	6

Table 14 clearly illustrates the vast increase in FP of "TWF" and inability to improve the performance of other minority classes. After this, this scenario with class weight was tested. However, class weights did not make any difference. The dataset, especially "TWF" and some of the extreme instances of the other minority classes proved to be challenging and are prone to overfit when the already hard-to-detect instances increase in number. Then, it was determined that class weights without oversampling, with post pruning, yielded better results. Only the confusion matrix will be shown because it is sufficient for understanding the response of the model. Table 15 below is the result of applying pruning to the class weight setup above.

Table 15: Confusion Matrix - DT with Class Weights and Post Pruning

	Pred: Safe	Pred: HDF	Pred: OSF	Pred: PWF	Pred: TWF
Actual: Safe	9664	3	0	0	3
Actual: HDF	9	103	1	2	0
Actual: OSF	3	1	88	0	0
Actual: PWF	0	1	0	79	0
Actual: TWF	42	0	0	0	0

The macro F1 score was 0,775 ± 0,005. As a result, with slight decrease in classification performances the FP count is reduced to 6 in total. The FN count is increased due to the removal of all correct classification of "TWF", but this may be an acceptable outcome in

order to reduce FP count. Now the progression of Random Forest will be analysed. Recall that the macro F1 score of Random Forest was 0,770 ± 0,08. First, sample weights were explored. The confusion matrix is seen in Table 17.

Table 16: Confusion Matrix - RF with Sample Weights

	Pred: Safe	Pred: HDF	Pred: OSF	Pred: PWF	Pred: TWF
Actual: Safe	9630	12	1	2	25
Actual: HDF	1	114	0	0	0
Actual: OSF	0	0	92	0	0
Actual: PWF	0	0	1	79	0
Actual: TWF	38	0	0	0	4

The macro-F1 score was recorded as 0,809 ± 0,032. Adding sample weights to an ensemble method has eliminated the over-fitting situation of the similar settings of that in a Decision Tree. The macro F1 score has increased, and it is noticeable that the recall has increased throughout all classes except for “TWF”. FP were clustered only on “TWF” and “HDF” . The classification performance of “HDF” was increased with a recall of 0,991, meaning among the true “HDF” cases, nearly all were correctly classified by the model. Building on this progression of RF, oversampling and undersampling methods were explored. However, similar to the case in DT, oversampling made the model perform worse because of creating synthetic instances of already imbalanced, noisy and random-natured “TWF” samples. In fact, it increased FP count of “TWF” by 38, without improving TP count, which is not acceptable. Also, the macro F1 score dropped to 0,781 ± 0,019. When undersampling method was applied, the results got worse, due to the same problem that caused oversampling not performing better.

In short, having 25 FP count of “TWF” and 38 of the “TWF” being labelled as “Safe” became the main issue, hence PCA method was applied on both “HDF” and “TWF”, separately, whether it could reduce this risks that could cause the model’s reliability to be questioned. In the Table 18 and 19 results can be seen.

Table 17: Per Class Scores - RF with Sample Weights and PCA

Class	Precision	Recall	F1 - Score	Support
HDF	0,933	0,965	0,949	115
OSF	0,979	1,000	0,989	92
PWF	0,975	0,988	0,981	80
TWF	0,667	0,095	0,167	42
Safe	0,996	0,999	0,997	9670
Accuracy			0,994	
Macro Avg.	0,941	0,816	0,817	9999
Weighted Avg.	0,994	0,995	0,994	9999

Table 18: Confusion Matrix - RF with Sample Weights and PCA

	Pred: Safe	Pred: HDF	Pred: OSF	Pred: PWF	Pred: TWF
Actual: Safe	9657	8	1	2	2
Actual: HDF	4	111	0	0	0
Actual: OSF	0	0	92	0	0
Actual: PWF	0	0	1	79	0
Actual: TWF	38	0	0	0	4

PCA method successfully reduced FP count of “TWF” dramatically to 2, while FP count of “HDF” was reduced by nearly half. After the exploration of DT and RF, XGBoost’s progression will be shown. Recall default XGBoost yielded a macro F1 score of 0,754 ± 0,016. After exploring the main parameters such as number of estimators, learning rate, gamma, max depth and min child weight, the optimal macro F1 score regarding these parameters were nearly the same with the default settings, with just 0.004 increase. This result meant that XGBoost algorithm should be supplemented with factors that alter the decisions in the node split level. With sample weights implemented the results became better, with a macro F1 score of 0,788 ± 0,026. The per class scores and the confusion matrix is displayed in Table 20.

Table 19: Per Class Scores - XGBoost with Sample Weights

Class	Precision	Recall	F1 - Score	Support
HDF	0,956	0,939	0,947	115
OSF	0,989	0,978	0,984	92
PWF	0,949	0,938	0,943	80
TWF	0,091	0,071	0,080	42
Safe	0,995	0,996	0,995	9670
Accuracy			0,991	
Macro Avg.	0,796	0,784	0,790	9999
Weighted Avg.	0,990	0,991	0,990	9999

Next, as done previously in other methods, oversampling method was tested aiming to increase the performance of "TWF". However the results were worse, which is a consistent outcome considering the oversampling trials on the previous cases. One of the main benefits of XGBoost classifier is the ability of explicitly using regularisation techniques. Considering the gain equation of the algorithm, alpha and lambda values were explored in order to increase the classification performance by reducing overfitting. The results were not very satisfactory. The macro F1 score didn't change, in fact FP count of "TWF" increased by 7. The slight changes of FP/FN underline the fact that the algorithm with its available parameters is right at the centre of bias-variance trade-off. However, combined with PCA application, the regularisations applied yielded slightly better results. This is due to the model's already overfitting vulnerability was crippled by PCA anomaly detection.

Table 20: Per Class Scores - XGBoost with Sample Weights, Regularisation and PCA

Class	Precision	Recall	F1 - Score	Support
HDF	0,982	0,939	0,960	115
OSF	0,989	0,978	0,984	92
PWF	0,949	0,938	0,943	80
TWF	0,429	0,071	0,122	42
Safe	0,995	0,999	0,997	9670
Accuracy			0,994	
Macro Avg.	0,869	0,785	0,801	9999
Weighted Avg.	0,992	0,994	0,992	9999

Table 21: Confusion Matrix - XGBoost with Sample Weights, Regularisation and PCA

	Pred: Safe	Pred: HDF	Pred: OSF	Pred: PWF	Pred: TWF
Actual: Safe	9659	2	1	4	4
Actual: HDF	7	108	0	0	0
Actual: OSF	2	0	90	0	0
Actual: PWF	5	0	0	75	0
Actual: TWF	39	0	0	0	3

The macro F1 score was concluded as 0,798 ± 0,034. This is very similar to the result of non – PCA version however the FP count is significantly lower, especially for "TWF". Table 23 combines and summarizes the results across all models.

Table 22: Model Comparison

Model & Configuration	Macro F1 ± Std
DT - Default	0,770 ± 0,07
DT – Class Weights	0,800 ± 0,018
DT – Class Weights & Pruning	0,775 ± 0,005
RF – Default	0,770 ± 0,008
RF – Sample Weights	0,809 ± 0,032
RF – Sample Weights & PCA	0,819 ± 0,035
XGBoost - Default	0,754 ± 0,016
XGBoost – Sample Weights	0,788 ± 0,026
XGBoost – Sample Weights & Regularization & PCA	0,798 ± 0,034

The performance of boosting, as mentioned before, was crippled due to the noise in the dataset. The persistent residuals caused by random nature of "TWF" disabled the convergence of the algorithm. In other words, XGBoost kept on splitting the noise since the gradient didn't converge to zero in these instances. Meanwhile bagging averaged these noisy splits and decreased the correlation of such inner trees, making sure they didn't dominate the majority vote. Oversampling/undersampling methods did not improve the performances because duplicating low information data or deleting various rows did not balance the noise. Hence, ensemble method was superior amongst the tree-based algorithms.

Up to this point, the key performance indicator of each algorithm is displayed. It would be clearer to see at which instances the algorithm made unsuccessful classifications. To achieve this goal, following tables will present the original features, along with predicted and actual labels. The tables will have a column to specify whether the output was FP or FN. To avoid excessive information, that is if there are many FP cases, a part of the rows will be shown. Likewise, FN counts vastly belonged to "TWF", that's why some of the "TWF" instances will be omitted. To make healthy judgements, it is necessary to recall the conditions of fault occurrence.

- TWF: Failure is randomly assigned at tool wear time between 200-240 mins.
- HDF: Failure occurs when the difference of air – process temperature is below 8,6 K and rotational speed is below 1380 rpm.
- PWF: Failure occurs when the product of torque and rotational speed, power is below 3500 W or above 900 W.
- OSF: Failure occurs when the product of tool wear and torque is above 11.000 minNm for L, 12.000 for M and 13.000 for H. L, M and H are product quality variants.

Table 23: Examples of Misclassified Instances - RF

Error Type	Temp. Delta	Rot. Speed	Torque	Tool Wear	Predicted Label	Actual Label
FP	8,6	1317	59,4	179	HDF	Safe
FP	8,6	1356	48,3	36	HDF	Safe
FP	8,6	1358	53,1	194	HDF	Safe
FP	9,8	1382	58,5	199	OSF	Safe
FP	9,5	1326	64,8	71	PWF	Safe
FP	12	1668	27,9	240	TWF	Safe
FN	10,1	1455	41,3	208	Safe	TWF
FN	9,5	1604	36,1	225	Safe	TWF
FN	9,6	1867	23,4	225	Safe	TWF
FN	9	1567	39	214	Safe	TWF
FN	9,1	1530	37,3	207	Safe	TWF

Considering the FP instances and starting from “HDF”, Temperature difference is at the border level and Rotational Speed is just below the failure mark. This is a case of underfitting, and it is caused by the model’s struggle around border levels. Likewise, “OSF” misclassification could be resolved with better feature engineering and sample weight configuration. The “PWF” case is a good example of an overfitting, since the failure band criteria was not met. On the other hand, “TWF” cases are notably misclassified due to the dataset ambiguity. The anomaly detection step erased the FP of “TWF” but the model’s training was damaged by the instances that were in failure bound, although labelled as “Safe”. Only “TWF” label falls into this category of FN and the reasoning that was established previously remains the same, the main challenge is dataset ambiguity. This proposition can be fortified with counter examples from FP instances of RF and DT misclassifications. There is a room for further fine tuning for the regular failure criteria, but “TWF” instances are a challenge since the algorithm training is being complicated by random “Safe” and “TWF” labels.

It can be confidently said that default versions of the tree-based algorithms show promise and yield good overall performances. Across all the models, the majority “Safe” class was consistently classified successfully with the minimum F1 score of 0,995. However, parameter alterations were necessary to overcome underfitting that caused high FN predictions. The implementation of parameter tweaks proved beneficial and lead to some over-fitting issues. Oversampling on the other hand, although being a highly used technique in cases which deal with imbalanced dataset, consistently decreased the performance of the model by introducing more randomness and harder to capture data instances. Eventually, with theoretically optimal parameter combinations, PCA-anomaly detection method proved to be the most rewarding technique to reduce overfitting by eliminating FP instances for the class “TWF” and “HDF”. That being said, classification of “TWF” remained a major challenge throughout all model progressions. This is mainly due to some “TWF” instances being completely random and some being inconsistent in their occurrence of a failure. Hence, reduction in FP count of “TWF” was in focus since for every loss of 1 TP, in average 10 to 15 reduction was made to FP count. The models could yield better results if more powerful set of features were available suggesting failures only due to tool wear failure at a certain level tool wear within some variance, or a timestamped data with relevant data. Overall, after the configuring the selected model parameters, application of post-pruning or PCA to reduce overfitting yielded

good results considering classes “HDF”, “OSF”, “PWF” and “Safe”. For the poor classification of “TWF”, the possible reasons have been explained.

The main goal of predictive maintenance is to establish a more efficient production process, eliminating inefficiencies and costs of unplanned stoppages or planned but unnecessary maintenance. Having a low error system by leveraging powerful tools of AI is deterministic on how much positive impact can be get from adopting PdM. It was and will be important to focus on reducing false positive outputs of the algorithm, apart from the main goal of correctly classifying the actual failure criteria. All these factors come into play to save costs and time. The system may raise an early warning when the sensor values reach failure criteria or self-stops for human supervision. To solidify these findings and claims, an exemplary deduction can be made. Based on a study made, typical manufacturing operations lose approximately 200 hours annually per machine due to unplanned and unnecessary planned maintenance [23]. In that study, by implementing PdM, it was reported a 62.38% reduction of stoppages. However, to be conservative and to approach this issue as a benchmark or illustrative reference, 30% reduction of stoppages is assumed.

- Operating Costs of a Single CNC Machine (including labour, general operation and machine value depreciation): 80 USD/Hour
- Operation Pause: 200 Hours Annualy
- Reduction via PdM: 30% ~ 60 Hours
- Cost Saving Due to Reduction of Stoppage: 4800 USD
- Exchange Rate: Approximately 40 TL/USD
- Cost Saving Due to Reduction of Stoppage: 192.000 TL

As the efficiency of the PdM process increases, the reduction of stoppages in the production will decrease and each maintenance operation will be more accurate and necessary. Hence, reducing each false positive alerts of the model increases the reliability of the model and return as cost-saving to the business owners.

Another important way of utilizing PdM is to associate the process to tool life and measure the remaining useful life (RUL). Eventually, tools need to be replaced nonetheless but replacing them early or witnessing a failure in tools is an unwanted situation which causes unexpected pauses in production which costs money and time. This thesis uses a dataset that

has no timestamp or relevant features to suggest RUL. However, in literature there are works to predict RUL and guide timely tool replacement in return. One example to this is accurately predicting tool life based on spindle power signals using neural network techniques. Utilizing this approach minimizes disruptions due to unnecessary or failure-caused replacements of tools [24].

5. Conclusion

In this thesis, the effectiveness and capability of decision-tree based algorithms which are Decision Tree, Random Forest and XGBoost were studied. The diagnostics using AI on CNC machines, or any relevant settings are the subjects of predictive maintenance. It is stated that PdM has advantages over reactive and preventive maintenance. The focus in this thesis was on the AI algorithm that is an essential part of a PdM setup. Tree-based algorithms, in general, are more explainable and more interpretable than "black-box" algorithms like deep learning, neural network techniques. Initially, default settings of each algorithm were presented as benchmarks. Then, the progression of applying various techniques such as class/sample weights, oversampling, regularisation, and PCA anomaly detection were displayed. Performance summaries were given, per class, in terms of precision, recall and f1-score. Also, confusion matrix including all the classes were provided. While analysing various techniques and their corresponding results, threshold selection and feature importance were monitored to maintain a steady ratio of true/false positive, and bias-variance trade off.

In all the methods, "Safe" was successfully classified with a minimum F1-score of 0,995 and precision of 0,993. This result was interpreted such that tree-based algorithms could be successful when provided with sufficient data, while keeping in mind the noise. "PWF", "OSF" and "HDF" were well classified as well, especially "OSF" with a recall of 1 while having only 1 FP count. Due to its randomness of occurrence and explained reasonings, "TWF" was the main challenging class and the main factor that decreased the overall classification score of each algorithm.

Ultimately, this study shows the capabilities of tree-based algorithms on a relatively small sized, synthetic data. Various techniques of tree-based algorithms were leveraged and together with dimensionality reduction for anomaly detection yielded a macro F1-score of $0{,}819 \pm 0{,}035$, with major three conventional failure types and no-failure type having F1-

score higher than 0,95. These findings conclude diagnostics with AI is very promising to use in manufacturing settings to improve efficiency and reduce costs. Neural networks and deep learning could be implemented with larger and different type of data in future studies, along with signal data from real settings.

6. References

1. Krupitzer, C., Wagenhals, T., Züfle, M., Lesch, V., Schäfer, D., Mozaffarin, A., Edinger, J., Becker, C., & Kounev, S. (2020). A survey on predictive maintenance for Industry 4.0. arXiv preprint arXiv:2002.08224. https://doi.org/10.48550/arXiv.2002.08224
2. Zhu, T., Ran, Y., Zhou, X., & Wen, Y. (2024). survey of predictive maintenance: Systems, purposes, and approaches. arXiv. https://arxiv.org/abs/1912.07383
3. Luo, W., Hu, T., Ye, Y., Zhang, C., & Wei, Y. (2020). A hybrid predictive maintenance approach for CNC machine tool driven by Digital Twin. Robotics and Computer-Integrated Manufacturing, 65, 101974. https://doi.org/10.1016/j.rcim.2020.101974
4. lsayed, A. E. (2012). Reliability engineering (2nd ed.). Wiley
5. Özgür-Ünlüakın, D., Türkali, B., Karacaörenli, A., & Aksezer, S. Ç. (2019). A DBN based reactive maintenance model for a complex system in thermal power plants. Reliability Engineering & System Safety, 190, 106505. https://doi.org/10.1016/j.ress.2019.106505
6. Raharjo, R., Wicaksono, S., & Budiwantoro, B. (2024). Planning and scheduling in automatic train signaling maintenance: A review. Jurnal Rekayasa Mesin, 15, 823–835. https://doi.org/10.21776/jrm.v15i2.1552
7. Breiman, L. (2001). Statistical modeling: The two cultures (with comments and a rejoinder by the author). Statistical Science, 16(3), 199–231. https://doi.org/10.1214/ss/1009213726
8. Arena, S., Florian, E., Sgarbossa, F., Sølvsberg, E., & Zennaro, I. (2024). A conceptual framework for machine learning algorithm selection for predictive maintenance. Engineering Applications of Artificial Intelligence, 133(D), 108340
9. Fisher, R. (1936). Iris [Dataset]. UCI Machine Learning Repository. https://doi.org/10.24432/C56C76.
10. Géron, A. (2019). Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow (3rd ed.). O'Reilly Media.
11. Breiman, L., Friedman, J. H., Olshen, R. A., & Stone, C. J. (1984). Classification and regression trees. Wadsworth International Group.
12. Sammut, C., & Webb, G. I. (Eds.). (2017). Encyclopedia of machine learning and data mining (2nd ed.). Springer. https://doi.org/10.1007/978-1-4899-7687-1

13. Patgiri, R., Hussain, S., & Nongmeikapam, A. (2020, February 17). Empirical study on airline delay analysis and prediction. arXiv. https://doi.org/10.48550/arXiv.2002.10254
14. Hastie, T., Tibshirani, R., & Friedman, J. (2009). The Elements of Statistical Learning: Data Mining, Inference, and Prediction (2nd ed.). Springer.
15. Tan, Jimin & Yang, Jianan & Wu, Sai & Chen, Gang & Zhao, Jake. (2021). A critical look at the current train/test split in machine learning. 10.48550/arXiv.2106.04525.
16. Zollanvari, A. (2023). Machine learning with Python: Theory and implementation. Springer. https://doi.org/10.1007/978-3-031-33342-2
17. Garnett, R. (2023). Bayesian Optimization. Cambridge: Cambridge University Press.
18. Raschka, S., Patterson, J., & Nolet, C. (2020). Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence. Information, 11(4), 193.
19. Awe, O. O. (2024, February). Computational strategies for handling imbalanced data in machine learning [Slide deck]. International Statistical Institute. https://isi-web.org/sites/default/files/2024-02/Handling-Data-Imbalance-in-Machine-Learning.pdf
20. Breiman, L. (2001). Random forests. Machine Learning, 45(1). https://doi.org/10.1023/A:1010933404324
21. Amjad, M., Ahmad, I., Ahmad, M., Wróblewski, P., Kamiński, P., & Amjad, U. (2022). Prediction of pile bearing capacity using XGBoost algorithm: Modeling and performance evaluation. Applied Sciences, 12(4), 2126. https://doi.org/10.3390/app12042126
22. Chen, T., & Guestrin, C. (2016, August). XGBoost: A scalable tree boosting system. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (pp. 785–794). ACM. https://doi.org/10.1145/2939672.2939785
23. Natanael, D., & Sutanto, H. (2022). Machine Learning Application Using Cost-Effective Components for Predictive Maintenance in Industry: A Tube Filling Machine Case Study. Journal of Manufacturing and Materials Processing, 6(5), 108. https://doi.org/10.3390/jmmp6050108
24. Drouillet, C., Karandikar, J., Nath, C., Journeaux, A.-C., El Mansori, M., & Kurfess, T. (2016). Tool life predictions in milling using spindle power with the neural network

technique. Journal of Manufacturing Science and Engineering, 138(6), 061005. https://doi.org/10.1115/1.4032622

7. Appendix

7.1. Necessary libraries

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import OneHotEncoder
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.calibration import CalibratedClassifierCV
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.metrics import classification_report, confusion_matrix, f1_score,
roc_auc_score
from sklearn.compose import ColumnTransformer
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from imblearn.over_sampling import SVMSMOTE
from imblearn.under_sampling import TomekLinks
from skopt import BayesSearchCV
from skopt.space import Integer, Categorical, Real
from xgboost import XGBClassifier
from collections import Counter
```

7.2. Data Preparation

```
def read_data():
    folder = '/Users/cemkarahan/Desktop/'
    df = pd.read_feather(folder + 'thesis_data.fth') # Reading and loading the data
    return df

def data_prep_feature(df):
     #### Data Prep & Feature Engineering ###
    # One Hot Encoding - Type #
    df = df[~((df['Tool wear [min]'] < 200) & (df['TWF'] ==1))]
    ohe = OneHotEncoder(sparse_output = False)
    encoded_features = ohe.fit_transform(df[['Type']])
    df_encoded = pd.DataFrame(encoded_features, columns =
ohe.get_feature_names_out(['Type']))
    df_encoded['Product ID'] = df['Product ID'].values
    df = df.merge(df_encoded, how = 'left', on = 'Product ID')
    ## Finalizing the features and responses ###
    df.drop(columns = {'UDI', 'Product ID'}, inplace = True) # # Dropping
unnecessary fields
```

```
    df['Dumy'] = df['TWF'] + df['HDF'] + df['PWF'] + df['OSF']
    df['Safe'] = np.where(df['Dumy'] > 0, 0, 0.5)
    df['Failure Type'] = df[['Safe', 'HDF', 'OSF', 'PWF', 'TWF']].idxmax(axis = 1)
# Gathering failure types into one column
    df['Failure Type - All'] = df[['Safe', 'HDF', 'OSF', 'PWF', 'RNF',
'TWF']].idxmax(axis = 1) # Gathering failure types into one column WITH RANDOM
FAILURE
    df['Temperature Delta'] = abs(df['Process temperature [K]'] - df['Air
temperature [K]'])
    df = df[['Type_H', 'Type_M', 'Type_L', 'Temperature Delta', 'Rotational speed
[rpm]', 'Torque [Nm]', 'Tool wear [min]', 'Failure Type', 'Safe', 'HDF', 'OSF',
'PWF', 'TWF']] #'RNF',
    return df

def parameter_creation(df):
    tool_multipliers = {'L': 3, 'M': 2, 'H': 1}
    types = ['L', 'M', 'H']
    for i in types: # Only for Overstrain Parameter
        df[f'Overstrain Parameter_{i}'] = df['Torque [Nm]'] * df['Tool wear [min]']
    for i in types: # Only for Power Parameter
        df[f'Power Parameter_{i}'] = df['Torque [Nm]'] * df['Rotational speed
[rpm]']
    tool_wear_failure_factor = df['Tool wear [min]'].between(200, 240).astype(int)
    for i in types: # Only for Tool Parameter
            df[f'Tool Parameter_{i}'] = df['Tool wear [min]'] * tool_multipliers[i]
* tool_wear_failure_factor * df[f'Type_{i}']
    return df
```

7.3. Parameter Search

7.3.1. Bayes Optimization Search

```
class WeightedDecisionTreeClassifier(DecisionTreeClassifier):
    def __init__(self,
                 criterion='entropy',
                 random_state=42,
                 # five per-class weight knobs:
                 twf_w=1.0, osf_w=1.0, pwf_w=1.0, hdf_w=1.0, safe_w=1.0,
                 class_weight = None,
                 **kwargs):
        super().__init__(criterion=criterion, random_state=random_state,
class_weight = class_weight, **kwargs)
        self.twf_w = twf_w
        self.osf_w = osf_w
        self.pwf_w = pwf_w
        self.hdf_w = hdf_w
        self.safe_w = safe_w

    def fit(self, X, y, **fit_kwargs):
```

```
        # build the class_weight dict from our five knobs
        cw = {
            'Safe': self.safe_w,
            'HDF': self.hdf_w,
            'OSF': self.osf_w,
            'PWF': self.pwf_w,
            'TWF': self.twf_w
        }
        # inject it into the real tree
        super().set_params(class_weight=cw)
        return super().fit(X, y, **fit_kwargs)
df = ft.read_data()
df = ft.data_prep_feature(df)
df = ft.parameter_creation(df)
x_fields = [col for col in df.columns if col not in ['Failure Type', 'Safe', 'HDF',
'OSF', 'PWF',  'TWF']] # 'RNF',
x = df[x_fields]
y = df['Failure Type']
x.rename(columns = {'Rotational speed [rpm]': 'Rotational speed', 'Torque [Nm]':
'Torque', 'Tool wear [min]': 'Tool wear'}, inplace = True)

## outer 5-fold split ##
outer_cv = StratifiedKFold(n_splits = 5, shuffle = True, random_state = 42)
inner_cv = StratifiedShuffleSplit(n_splits = 5, test_size = 0.25, random_state =
42)
def minority_auc_scorer(estimator, x, y_true):
    # Predict class probabilities
    y_prob = estimator.predict_proba(x)

    # Build sample weights: Safe=0.1, failures=1.0
    w = np.where(y_true == 'Safe', 0.1, 1.0)

    # Compute weighted AUC
    return roc_auc_score(
        y_true,
        y_prob,
        average="weighted",
        multi_class="ovr",
        sample_weight=w
    )
param_space = {
    'twf_w' : Real(1.0,  20.0),
    'osf_w' : Real(1.0,  20.0),
    'pwf_w' : Real(1.0,  20.0),
    'hdf_w' : Real(1.0,  20.0),
    'safe_w': Real(0.5,   5.0),
}

results = []
for fold, (train_idx, test_idx) in enumerate(outer_cv.split(x, y), 1):
    x_train, y_train, x_test, y_test = ft.split_data_cv(x, y, train_idx, test_idx)
```

```
    x_train = x_train.reindex(columns=x.columns, fill_value=0)
    x_test  = x_test .reindex(columns=x.columns, fill_value=0)

    # 3) run BayesSearchCV on a DecisionTreeClassifier
    bayes_cv = BayesSearchCV(WeightedDecisionTreeClassifier(random_state=42),
        search_spaces = param_space,
        scoring = minority_auc_scorer,   # custom AUC scorer
        cv = inner_cv,
        n_iter = 80,
        n_jobs = 6,
        refit = True,
        random_state = 42,
        verbose = 0
    )
    bayes_cv.fit(x_train, y_train)

    # 4) evaluate on the held-out fold
    best = bayes_cv.best_estimator_
    y_pred = best.predict(x_test)
    f1_macro = f1_score(y_test, y_pred, average='macro', zero_division=0)
    auc_score = bayes_cv.best_score_
    # 5) append exactly the same way you did before
    results.append({
        'fold':      fold,
        **bayes_cv.best_params_,
        'f1_macro':  f1_macro,
        'auc':       auc_score,
    })
df_results = pd.DataFrame(results)
print(df_results)
```

7.3.2. Grid Search

```
df = ft.read_data()
df = ft.data_prep_feature(df)
df = ft.parameter_creation(df)

### X & Y - Splitting Data ###
x_fields = [col for col in df.columns if col not in ['Failure Type', 'Safe', 'HDF',
'OSF', 'PWF',  'TWF']]
x = df[x_fields]
y = df['Failure Type']

## outer 5-fold split ##
outer_cv = StratifiedKFold(n_splits = 5, shuffle = True, random_state = 42)
def run_experiment(params):
    # unpack params
    fold, x_train, y_train, x_test, y_test, n_pca_twf, t_twf, n_pca_hdf, t_hdf =
params

    rf = RandomForestClassifier(n_estimators=200, random_state=42, n_jobs=1)
```

```
    cal = CalibratedClassifierCV(rf, cv=3, method='sigmoid', n_jobs=1)

    sw = ft.sample_weights_gen_combined(x_train, y_train, 8, 5, 9, 1, 0.8)

    anomalies_test_twf = ft.anomaly_detect_pca(x_train, x_test, n_pca_twf, t_twf)
    anomalies_test_hdf = ft.anomaly_detect_pca(x_train, x_test, n_pca_hdf, t_hdf)

    cal.fit(x_train, y_train, sample_weight = sw)

    # predict
    y_pred = cal.predict(x_test)
    y_pred[(anomalies_test_twf == False) & (y_pred == 'TWF')] = 'Safe'
    y_pred[(anomalies_test_hdf == False) & (y_pred == 'HDF')] = 'Safe'
    f1_macro = f1_score(y_test, y_pred, average='macro', zero_division=0)

    return {
      'fold': fold,
      'n_components_twf': n_pca_twf, 'threshold_twf': t_twf, 'n_components_hdf':
n_pca_hdf, 'threshold_hdf': t_hdf,
      'f1_macro': f1_macro
    }
n_components_list_twf = np.round(np.linspace(0.65, 0.85, 5), 2)
threshold_list_twf = np.round(np.linspace(98, 99, 4), 1)

n_components_list_hdf = np.round(np.linspace(0.5, 0.7, 5), 2)
threshold_list_hdf = np.round(np.linspace(60, 80, 5), 1)

all_params = []
for fold,(ti,vi) in enumerate(outer_cv.split(x,y),1):
    x_train, y_train, x_test, y_test = ft.split_data_cv(x,y,ti,vi)
    for n_pca_twf in n_components_list_twf:
        for t_twf in threshold_list_twf:
            for n_pca_hdf in n_components_list_hdf:
                for t_hdf in threshold_list_hdf:
                    all_params.append((fold, x_train, y_train, x_test, y_test,
n_pca_twf, t_twf, n_pca_hdf, t_hdf))

n_jobs_outer = 5

results = Parallel(n_jobs = n_jobs_outer, verbose = 10)(delayed(run_experiment)(p)
for p in all_params)


df_results = pd.DataFrame(results)
df_results.drop(columns = {'fold'}, inplace= True)
df_results_avg = df_results.groupby(['n_components_twf', 'threshold_twf',
'n_components_hdf', 'threshold_hdf'], as_index = False)['f1_macro'].mean()
best = df_results_avg.loc[df_results_avg['f1_macro'].idxmax()]
print("\nBest macro F1 setting:\n", best)
fig = px.parallel_coordinates(
    df_results_avg,
```

```
    dimensions=[
        'n_components_twf',
        'threshold_twf',
        'n_components_hdf',
        'threshold_hdf',
        'f1_macro',
    ],
    color='f1_macro',
    color_continuous_scale=px.colors.sequential.Viridis,
    labels={
        'n_components_twf':'n_components_twf',
        'threshold_twf':'threshold_twf',
        'n_components_hdf':'n_components_hdf',
        'threshold_hdf':'threshold_hdf',
        'f1_macro':'F1 (macro)',
    }
)
fig.update_layout(title="RF PCA Sweep")
fig.show()
```

7.3. Configuration of Algorithms

7.3.1. Decision Tree

```
def evaluate_dt_folds(x, y, dt_params = None, smote_params = None, do_oversampling
= False):
    f1_per_fold = []
    y_true_list = []
    y_pred_list = []
    labels = ['Safe', 'HDF', 'OSF', 'PWF', 'TWF']
    false_positives = {lbl: [] for lbl in labels}
    false_negatives = {lbl: [] for lbl in labels}
    dt_params = dt_params or {}
    smote_params = smote_params or {}
    outer_cv = StratifiedKFold(n_splits = 5, shuffle = True, random_state = 42)

    for fold, (train_idx,test_idx) in enumerate(outer_cv.split(x, y), 1):
        x_train, y_train, x_test, y_test = split_data_cv(x, y, train_idx, test_idx)
        dt = DecisionTreeClassifier(**dt_params)

        if do_oversampling:
            oversampling = SVMSMOTE(**smote_params)
            x_train, y_train = oversampling.fit_resample(x_train, y_train)
        dt.fit(x_train, y_train)
        y_pred = dt.predict(x_test)

        ##Tracking the misclassifications
        for l in ['HDF', 'OSF', 'PWF', 'TWF']:
            fp = (y_pred == l) & (y_test == 'Safe')
            fn = (y_pred == 'Safe') & (y_test== l)
            false_positives[l].extend(test_idx[fp])
            false_negatives[l].extend(test_idx[fn])
```

```
        f1 = f1_score(y_test, y_pred, average = 'macro', zero_division = 0)
        f1_per_fold.append(f1)
        y_true_list.append(y_test)
        y_pred_list.append(y_pred)

    y_true_all = np.concatenate(y_true_list)
    y_pred_all = np.concatenate(y_pred_list)
    print("Classification Report Test:\n", classification_report(y_true_all,
y_pred_all, zero_division = 0))

    test_cm = confusion_matrix(y_true_all, y_pred_all, labels = labels) # Confusion
matrix preperation
    h.print_multiple_class_confusion_matrix(test_cm) # Confusion matrix, layout
    mean_f1 = np.mean(f1_per_fold)
    std_f1 = np.std(f1_per_fold, ddof = 1)
    print(f'Macro-F1= {mean_f1:.3f} ± {std_f1:.3f} (5-fold CV)')
    print(dt.get_params())

        # Prepare full DataFrames for export
    records_fp = []
    records_fn = []

    for label in labels:
        for idx in false_positives[label]:
            row = x.iloc[idx].to_dict()
            row.update({'true_label': y.iloc[idx], 'pred_label': label})
            records_fp.append(row)
        for idx in false_negatives[label]:
            row = x.iloc[idx].to_dict()
            row.update({'true_label': label, 'pred_label':
y_pred_all[np.where(np.concatenate(y_true_list) ==
y_true_all)[0][list(false_negatives[label]).index(idx)]]})
            records_fn.append(row)

    df_fp_all = pd.DataFrame.from_records(records_fp)
    df_fn_all = pd.DataFrame.from_records(records_fn)
    print(df_fp_all)
    print(df_fn_all)
```

7.3.2. Random Forest

```
def evaluate_rf_folds(x, y, rf_params = None, cal_params = None, svm_params = None,
do_sample_weights = False, sample_weights = None, do_pca_anomaly = False,
pca_parameters_twf = None, pca_parameters_hdf = None, do_oversampling = False,
twf_oversample = None, do_undersampling = False):
    rf_params = rf_params or {}
    cal_params = cal_params or {}
    sample_weights = sample_weights or {}
```

```
    pca_parameters_twf = pca_parameters_twf or {}
    pca_parameters_hdf = pca_parameters_hdf or {}
    outer_cv = StratifiedKFold(n_splits = 5, shuffle = True, random_state = 42)

    f1_per_fold = []
    y_true_list = []
    y_pred_list = []
    labels = ['Safe', 'HDF', 'OSF', 'PWF', 'TWF']
    false_positives = {lbl: [] for lbl in labels}
    false_negatives = {lbl: [] for lbl in labels}

    for fold, (train_idx,test_idx) in enumerate(outer_cv.split(x, y), 1):
        ### splitting data ###
        x_train, y_train, x_test, y_test = split_data_cv(x, y, train_idx, test_idx)
        if do_oversampling:
            x_train, y_train = svm_smote(x_train, y_train, twf_oversample)
        if do_undersampling:
            undersampling = TomekLinks(sampling_strategy = ['Safe'])
            mask_remove = (y_train == 'TWF') | (y_train == 'Safe')
            x_twf, y_twf = x_train[mask_remove], y_train[mask_remove]
            x_train = x_train[~mask_remove]
            y_train = y_train[~mask_remove]
            x_twf, y_twf = undersampling.fit_resample(x_twf, y_twf)
            x_train = pd.concat([x_train, x_twf])
            y_train = pd.concat([pd.Series(y_train), pd.Series(y_twf)])
        if do_sample_weights:
            sw = sample_weights_gen_combined(x_train, y_train, **sample_weights)
        else:
            sample_weights = None

        if do_pca_anomaly:
            anomalies_test_twf = anomaly_detect_pca(x_train, x_test,
**pca_parameters_twf)
            anomalies_test_hdf = anomaly_detect_pca(x_train, x_test,
**pca_parameters_hdf)
        base_rf = RandomForestClassifier(**rf_params)
        calibrator = CalibratedClassifierCV(base_rf, **cal_params)
        calibrator.fit(x_train, y_train, sample_weight = sw) if do_sample_weights
else calibrator.fit(x_train, y_train)

        y_pred = calibrator.predict(x_test)
        if do_pca_anomaly:
            y_pred[(anomalies_test_twf == False) & (y_pred == 'TWF')] = 'Safe'
            y_pred[(anomalies_test_hdf == False) & (y_pred == 'HDF')] = 'Safe'
        y_true_list.append(y_test)
        y_pred_list.append(y_pred)
        f1 = f1_score(y_test, y_pred, average = 'macro', zero_division = 0)
        f1_per_fold.append(f1)

        ##Tracking the misclassifications
        for l in ['HDF', 'OSF', 'PWF', 'TWF']:
```

```
            fp = (y_pred == l) & (y_test == 'Safe')
            fn = (y_pred == 'Safe') & (y_test== l)
            false_positives[l].extend(test_idx[fp])
            false_negatives[l].extend(test_idx[fn])

    y_true_all = np.concatenate(y_true_list)
    y_pred_all = np.concatenate(y_pred_list)
    print("Classification Report Test:\n", classification_report(y_true_all,
y_pred_all, zero_division = 0))
    labels = ['Safe', 'HDF', 'OSF', 'PWF', 'TWF'] #'RNF',
    test_cm = confusion_matrix(y_true_all, y_pred_all, labels = labels) # Confusion
matrix preperation
    h.print_multiple_class_confusion_matrix(test_cm) # Confusion matrix, layout

    mean_f1 = np.mean(f1_per_fold)
    std_f1 = np.std(f1_per_fold, ddof = 1)
    print(f'Macro-F1= {mean_f1:.3f} ± {std_f1:.3f} (5-fold CV)')
    print(base_rf.get_params())
    if do_sample_weights:
        print(sample_weights)

    # Prepare full DataFrames for export
    records_fp = []
    records_fn = []

    ### EXPORT TO EXCEL ###
    for label in labels:
        for idx in false_positives[label]:
            row = x.iloc[idx].to_dict()
            row.update({'true_label': y.iloc[idx], 'pred_label': label})
            records_fp.append(row)
        for idx in false_negatives[label]:
            row = x.iloc[idx].to_dict()
            row.update({'true_label': label, 'pred_label':
y_pred_all[np.where(np.concatenate(y_true_list) ==
y_true_all)[0][list(false_negatives[label]).index(idx)]]})
            records_fn.append(row)

    df_fp_all = pd.DataFrame.from_records(records_fp)
    df_fn_all = pd.DataFrame.from_records(records_fn)
```

7.3.3. XGBoost

```
def evaluate_xgb_folds(x, y, xgb_params = None, oversampling_params = None,
do_sample_weights = False, do_oversampling = False, do_undersampling = False,
weights = None, do_PCA = False, pca_parameters_twf = None, pca_parameters_hdf =
None):
    f1_per_fold = []
    y_true_list = []
    y_pred_list = []
    # label setup
```

```
    class_map    = {0: 'Safe', 1: 'HDF', 2: 'OSF', 3: 'PWF', 4: 'TWF'}
    inv_map      = {v:k for k,v in class_map.items()}
    labels       = list(class_map.values())
    false_pos    = {lbl: [] for lbl in labels}
    false_neg    = {lbl: [] for lbl in labels}
    xgb_params = xgb_params or {}
    oversampling_params = oversampling_params or {}
    twf_w, osf_w, pwf_w, hdf_w, safe_w = weights if weights is not None else (None,
None, None, None, None)
    n_pca_twf, t_twf = pca_parameters_twf if pca_parameters_twf is not None else
(None, None, None)
    n_pca_hdf, t_hdf = pca_parameters_hdf if pca_parameters_hdf is not None else
(None, None, None)

    outer_cv = StratifiedKFold(n_splits = 5, shuffle = True, random_state = 42)

    for fold, (train_idx,test_idx) in enumerate(outer_cv.split(x, y), 1):
        x_train, y_train, x_test, y_test = split_data_cv(x, y, train_idx, test_idx)
        xgb = XGBClassifier(**xgb_params)

        if do_PCA:
            anomalies_test_twf = anomaly_detect_pca_xgb(x_train, x_test, n_c =
n_pca_twf, th = t_twf)
            anomalies_test_hdf = anomaly_detect_pca_xgb(x_train, x_test, n_c =
n_pca_hdf, th = t_hdf)

        if do_oversampling:
            oversampling = SVMSMOTE(**oversampling_params)
            x_train, y_train = oversampling.fit_resample(x_train, y_train)

        if do_undersampling:
            undersampling = TomekLinks(sampling_strategy = [0])
            mask_remove = (y_train == 4) | (y_train == 0)
            x_twf, y_twf = x_train[mask_remove], y_train[mask_remove]
            x_train = x_train[~mask_remove]
            y_train = y_train[~mask_remove]
            x_twf, y_twf = undersampling.fit_resample(x_twf, y_twf)
            x_train = pd.concat([x_train, x_twf])
            y_train = pd.concat([pd.Series(y_train), pd.Series(y_twf)])

        if do_sample_weights:
            sample_weights = sample_weights_gen_xgb_combined(x_train, y_train,
twf_weight = twf_w, osf_weight = osf_w, pwf_weight = pwf_w, hdf_weight = hdf_w,
safe_weight = safe_w)
        else:
            sample_weights = None

        xgb.fit(x_train, y_train, sample_weight = sample_weights)
        y_pred = xgb.predict(x_test)
        if do_PCA:
            y_pred[(anomalies_test_twf == False) & (y_pred == 4)] = 0
```

```
            y_pred[(anomalies_test_hdf == False) & (y_pred == 1)] = 0

        # track misclassifications
        safe_id = inv_map['Safe']
        for fault in ['HDF','OSF','PWF','TWF']:
            fid = inv_map[fault]
            fp = (y_pred==fid) & (y_test==safe_id)
            fn = (y_pred==safe_id) & (y_test==fid)
            false_pos[fault].extend(test_idx[fp])
            false_neg[fault].extend(test_idx[fn])

        f1 = f1_score(y_test, y_pred, average = 'macro', zero_division = 0)
        f1_per_fold.append(f1)
        y_true_list.append(y_test)
        y_pred_list.append(y_pred)


    y_true_all = np.concatenate(y_true_list)
    y_pred_all = np.concatenate(y_pred_list)
    targets = [class_map[i] for i in range(len(class_map))]
    print("Classification Report Test:\n", classification_report(y_true_all,
y_pred_all, target_names = targets, zero_division = 0))
    test_cm = confusion_matrix(y_true_all, y_pred_all) #, labels = labels) #
Confusion matrix preperation
    h.print_multiple_class_confusion_matrix(test_cm) # Confusion matrix, layout
    print(xgb.get_params())
    mean_f1 = np.mean(f1_per_fold)
    std_f1 = np.std(f1_per_fold, ddof = 1)
    print(f'Macro-F1= {mean_f1:.3f} ± {std_f1:.3f} (5-fold CV)')
    if do_sample_weights:
        print('Sample Weights (TWF, All, Safe): ',weights)
    if do_oversampling:
        print(oversampling.get_params())

    # build full DataFrames & map back to text labels
    records_fp = []
    records_fn = []

    for fault in ['HDF','OSF','PWF','TWF']:
        # false positives: true_label was Safe, pred_label is the fault
        for idx in false_pos[fault]:
            row = x.iloc[idx].to_dict()
            row.update({
                'true_label': 'Safe',
                'pred_label':   fault
            })
            records_fp.append(row)

        # false negatives: true_label was the fault, pred_label was Safe
        for idx in false_neg[fault]:
            row = x.iloc[idx].to_dict()
```

```
            row.update({
                'true_label': fault,
                'pred_label': 'Safe'
            })
            records_fn.append(row)

    df_fp_all = pd.DataFrame(records_fp)
    df_fn_all = pd.DataFrame(records_fn)
```

7.4. Execution of Algorithms

7.4.1. Decision Tree

```
### X & Y - Splitting Data ###
x_fields = [col for col in df.columns if col not in ['Failure Type', 'Safe', 'HDF',
'OSF', 'PWF',  'TWF']]
x = df[x_fields]
y = df['Failure Type']

sampling_strategy = {'TWF': 866, 'OSF': 750, 'PWF': 200, 'HDF': 1000}
cw = {'Safe': 0.8, 'HDF': 1, 'OSF': 7.5, 'PWF': 3.5, 'TWF': 1}

ft.evaluate_dt_folds(x, y, dt_params = {'criterion': 'gini', 'class_weight': cw,
'random_state': 42, 'ccp_alpha' : 0.0003},
                     smote_params = {'sampling_strategy': sampling_strategy,
'random_state': 42}, do_oversampling = False)
```

7.4.2. Random Forest

```
### X & Y - Splitting Data ###
x_fields = [col for col in df.columns if col not in ['Failure Type', 'Safe', 'HDF',
'OSF', 'PWF',  'TWF']]
x = df[x_fields]
y = df['Failure Type']
ft.evaluate_rf_folds(x, y, rf_params = {'criterion': 'gini', 'random_state': 42,
'n_jobs': 8}, cal_params= {'cv': 5},
                     do_sample_weights = True, sample_weights = {'twf_weight': 9.0,
'osf_weight': 3.0, 'pwf_weight': 7.0, 'hdf_weight': 9.0, 'safe_weight': 0.8},
                     do_pca_anomaly = True, pca_parameters_twf = {'n_pca': 0.8,
't': 99.0}, pca_parameters_hdf = {'n_pca': 0.7, 't': 71.0}, do_oversampling =
False, twf_oversample = 400, do_undersampling = False)
```

7.4.3. XGBoost

```
df = ft.read_data()
df = ft.data_prep_feature(df)
df = ft.parameter_creation(df)
```

```
### X & Y - Splitting Data ###
x_fields = [col for col in df.columns if col not in ['Failure Type', 'Safe', 'HDF',
'OSF', 'PWF',  'TWF']] # 'RNF',
x = df[x_fields]
y = df['Failure Type']
class_map = {'Safe': 0, 'HDF': 1, 'OSF': 2, 'PWF': 3, 'TWF': 4}
y = y.map(class_map)
x.rename(columns = {'Rotational speed [rpm]': 'Rotational speed', 'Torque [Nm]':
'Torque', 'Tool wear [min]': 'Tool wear'}, inplace = True)

ratio = 0.05
counts = Counter(y)
sampling_twf = int(counts[0] * ratio)
sampling_pwf = int(counts[0] * 0.2)
sampling_osf = int(counts[0] * 0.05)
sampling_hdf = int(counts[0] * 0.3)

ft.evaluate_xgb_folds(x, y,
                      xgb_params = {'random_state': 42, 'n_estimators': 1000,
'learning_rate': 0.3, 'min_child_weight': 3.0, 'max_depth': 6, 'gamma': 0.30,
'reg_alpha': 0.7, 'reg_lambda': 2.5},
                      oversampling_params = {'sampling_strategy': {4: 700},
'random_state': 42}, #, 3: sampling_pwf, 2: sampling_osf, 1: sampling_hdf}},
                        do_sample_weights = True,
                        do_oversampling = False,
                        do_undersampling = False,
                        weights = ft.weights(6.6, 8.0, 5.2, 1.0, 0.8),
                                          # twf, osf, pwf, hdf, safe
                        do_PCA = True,
                        pca_parameters_twf = (0.80, 98.5),
                        pca_parameters_hdf = (0.70, 67.0)
                        )
```