To what Extent is the Implementation of Tree-Based Models Effective and Feasible in Predictive Maintenance under Industry 4.0?

Martin Kairay Bernards

Dissertation written under the supervision of

Pedro Afonso Fernandes

Dissertation submitted in partial fulfilment of requirements for the MSc in Business Analytics, at the Universidade Católica Portuguesa, 01.04.2023.

Abstract

A tecnologia disponível na Indústria 4.0, combinada com as técnicas de aprendizagem de máquinas em constante evolução, permitiram aos gestores e intervenientes dos sectores de fabrico, transporte e logística prever a probabilidade das suas máquinas falharem com uma fiabilidade sem precedentes. Embora os modelos de aprendizagem de máquinas se tenham tornado altamente complexos nos últimos anos, os modelos baseados em árvores a partir de Árvores de Decisão para Random Forests e modelos de reforço, continuam a existir no meio académico devido à sua viabilidade, interpretação e mesmo eficácia. O objectivo deste estudo é captar a relevância de tais modelos no mundo real e se vale a pena investir neles. Para orientar o processo de construção de modelos baseados em árvores, será utilizado o Processo Padrão Cruzado de Mineração de Dados (CRISP-DM) para compreender o negócio, bem como os dados, preparar os dados, criar e avaliar os modelos, e construir uma estratégia de implantação de uma forma iterativa e flexível. Um conjunto de dados sintéticos que simulava os dados dos sensores de uma fresadora foi utilizado para a investigação, e os resultados através de múltiplas técnicas de avaliação, indicaram que o modelo de impulsionamento, XGBoost, superou os modelos de Random Forests, Árvore de Decisão, e Regressão Logística. Embora os modelos de outras pesquisas tenham superado o XGBoost, contudo, o XGBoost, juntamente com as Random Forests, são aconselhados a serem ainda tomados em consideração devido à sua viabilidade de serem produzidos, treinados e interpretados, gerando ao mesmo tempo bons resultados.

The technology available in Industry 4.0, combined with the constantly evolving machine learning techniques have allowed managers and stakeholders of the manufacturing, transportation, and logistics sectors to predict the likelihood of their machines failing at an unprecedented reliability. Although Machine learning models have become highly complex in recent years, tree-based models starting from Decision Trees to Random Forests and boosting models, continue to exist in academia due to their feasibility, interpretation and even effectiveness. The purpose of this study is to grasp the relevance of such models in the real-world and whether they are worth investing into. To guide the process of building tree-based models, the Cross-Industry Standard Process for Data-Mining (CRISP-DM) will be utilized in order to understand the business as well as the data, prepare the data, create and evaluate the models, and build a deployment strategy in an iterative and flexible manner. A synthetic dataset that simulated the sensor data of a milling machine was used for the research, and the results through multiple evaluation techniques, indicated that the boosting model, XGBoost, outperformed the Random Forest, Decision Tree, and Logistic Regression models. Though the models from other research outperformed XGBoost, however, XGBoost along with Random Forests are advised to still be taken under consideration due to their feasibility to be produced, trained, and interpreted, while still generating good results.

Keywords: Tree-Based Models, Industry 4.0, Predictive Maintenance, Decision Trees, Random Forest, Boosting, XGBoost

Contents

List of Figures

Acronyms

AI Artificial Intelligence. 1, 2, 4

ARIMA Autoregressive Integrated Moving Average. 6

AUC-ROC Area under the Receiver Opererating Characteristic Curve. 13, 23, 24, 26–28

CPS Cyber Physical System. 4, 5

CRISP-DM Cross Industry Standard for Data Mining. 3, 14

DT Decision Tree. 2, 3, 8–12, 19–21, 25–27

FN False Negative. 22, 24

FP False Positive. 22, 24

IoT Internet of Things. 1, 4–6, 14

LR Logistic Regression. 3, 7, 8, 19, 20, 25, 26

LSTM Long Short-Term Memory. 6, 13

ML Machine Learning. 2–11, 14, 16, 19

PdM Predictive Maintenance. 1–3, 5–8, 11, 13–15, 29

RF Random Forest. 2, 3, 10–13, 19, 20, 25–28

RFID Radio Frequency Identification. 5

ROC Receiver Operating Characteristic. 23

TN True Negative. 22–25

TP True Positive. 22, 24, 25

XGB XGBoost. 3, 13, 18, 19, 25–29

1 Introduction

1.1 Overview

To accurately know the probability of machine failure and its rate of decay has become a graspable concept through the evolution of Internet of Things (IoT) and cloud technologies, data mining processes and powerful Artificial Intelligence (AI) structures for data analysis. For decades, the manufacturing industry settled with reactive maintenance or Run-2-Failure, which describes the reparation and replacement of machine parts after the machine had already malfunctioned. These common events are, till today, a nuisance to businesses as overall production can come to a halt, as well as unpredictably causing malfunctions in other areas of an entire physical system (Çinar et al., 2020; van Dinter et al., 2022; CMMS, 2022).

In 2011, the arrival of the fourth industrial revolution, industry 4.0., had been dubbed by the German government in which production would be overtaken by an intelligent enterprise, where machines, computers and human capital would be seamlessly interconnected in a digital space (Zhang et al., 2021; Zheng et al., 2021). For the idea to come into effect, industry 4.0. requires implementation of IoT technologies, such as equipment sensors, cloud computing for machine-to-machine and human-to-machine communication, and AI for the potential to automate processes of entire product life cycles (Rehman et al., 2018). Among such processes, to replace the outdated and unconventional reactive maintenance strategy, Predictive Maintenance (PdM) became a part of the reality that Industry 4.0. offered (Zhong et al., 2017).

During the turn of the century, statistical models were implemented to carry out survival analyses in the medical fields in order to predict the likelihood of survival. These statistical models had then in recent years been adopted by the manufacturing and civil engineering sectors to predict the likelihood of failure of machinery, buildings bridges, etc., such as the Kaplan-Meier Analysis and Cox's Regression (Kaplan and Meier, 1958; Rich et al., 2010; Wen et al., 2022). The applications of such models would continue to evolve through more powerful machine-learning structures and expand across industries, until finally, the immense volume of data that industry 4.0. produced made way for PdM. PdM is a maintenance strategy that utilizes various condition monitoring sensors and statistical tools to predict the likeliness of machine failure during a given time. The

sensors can feed both historical and real-time data, such as vibration and temperature data, to a computerized maintenance management system (CMMS, 2022) where appropriate Machine Learning (ML) tools are implemented, from basic statistical models like regression, to more complex ML structures like Long Short-Term Memory (LSTM) deep learning (Chen et al., 2021; Udo and Muhammad, 2021).

Although high initial investments are required, the competitive advantage of organizations having PdM normalized and automated in production is powerful, as costs could be reduced in an unprecedented manner. Sources claim a reduction of maintenance costs by up to 30%, reduction of downtime by up to 45%, and reduction in machine breakdowns by up to 70% (Fiix, 2022; Infraspeak, 2022).

In recent years, through new advancements in AI capabilities, academia has witnessed a rise in research into PdM using ML techniques. A common method that had been frequently used as a benchmark model are tree-based models, like Decision Tree (DT) and Random Forest (RF), as these models have pertained to acceptable results with straight-forward interpretation (Carvalho et al., 2019; Calabrese et al., 2020; Kulkarni et al., 2018). Despite its popularity in academic research, a topic of interest arises of whether the implementation of tree-based analyses for PdM has real-world application in terms of accurate predictions, feasibility in use and interpretation, as well as robustness across multiple systems. Thus, the following research question has been drafted.

1.2 Research question

To what extent is the implementation of tree-based models effective and feasible in predictive maintenance under industry 4.0?

1.3 Research Aim and Objectives

The aim of this research is to aid in providing stakeholders of the manufacturing sector and providers of relevant software and solutions an oversight of the effectiveness, feasibility and versatility of tree-based models in the context of PdM. In order to provide this oversight, the following objectives have been drafted to support in answering the research question.

- Collect and explore a cross-sectional dataset that produces or simulates a PdM classification problem;

- Clean and prepare the cross-sectional dataset for different classification models to be fit to;
- Produce and compare tree-based models, as well as one benchmark classification model, using appropriate evaluation techniques;
- Recommend to stakeholders, the extent to which tree-based models are appropriate for real-life application;
- Recommend how future research can continue building towards more optimal and business relevant solutions based on the findings of this research;

1.4 Structure Overview

In order for the readers to understand the intentions and intricacies of the research, this paper will begin with the literature review which will define and explain all necessary and relevant phenomena, statistical and ML concepts. This will begin with a definition and background on Industry 4.0. and PdM in order to grasp its importance and relevance across industries. The second part of the literature review will involve a deep-dive into the necessary mathematical concepts that form the various statistical and ML models. The models include Logistic Regression (LR), DT, RF and XGBoost (XGB).

The next section is dedicated to the methodology in which the structure of the Cross Industry Standard for Data Mining (CRISP-DM) is utilized in order for the reader to clearly navigate the processes behind this research. CRISP-DM is a common practice for data mining experts, and splits the entire methodology into six parts. Namely, business understanding, data understanding, data preparation, modeling, evaluation, and finally deployment.

The ensuing two sections of the paper are research and discussion, and finally the conclusion. Research and discussion will display and deep dive into the results of each model's performance, as well as discussing the scores each model attained from all chose evaluation techniques. The conclusion, will provide the reader with the researcher's recommendations to the real-world application of tree-based models as well as recommendations to the direction related future academic research should take.

2 Literature review

2.1 Industry 4.0

2.1.1 Definition

Industry 4.0 is a term dubbed by the German government in 2011 and Klaus Schwab, founder of the World Economic Forum as the fourth industrial revolution (Zhang et al., 2021). An industrial revolution of the 21st century that describes the intelligent interconnectivity between the machinery of the manufacturing industry, automated process and its stakeholders (Zhang et al., 2021; Zhong et al., 2017). This concept is made possible through the emergence of various information and communications technologies, automation through AI, specifically a vast and diverse configuration of ML techniques, as well as the internet-of-things, allowing the physical manufacturing realm to bridge with the digital that creates a business model called a Cyber Physical System (CPS) (Zhong et al., 2017; Tao et al., 2014; Gohel et al., 2020).

A major vision to be pursued in industry 4.0 is the manifestation of smart factories. Smart factories are a product of the above-mentioned technologies under the CPS business model, in which machinery and equipment communicate with each other, ensuring a fluid work process, by alerting other systems when sub-optimal production levels are being achieved, or more significantly, malfunctions and machine failure (Zhong et al., 2017; Zheng et al., 2021). Human intervention with technologies have also evolved under the philosophy of smart factories, in which the format of interaction has changed. Through a CPS, people are linked with the machines through a digital space, ranging from intelligent maintenance systems to on-site simulations using virtual and augmented reality. Ultimately, the goal of smart factories is to maximize productivity and sustainability of manufacturing processes. For this goal to come to fruition, substantial amount of raw data is demanded real-time to give the relevant stakeholders a detailed overview of each equipment's output and condition, and for this to be possible, all conventional equipment and components need to be equipped with IoT technologies (Khelif et al., 2017; Chen et al., 2021; Nacchia et al., 2021).

2.1.2 Internet-of-Things and Cloud Manufacturing

IoT-enabled manufacturing refers to traditional machinery and equipment fitted with IoT technologies, in which they attain the ability for machine-to-human, machine-to-machine and human-to-machine communications enabling the business to intelligently monitor, analyze and support in making data-driven business decisions (Calabrese et al., 2020; van Dinter et al., 2022). IoT equipped machinery are able to sense, adapt and communicate manufacturing logic with one another and its human stakeholders, through the vast amounts of data it is able to acquire and share in real-time (Zhang et al., 2021).

An introductory method to kickstart IoT implementations and share data from one machine to the next, is through Radio Frequency Identification (RFID). In terms of manufacturing, RFID tags and readers are applied to the equipment which may grant its operators complete visibility of the production process when harmoniously integrated to a CPS (Zhong et al., 2017).

More suitable to the data-mining capabilities of industry 4.0., is the concept of cloud manufacturing, which is made possible through IoT technologies and cloud computing. Cloud computing is the ability to utilize computer services, such as storage, networking, software, intelligence, analytics, etc., via the internet rather than the traditional method of using local servers. Without the constraints of localization, the implications of cloud manufacturing reaches the entire product life cycle, from ordering supplies of raw materials up until the delivery of the finished good. Management of all phases of manufacturing via the internet is made possible with managers, specialists and other relevant stakeholders having access granted to their appointed phases (Jamwal et al., 2021; Nacchia et al., 2021; Canizo et al., 2017).

2.2 Predictive Maintenance

2.2.1 Concept

A significant advantage of data collected from sensors and monitoring technologies made available by the presence of the industrial IoT, is its applications with ML (Calabrese et al., 2020; Paolanti et al., 2018; Gohel et al., 2020). PdM, or also known as Condition-Based Maintenance or Prognostic and Health Management, is the process to intelligently monitor an organization's equipment and machinery to detect anomalies and deterioration

followed by support reliable analyses in order to accurately forecast when reparations, parts orders and replacements, or other actions needs to be taken Wen et al. (2022); Carvalho et al. (2019); Çinar et al. (2020).

PdM is a powerful improvement compared to its two predecessors, Reactive and Preventive Maintenance. Reactive Maintenance, as the name suggests, performs repairs after the equipment has been damaged, whilst Preventive Maintenance describes the process of consistent scheduling of maintenance activity in order to prevent costly damages in the future. Before PdM, entities following the Reactive and Preventive Maintenance process had to accept the inevitable disadvantages, that either the organization would face machine down-time from reacting to machine failures that accumulated costs through repairs and rescheduling with stakeholders, or through excessive maintenance that could have saved the organizations unnecessary expenses (CMMS, 2022; Zhong et al., 2017).

The emergence of ML and its application to PdM meant, that these disadvantages could be minimized. Although high initial costs in the required IoT equipment, Computerized Maintenance Management System, and data-oriented specialists are involved compared to reactive and predictive maintenance, machinery maintenance costs previously thought to be unavoidable could be prevented (CMMS, 2022). Nowadays, maintenance software and data personnel can produce forecasts that can accurately inform when maintenance is needed, either in the form of probabilities or absolute values, depending on what techniques are being implemented, and ultimately tackling the previously-known shortcomings.

In recent years, the technological innovations in PdM have taken major strides. The types of raw data collected started expanding, from feasible data types, like temperature, vibration and pressure to complex unstructured noise and image data (Wu et al., 2017; Çinar et al., 2020; CMMS, 2022). Such an ability for precise predictions of generic maintenance data and interpretation of abstract data has been made possible through the development of field-specific ML structures and techniques. From more simple techniques such as the time-series method, Autoregressive Integrated Moving Average (ARIMA), and linear regression, to deep learning and even more extensive neural networks, such as Artificial Neural Networks, Recurrent Neural Networks like Long Short-Term Memory (LSTM)s and Gated Recurrent Units, as well as transformers (Dalzochio et al., 2020; Wu et al., 2017; Nacchia et al., 2021). The ML structures relevant for the research of this

study, will be further discussed in the ensuing chapters.

2.3 Statistical and Machine Learning in Predictive Maintenance

2.3.1 Supervised learning and Classification Problems

The focus of this research is to make predictions using labeled data, hence to fully grasp the concepts in the following sub-chapters, the characteristics of supervised learning need to be mentioned. Supervised learning is a technique in which a statistical or ML model is fit onto data observations that include predictor variables X and also have a corresponding outcome variable Y. Its counterpart, unsupervised learning also includes some form of measurement X, but does not come with an outcome Y, making statistical learning problems that fall in this category much more challenging, as models cannot just learn what outcome they should detect (Provost and Fawcett, 2013).

Furthermore, PdM can be tackled as a regression or classification problem based on what form the outcome variable Y comes in. If the outcome variable is quantitative or numeric, then the research typically faces a regression problem. if the outcome variable can be defined as categorical or binary, then classification. As the datasets the models of this research will be fit to contain categorical outcome variables, i.e., the type of failure the machine is experiencing, the model will solve a supervised classification problem (James et al., 2013).

Although having a contradicting name, one of the most elementary statistical classification models is LR. For comparison, the function of a simple linear regression takes the form of:

$$Y = \beta_0 + \beta_1 X \tag{1}$$

Y is the approximate prediction at a given observation X, and is made up of a constant, β_0, a slope, β_1 and the observation X at a given instance and possibly time, and can be a value between $-\infty$ and ∞.

The logistic function takes a similar approach, but as the interest in classification problems is to attain a probability of which class a certain observation belongs to regarding the outcome variable, the value needs to lie between 0 and 1. Hence:

$$p(X) = \frac{e^{\beta_0+\beta_1 X}}{1 + e^{\beta_0+\beta_1 X}}, \tag{2}$$

where the foundational logarithmic value, Euler's number, e, is implemented to transform the values, to always lie between 0 and 1. The betas under the logistic function are estimated through maximum likelihood where the likelihoods of an observation being assigned to one class or the other, is maximized. Thus, rather than a straight line being plotted like in linear regression, plotting the logistic function returns an S shaped slope that will never exceed the boundaries of 0 and 1 (Gohel et al., 2020; James et al., 2013).

Although empirical research found that LR provided good results when dealing with simple data, under the context of PdM, the model is unlikely to outperform more complex statistical and ML models in terms of accuracy and errors made in classifying cite (Çinar et al., 2020; Binding et al., 2019).

2.3.2 Tree-based Models and the Decision Tree

Tree-based models are popular forms of ML algorithms that use one or more DT. If successfully executed, the DT starts at the 'root', that represents the entire predictor variable space, and creates conditions the observations have to meet for the selected input variable called decision nodes, subsequently splitting the observations into two subsets (James et al., 2013). In a hierarchical manner, this splitting process continues with each input variable like branches called internal nodes, until it reaches a conclusion called leaf nodes or terminal nodes that satisfy certain predicted specifications of the target variable. Towards the end, visually, this process a tree-like structure, with the values in each node representing the average of that specific sub-group (Taddy, 2019).

The underlying concept of the decision tree reveals a parametric regression model, that can be utilized for both regression and classification problems. Parallel to a conventional regression model $p(X)$, the tree has predictor variables X and output variables Y, with a predicted response, or predicted $\hat{Y}$ after each split into the two new daughter nodes or $J-$disjointed Regions $[R_j]_1^J$, as well as at the conclusion of the model that is shown in the form of the tree's leaf nodes (Friedman, 2006). For any given number of dimensions or quantity of predictor variables X, $\hat{Y} = T_J(X)$ is is assigned to the specified partitioned region X, displayed as:

$$X \in R_j \Rightarrow T_J(X) = \hat{Y}_j \tag{3}$$

Within each partitioned subgroup, the difficulty does not lie in searching for the optimal value that minimizes the its prediction risk, as it is a feasible process and is described as:

$$\hat{Y}_j = \underset{y'}{\operatorname{argmin}} E_y[L(y, y')|x \in R_j] \tag{4}$$

The difficulty actually lies within the act of partitioning the subgroups into the structured and methodical internal and terminal nodes that the researcher desires, as their are a myriad of possibilities to how the nodes are partitioned into the next. However, most possibilities will have poor predictive performance (Friedman, 2006). Hence, to create a well-performing splitting rule, the goal will be to minimize the loss function at each level of the hierarchy, and similar to the deviance functions of a parametric regression model, the loss function of the tree model is examined to be identical (Taddy, 2019). For example, while sum squared minimization on a regression, that is also used for classification trees, is written as,

$$\sum_{i-1}^{n} (y_i - \hat{y}_i)^2, \tag{5}$$

two examples of tree model loss functions that are numerically similar, multinomial deviance (entropy) and Gini impurity minimization, are formulated in the following way, respectively:

$$-\sum_{i-1}^{n} \sum_{k=1}^{K} y_{ik} \log(\hat{y}_{ik}) \tag{6}$$

$$\sum_{i-1}^{n} \hat{y}_{ik}(1 - \hat{y}_{ik}) \tag{7}$$

The advantages of DTs, as well as its succeeding models have continued to display a strong presence in ML, firstly, due to its simplicity in interpretation. Their signature binary tree graphic as Friedman et al (2006) stated, is the source of the model's popularity. Regardless of the number of dimensions or variables, this model can be illustrated in two dimensions (Friedman, 2006; James et al., 2013).

Furthermore, its simplicity also derives from its unparalleled feasibility. Without compromise, these models can process numerical, binary and categorical predictor variables, as well as handling missing data in an elegant manner. They are flexible to the transformation of explanatory variables whilst ignoring irrelevant ones (Friedman, 2006).

The disadvantage however, is that DTs are not stable. Meaning, any minor replacements in the observations can completely alter the structure of the tree, and ultimately its predictions. Hence, causing high potential variance in its predictions. Secondly, continuously, partitioning the data, DTs tend to end up making predictions at the terminal nodes with minimal observations. Researchers are able to overcome this obstacle through substantial data and tuning. Finally, DTs alone will not have the same predictive prowess as other ML algorithms. The ensuing sub-chapter will shed light on how an extension of the DT concept can be exponentially improved through the aggregation of multiple DTs, in the form of ensemble models (Friedman, 2006; James et al., 2013).

2.3.3 Ensemble Models: Random Forests

Ensemble models can be defined as an aggregation of multiple simple models, like DTs, to build a more complex and potentially more powerful ML structure. The first ensemble model to examine is the RF (James et al., 2013).

RFs have the attribute to overcome the disadvantage of DT and their tendency for high variance in their prediction, by utilizing a more elaborate method of bootstrapping (Taddy, 2019). Bootstrapping is a popular sampling technique, where a sample is randomly selected from the population, and re-sampled by replacing the observations with, either unseen observations drawn randomly from the population, or even duplicating existing observations in the sample (Shin, 2020).

According to James et al. (2013), the act of generating multiple training sets based on bootstrapping, is called bootstrap aggregation, or bagging. Bagging can be understood by the assumption that each set of n independent observations $(Z_1, Z_2, Z_3, \ldots, Z_n)$ comes with a variance σ^2. Hence, the variance of the mean of the observations $\bar{Z}$ can be formulated as $\frac{\sigma^2}{n}$, meaning averaging the observations reduces variance.

The premise of averaging observations to reduce variance can be utilized for bagging. Each training set is fitted into their own prediction models, and the prediction results are then transformed into one mean prediction, that now has an increased accuracy for

testing and desired lower variance (James et al., 2013). To formulate:

$$\hat{f}_{bag}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^{*b}(x) \tag{8}$$

where $\hat{f}_{bag}(x)$ describes the averaged value of each individual prediction $\hat{f}^{*b}(x)$ of each learning model in the ensemble that resulted from their own exclusively bootstrapped training sets B.

RF is a learning method that embodies these statistical techniques, yet exceeds them by its capacity to de-correlate the DTs within its ensemble. The method used to de-correlate the trees is by allowing each DT to only utilize a marginal number of m predictor variables from the full set of p predictor variables. The common rule, behind the amount of m predictors the DT is allowed to consider for its splits can be drafted as $m \approx \sqrt{p}$ In other words, the number of m predictors is equal to the square root of all available predictor variables. Without this added trait, all DTs in the ensemble are likely to resemble each other in, that it selects the same strong predictor variables for the initial split at the top of the tree. Furthermore, the trees grown in an RF utilizes tree pruning to create smaller and more effective trees, which in combination with the randomized m predictors, prevents the model from overfitting, meaning that it does not get too tailored with the dataset it is trained on (James et al., 2013).

Under the context of PdM, RF classifications have been empirically successful in terms of prediction accuracy on real and synthetic data. The approach by Canizo et al. (2017), in which an RF model was implemented on the status and operational dataset of wind turbines, was able to achieve an overall predictive accuracy of 82.04%. Kulkarni et al. (2018), utilized the RF model on sensor data of supermarket refrigeration systems and achieved a predictive accuracy of 89%. Across literature, a common obstacle researchers faced that hindered more optimal results was the lack of observations, as the status of machine failure only represented a small amount of the overall data, the classes of the raw dataset were significantly imbalanced (Canizo et al., 2017; Kulkarni et al., 2018; Carvalho et al., 2019). Furthermore, the complexity of growing an RF, as well as the time of computation, can make other ML models more desirable (Carvalho et al., 2019).

2.3.4 Ensemble Models: Boosting

Another family of ensemble methods belongs to the boosting technique. Much like RFs, boosting relies on an extensive number of DTs, or other simple base-learner models, making boosting methods also suitable for regression and classification problems. However, in contrast to RFs with each individual tree fitted to its bootstrapped sample to simultaneously produce predictions, boosting does this sequentially and with a modified version of the data set after each DT is fit (Canizo et al., 2017; Kulkarni et al., 2018).

With each iteration m, the boosting model evaluates how it had performed with all the observations of the training set and applies weights w to the ones it has misclassified. At each iteration, the model focuses on the observations it did not classify correctly, ultimately concentrating on the most complex observations. Every time the model iterates $m = m_1, \ldots, m_{stop}$, the weight vector $w[m] = w_1[m], \ldots, w_n[m]$ appends the new weights whilst saving the previous weights. Finally, an iteration-specific coefficient grants heavier weights onto iterations that performed best and decides to focus on the class observations that were assigned to the most, whilst also considering all error rates achieved during the entire sequence. This gives the overall model one last attempt to strategize a solution to apply weight to the observations that are difficult to class.

An evolution of boosting is gradient boosting, which was elaborated by Friedman et al. (2004), adopts a classic non-linear optimization function of regression models known as steepest descent. Hence, with a defined problem containing predictor variables X and the outcome variable Y, we minimize the risk of the predicted regression function $\hat{f}(.)$ as such:

$$\hat{f}(.) = \underset{f(.)}{\operatorname{argmin}} \left\{ \frac{1}{n} \sum_{i=1}^{n} p(y_i, f(x_i)) \right\} \tag{9}$$

where its aim is the minimize the $p(.)$ that indicates the loss function, for example one that uses L2 regularization $p(y, f(.)) = (y - f(.))$ to adjust the penalty, y denotes each instance i of the outcome in the sample and $f(x_i)$ the selected function for the predictor variable x at each instance i.

The way this is utilized in boosting is that rather than fitting the base-learner model by considering and appending the weight vector $w[m]$ as described before, the gradient booster takes its considerations and improves using the negative gradient vector $u[m]$ of the loss function. The model will build upon the vector value produced in the last

iteration $m-1$.

In 2016, a scalable model of gradient boosting was introduced by Chen and He (2014), called XGBoost (XGB) or extreme gradient boosting (Azmi and Baliga, 2020). Though possibly lower in accuracy compared to gradient boosting, the advantage of XGB is it does not overfit the training data by implementing regularization in the form of an L1 or L2 loss function, in which either the absolute value is considered as penalization or the squared value, respectively.

According to a review by Çinar et al. (2020), in contrast to RFs, XGBs have not been very popular in PdM. Lim and Chi (2019) had shown acceptable results in accuracy and its evaluations such as the Area under the Receiver Opererating Characteristic Curve (AUC-ROC), an evaluation technique used for classification models, which will be further elaborated on in the methodology chapter. However, in another study by Udo and Muhammad (2021), the variance of XGB was consistently higher when predicting machine failure in wind turbine components compared to the deep learning structure LSTM.

3 Methodology

To aid in structuring the entire data mining and analysis process in a flexible manner, the Cross-Industry Standard Process for Data Mining (CRISP-DM) will be implemented. CRISP-DM was introduced in 1999 as a universal data science process that is aimed to be applicable in any industry (Chapman et al., 1999; Smart-Vision-Europe, 2015). The process is broken down into six parts to report on, consisting of the business understanding, data understanding, data preparation, modeling, evaluation and deployment. Although the general essence of CRISP-DM is to iterate in the mentioned order, the process is considered agile, as the data researcher needs to evaluate and make adjustments in the previous steps. For example, jumping back to data preparation in order create or adjust new data transformations or jumping to modeling in order to fine tune the statistical and ML algorithms.

3.1 Business Understanding

As no specific organization is involved with the drafting of this research, the general characteristics and objectives of companies utilizing PdM will be mentioned. For a manufacturing, logistic or transportation business to benefit from PdM, the business would have already made heavy substantial investments in IoT technologies, i.e. the sensors that need to be installed throughout the entire manufacturing process, to help collect the substantial data the PdM models need to produce reliable results. The primary objective for the company to implement PdM is to reduce operational costs and increase productivity, which is achieved through minimal operational downtime, reduced maintenance costs and efficient planning that is enabled through a reliable health overview of the machinery (Carvalho et al., 2019; Florian et al., 2021).

3.2 Data Understanding

To overcome the difficulties of obtaining and publishing results on a real machine failure dataset, the AI4I 2020 Predictive Maintenance Dataset (Dua and Graff, 2017; Matzka, 2020)is a synthetic dataset that simulates a real milling machine and mimics the characteristics of real PdM problems. The dataset was acquired from the repository of the University of California, School of Information and Computer Sciences, and is open to

researchers aiming to advocate, exchange and develop experiments involved with PdM.

This multivariate dataset consists of 10,000 observations with 14 feature variables and a binary outcome variable, namely, whether the machine has failed or not. A description of each variable, including the data types, is listed below:

- **UDI** (int64): A unique identifier for each observation ranging from 1 to 10,000;
- **Product ID** (object): An identifier for the relevant machine of the observation;
- **Type** (object): This variable consists of either the letter L (low), M (medium), or H (high) ;
- **Air temperature [K]** (float64): Air Temperature in Kelvin;
- **Process temperature [K]** (float64): Process temperature in Kelvin;
- **Rotational speed [rpm]** (int64): The machine's rotational speed in rounds per minute. The values are created with the assumption of 2860 Watts capacity;
- **Torque [nm]** (float64): Torque in Newton-Meters;
- **Tool wear [min]** (int64): The gradual failure of the tool or machine in minutes;
- **Machine failure** (int64): A binary value indicating whether the machine has failed (1) or not (0);
- **TWF** (int64): Binary value of whether the machine failure was specifically a tool wear failure;
- **HDF** (int64): Binary value of whether the machine failure was specifically a heat dissipation failure;
- **PWF** (int64): Binary value of whether the machine failure was specifically a power failure;
- **OSF** (int64): Binary value of whether the machine failure was specifically an over-strain failure ;
- **RNF** (int64): Binary value of whether the machine failure was a random failure;

Overall, the dataset is clean, and contains no missing values or suspicious outliers. As described in the literature review, like many ML algorithms, the results of tree-based models are heavily affected by the existence of imbalanced classes of the binary outcome variable. This is true with the AI4I dataset, as 96.61% of the `Machine failure variable` showed no failure, whilst 3.39% of all observations did fail. This of course, reflects the reality of gathering machine failure data, since machinery and tools do not tend to fail in as many instances as they are operational.

To summarize the relevant continuous feature variables, the `Air temperature` and `Process temperature` recorded a mean of around 300 K and 310 K with a standard deviation of 2 and 1.48, respectively. The `Rotational speed` displays a mean of 1,536.78 RPM with a standard deviation of 179.28 RPM. `Torque` has a mean of around 40 newton-meters, deviating about 9.97 newton-meters. And lastly, the `Tool wear` has a mean of around 107.95 minutes with a standard deviation of 63.65 minutes.

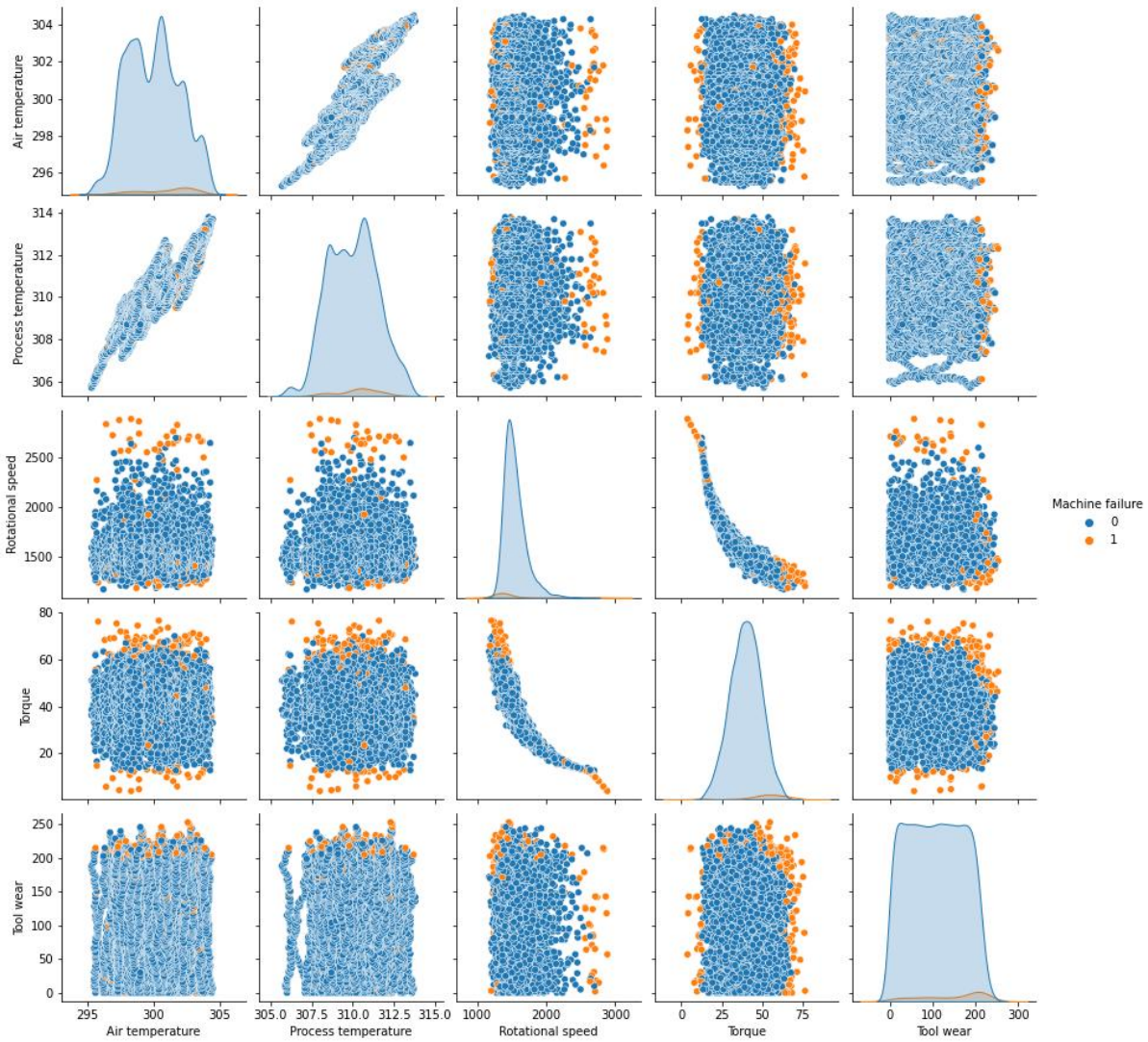


Figure 1: Plot of Pairwise Relationships between Feature Variables

As shown in 1 some of the feature variables show signs of correlation. These correlations can be found with the Air temperature and Process temperature variable, in which a correlation of 0.876 has been recorded. Torque and Rotational speed also recorded high correlations at -0.875. Although valuable to detect beforehand, since this research is concerned with tree-based models, no disqualification of feature variables due to multicollinearity is necessary, as tree-based models are not hindered by this assumption. However, the distribution of the variable Type has much more data points classified as as a 1 for `Failure` in the `Type_L` machines compared to the other two types, which the tree-based models could favor as a dominant variable to help make decisions further along.

3.3 Data Preparation

Minimal data cleaning steps were undertaken to prepare the dataset to be universally applicable for various models. As mentioned prior, no data points needed to be filled with missing-value solutions, and feature variable transformations were also unnecessary. To initiate with the preparation, several feature variables were removed that would affect the results of the models. The dropped variables included the UDI, the product ID, as well as the different types of failures as the desired outcome variable for prediction is exclusively the binary indication of whether the machine had failed or not in general.

Next, though assumed the feature variable Type, that consisted of the categories, low, medium and high had to be transformed into dummy variables, the entire column had to be dropped as the distribution of failures were too high in `Type_L`. The final alteration for the data cleaning involves tailoring, involves tailoring the feature variable names to suit the requirements of the XGB documentation. XGB cannot function when column names include special symbols, which this dataset contains, as some feature names have square brackets. These square brackets are removed as the final measure.

After the cleaning, the dataset had to be segregated into two new subsets. One dataset, that includes just the relevant predictor variables (predictors), and the other being an array containing the machine failure outcomes (target). Afterwards, the two datasets are then employed for further partitioning into training and testing datasets. Scikit Learn's `train_test_split` aids the data preparation by randomly splitting all observations of the predictor and target variables into training and testing subsets. The reason behind the train-test-split is to prevent the model from overfitting, meaning it is tailored to well to the dataset and has no ability to generalize to other datasets. The test subset of data is more commonly known as the holdout data. The hold out data is a portion of actual historical outcome data, that is left after training for the models to be exposed to, so that the researcher can compare the models' predictive prowess to the actual outcome results.

Using Scikit Learn, 80% of the feature and outcome data is split into a training subset, while the rest is held out for testing. As stated in data understanding, there is a substantial class imbalance between machines that have failed and machines that have not. Due to the randomization of observation assignments into the training and testing subset, their is a high likeliness that one of the subsets may not even contain any of the 3.39% of the failed instances. To overcome this obstacle, the train_test_split function offers the option

to stratify, meaning the data will be split contain the same proportions of the two class labels.

3.4 Modeling

Four models have been selected for the purpose of the research. Namely, LR, DT, RF and the boosting ensemble technique, XGB. The dataset has been prepared to seamlessly, integrate for each model to learn from. Also implemented on each model is GridSearchCV in Scikit Learn's library. GridSearchCV supports the researcher in finding the optimal parameters in each model. To use GridSearchCV, a model needs to be defined as well as a dictionary containing all of the desired parameter values.

Another feature of GridSearchCV is the use of cross-validation (CV) in the process of searching for the optimal parameters of each specified model. Cross-validation is an extension to the concept behind the holdout dataset, and functions by splitting the dataset into a user-defined number of subsets. These subsets are called folds and are typically split into 5 or 10 folds. At each iteration, and for one iteration only, one of the folds becomes the testing subset and the rest are training subsets. This continues until the each fold has been used as a testing subset. At each iteration an accuracy score is produced, and is ultimately used to calculate the average accuracy as well as the variance. To ensure the proportion of class labels remain the same as in the train and test subsets, GridSearchCV automatically implements stratification splitting the dataset.

Finally, each model used to conduct the research has different parameters. Hence multiple dictionaries need to be written that specify the parameter as well as the values the researcher wants to test for optimal results. Below are the selected statistical and ML models as well as the chosen parameters and hyperparameters that will be tuned.

3.4.1 Logistic Regression

- **'penalty'**: This parameter helps improve the fit and simplicity of the data by adding some form of regularization through weights on linear models. Regularization was briefly discussed in the theoretical discussion of boosting. The penalty terms considered are the L1 and L2 penalty, elasticnet or none;

- **'C'**: A hyperparameter that adjusts the power of the chosen penalty. The value has

to be a float gives more power to the penalty the smaller the value is;

- **'solver'**: A selection of different optimization algorithms to minimize the loss function;

LR will be used as the benchmark model to compare the tree-based models with.

3.4.2 Decision Tree Classifier

- **'criterion'**: A selection of functions to how the researcher wants to measure the quality of the split. The choice is between 'gini', 'entropy' and 'log_loss';
- **'max_depth'** (object): Specifies how many levels the DT should have. This parameter requires an integer value;

The DT can also be considered a benchmark as empirical evidence suggests that this model will not be competitive with more complex models (Friedman, 2006; James et al., 2013).

3.4.3 Random Forest Classifier

- **'n_estimators'**: The number of DT classifiers the ensemble model should build;
- **'max_features'**: The number of feature variables to consider when identifying the best split;
- **'min_samples_split'**: The minimum number of samples needed for an internal node split to be allowed;
- **'min_samples_leaf'**: The minimum samples required to be a leaf node, or the terminal node;

Based on the reviewed empirical research (Carvalho et al., 2019; Calabrese et al., 2020; Kulkarni et al., 2018), the RF classifier should yield acceptable results that can be considered for deployment in a real-life manufacturing, logistics or transportation problem.

3.4.4 XGBoost Classifer

- **'n_estimators'**: The number of DT classifiers the ensemble model should build;
- **'learning_rate'**: The learning rate controls the speed of which the model is allowed to modify and optimze after each iteration of grown DTs;
- **'max_depth'**: Specifies how many levels the DT should have. This parameter requires an integer value;
- **'subsample'**: The ratio of the entire training sample the model should consider growing the next estimator;
- **'colsample_bytree'**: The ratio of columns the model should consider before growing each tree. This step repeats after each iteration of growing an estimator;
- **'gamma'**: Minimum loss reduction that is needed to continue splitting. The larger the value, the more conservative the model is;

Based on empirical research (Lim and Chi, 2019; Azmi and Baliga, 2020), this model is likely to have the most optimal outcomes when evaluating and comparing against the other models.

3.5 Evaluation

3.5.1 Accuracy

A value that is computed simultaneously with the fitting and predicting of the model will be the accuracy score. The accuracy score, or classifier accuracy is simply the number of correct decisions divided by total number of decisions made by the classification model. It is a fundamental practice to compute this score and simultaneously informs the researcher of the error rate, which is simply calculated $1 - accuracy$ (James et al., 2013; Pedregosa et al., 2011). It is simple to measure and interpret but it can be often misleading, especially when dealing with class imbalances, as mentioned in previous subchapters.

3.5.2 Confusion Matrix

A confusion matrix is also a feasible and interpretive evaluation technique that provides valuable insights to how well a model performs, and specifically, which class the estimator

is better at assigning. The shape of the matrix considers the number of outcome classes n to produce an $n \times n$ table. The columns represents the actual classes, while the row represents what the model had predicted (Provost and Fawcett, 2013; Pedregosa et al., 2011). A simple binary problem (0 and 1), identical to the research, can be represented by a confusion matrix as illustrated below:

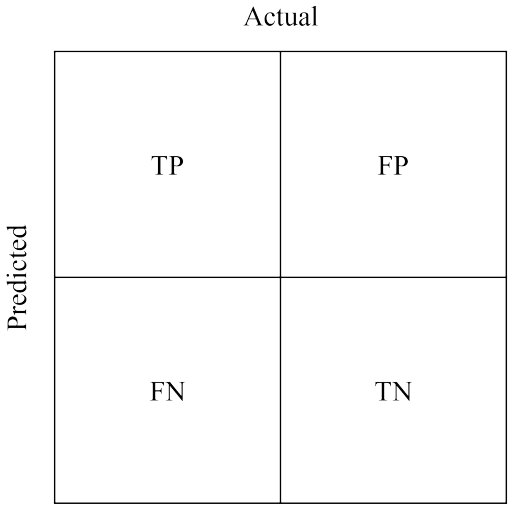


Figure 2: Confusion Matrix

The first cell is called True Positive (TP) and represents all observations in the sample that were predicted to be 0 and and actually are. False Positive (FP), to the right, describe the observations that were predicted to be 1 but are in fact a 0. In contrast, False Negative (FN) in the second row are the observations classified as 0 by the model but are indeed a 1. Finally, True Negative (TN), are the observations classified as 1 and are actually 1.

Asides from informing what class each model is better at assigning observations to, the confusion matrix displays the extent to how the skewed classes make the accuracy score unreliable. Furthermore, the value of each cell of the confusion matrix allows for the computation of some informative metrics that make up the classification report.

3.5.3 Classification Report

An extension to the findings of the confusion matrix is the classification report, that in the case of this research, is represented by three different metrics computed on both outcome classes. The first metric is precision, which is simply the model's accuracy to predict a

single class, rather than the accuracy of both classes combined. Precision can be written as $TP/(TP + FP)$ for class 0 or $TN/(TN + FN)$ for class 1. The second metric is the recall which describes the frequency of the model having been correct and it is computed with the formula $TP/(TP + FN)$ for 0, or $TN/(TN + FP)$ for 1. The third and final metric is the F1-Score which describes the proportional average of the precision and recall value, and it is computed through $2 \times (precision \times recall)/(precision + recall)$ (Pedregosa et al., 2011).

3.5.4 AUC-ROC

The final evaluation approach is the Area Under the ROC Curve (AUC-ROC). The Receiver Operating Characteristic (ROC) is a two-dimensional graph that plots the *recall* of positive instances 0 on the y-axis, and $1 - recall$ of negative instances 1 on the x-axis at different thresholds. The two recalls are also commonly known as sensitivity and specificity, respectively (Taddy, 2019). This visualizes the trade-off between *sensitivity* (or true positive rate) and $1 - specificity$ (or false positive rate), with the ROC curve representing the probability curve that is plotted across each threshold instance. The closer the plotted model is at a given threshold to $(0, 1)$ on the graph, the better the model is at classifying. The diagonal spanning from $(0, 0)$ to $(1, 1)$, describes perfect random classification. Hence, the closer the model is plotted to this diagonal, the worse it is at classifying. The AUC-ROC is a calculation of the entire area underneath the ROC curve and is displayed as a value between 0 and 1. The higher the value, the better the model is at predicting and assigning the sample into the correct classes (James et al., 2013; Pedregosa et al., 2011).

3.6 Deployment

The evaluation of each model will help the researcher review their credibility to help relevant stakeholders make their daily and long-term business decisions. If deemed worthy, a deployment plan will be drafted out of the data mining results (Chapman et al., 1999). Since the class proportions of the data is skewed, the researcher will not rely on classification accuracy to determine a model's initiation into the deployment strategy. The confusion matrix and classification report will be more suitable to assess the imbalanced dataset. Specifically, the results of interest will be the proportion of TNs compared to the

combined total of FNs and TNs that is displayed in the confusion matrix, the recall of failures in the classification report, as well as the F1-score that will inform the researcher of tradeoff between precision and recall. The AUC-ROC will bring even more clarity to the deployment strategy by informing stakeholders of potentially deploying another model with a better TP rate to FP rate that could be more desirable in the event the data size increases or becomes more complex. The attractiveness to deploy one of the tree-based models will be determined through the attractiveness of the summary statistics and the visualizations of the AUC-ROC.

After the deployment of the chosen model, the continuous monitoring and maintenance of the model's performance becomes crucial. This can be done using the above-mentioned evaluation methods. These evaluation methods should be utilized in either a scheduled manner, such as automatically on a daily or weekly basis, or whenever there are changes in production, such as increases or decreases of the output level or the installation of new machine components. Additionally, performance constraints can be added that will alert the relevant parties whether the model is satisfying the thresholds. For example, alerting analysts when the recall of failure is under 0.7. Maintenance can be further enhanced through the constant training of the model with the new data collected which will enhance the predictive prowess of the deployed model.

4 Results

For all models in this paper, as mentioned, Scikit Learn's GridSearchCV with a 10-fold cross validation is utilized to identify the optimal parameters and hyperparameters for the defined machine failure problem. The optimal parameters for LR is a `'C'` value of 31.62, an L1 `'penalty'`, and the `'solver'` was set to *liblinear*. Through these user-defined parameters, the LR model was able to classify whether there was a machine failure or not by a 97.25% accuracy on the test set. The optimal parameters for DT is identified to be the `'criterion'` *gini* with a `'max_depth'` of 9 levels within the tree. The optimized DT scored a model accuracy of 98.3%. The optimal parameters of the RF model are a `'max_depth` of 15. The `'max_features'` to select for each split used the square root ('sqrt') of all available feature variables. With the `'min_sample_leaf'` at 1, 'min_sample_split' at 3, and `'n_estimators'`) at 16. This gave the RF an accuracy score of 98.2%. Finally the XGB parameters included `'colsample_bytree'` at 1, the `'gamma'` at 1, the `'learning_rate'` at 0.1, `'max_depth'` of each tree at 10, number of trees (`'n_estimators'`) grown at 17 and the `'sub_sample'` at 0.8. These setting gave the XGB an accuracy of 98.4%.

The models at first glance can be considered at performing well. However, as thoroughly stated before, the class imbalance has a substantial effect on the the accuracy score, considering that the model can already score a 96.61% accuracy if it classifies just all the non-failure instances correctly. The confusion matrix along with the classification report will provide some pivotal insights into the each model's performance and behavior.

When analyzing the confusion matrix, it becomes clear that the accuracy score is indeed misleading. All models performed excellently at classifying the TPs, which are `No Failure`. In fact, the LR model was outstanding in classifying all but one of the TPs correctly. It is only when looking at the `Failure` class that clarity is provided. Of a total of 68 failed instances in the holdout subsample, the LR model only achieved 14 TNs, classifying only 14 observations correctly as a `Failure`. This suggests that the statistical learning model focused its fit through the `No Failure` data points. Although the DT, RF and XGB model all had minimal errors in classifying the TPs, they exceedingly outperformed the LR in achieving TNs. In this evaluation, XGB predicted the most `Failure` observations correct with 46 out of a possible 68, with DT classifying 45 correctly and RF classifying 43. Out of the values of each confusion matrix, a classification report

can be produced.

Table 1: Classification Report

		precision	recall	f1-score	support
Model	index				
Decision Tree	No Failure	0.988169	0.994306	0.991228	1932.0000
	Failure	0.803571	0.661765	0.725806	68.0000
	accuracy	0.983000	0.983000	0.983000	0.9830
Logistic Regression	No Failure	0.972796	0.999482	0.985959	1932.0000
	Failure	0.933333	0.205882	0.337349	68.0000
	accuracy	0.972500	0.972500	0.972500	0.97250
Random Forest	No Failure	0.987153	0.994306	0.990717	1932.0000
	Failure	0.796296	0.632353	0.704918	68.0000
	accuracy	0.982000	0.982000	0.982000	0.9820
XGBoost	No Failure	0.988683	0.994824	0.991744	1932.0000
	Failure	0.821429	0.676471	0.741935	68.0000
	accuracy	0.984000	0.984000	0.984000	0.98400

The summary statistics in the classification report further confirm the results of the confusion matrix. By examining each model's `Failure` row in the `f1-score` column, the proportional average of precision and recall from the XGB model scores the highest at 0.676, whilst LR scored the lowest at 0.206. Both `f1-score`s of the RF and DT models also performed subpar compared to DT at 0.632 and 0.662, respectively.

The final evaluation technique for classification that can be examined is the AUC-ROC, starting with the ROC-Plot. As shown below, XGB has visually the best trade-off of the sensitivity and specificity. This is followed by RF and then DT. The worst performing model in terms of that trade-off is LR.

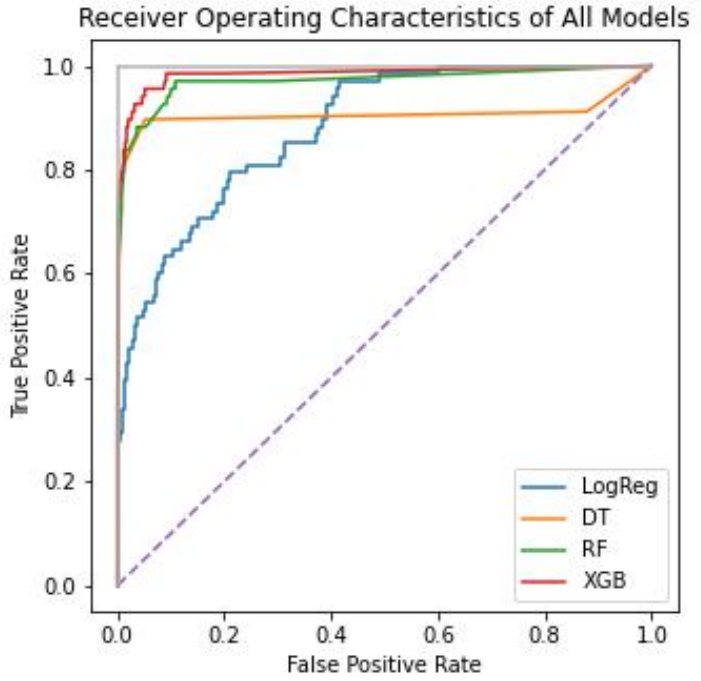


Figure 3: ROC graph of All Models

To provide summary statistics of the graph, the AUC-ROC of each model can be examined. The AUC-ROC confirms the graph's visual representation of each model's performance. XGB scored the highest at 0.986 out of a possible 1.0, RF ranks second with 0.928, with LR last at 0.903 and DT at 0.799, .

5 Discussion

By interpreting the overall evaluation results of the classification models, concurring to empirical evidence, XGB outperformed RF and DT. The researcher conjectures that XGB is a suitable choice for dataset at hand, as this still relatively straight-forward, yet powerful model is accompanied by observations that are simple and minimal, especially the proportion of data points classed as `Failure`, along with a feasible number of feature variables to predict the binary outcome. The recall, or proportion of correctly labeled `Failure` observations in the holdout group, along with the f1-score, are deemed the most important metric in the classification report. Hence, XGB must be assumed the most suitable classification model, with a recall of 76.5%. Nevertheless, the AUC-ROC also favors RF to a high standar, which derives the assumption that a more complex classification

problem could select this ensemble model.

For comparison, Kirstein and Billet (2022) have also utilized the AI4I dataset to tackle its classification problem. In terms of overall accuracy the highest accuracy computed in the research, RF's 98.4%, is comparable to both the XGB of Kirstein and the stacking classifier of Billet (2022), which is computed to be 98.6% and 95.3%, respectively. The summary statistic, AUC-ROC, can also compete, where as the research at hand produced an XGB AUC-ROC of 0.986, whilst Kirstein produced an AUC-ROC of 0.964% for the XGB model, and Billet (2022), an AUC-ROC score of 0.99.

However, the pivotal summary statistics are found in each model's ability to predict the significantly smaller class of `failures`. By first examining the F1-score, a more reliable summary statistic for imbalanced classes, it becomes apparent that the research conducted in this paper does not produce as good of a harmonious balance between percision and recall compared to the F1-score of Kirstein and Billet (2022). While this research produced an XGB F1-score of 86.7%, Kirstein produced an F1-score of 98.47% with the gradient boosting classifier and Billet (2022) had 95.3% with the stacking classifier. This becomes even clearer in the unweighted average, or macro average, of recall amongst all classification reports. The XGB in this paper scored a recall of 83.6%, while Kirstein and Billet (2022) scored a 98.6% and 95.4% respectively.

There are several reasons to why the models of Kirstein and Billet (2022) performed better with imbalanced datasets. Although it is difficulte to compare with Billet (2022), as a myriad of models were implemented for the stacking classifier, an indicator could have been in the pre-processing stage of the dataset where undersampling was used to tackle the problem of the imbalanced classes. The models of Kirstein are more comparable to the models of this paper, as the same tree-based models were utilized for classifying. Though little insight can be given on the selection of parameters, the differences are apparent. A significant example is for example, that the RF in this paper grew 16 trees, whilst the RF of Kirstein grew 100 trees. During the data preparation, what Kirstein had also implemented what this paper had not, was the implementation of a simple normalization technique that scaled the feature variables. However, in academia, the relevant classification report metrics in combination with the AUC-ROC do create acceptable results in relation to some of the empirical research described in the previous chapter (Canizo et al., 2017; Kulkarni et al., 2018; Udo and Muhammad, 2021).

6 Conclusion and Recommendations

The provide an answer to the research question, "To what extent is the implementation of tree-based models effective and feasible in PdM under industry 4.0?", the effectiveness of tree-based models vary, depending on the complexity of the PdM problem at hand. It can however be concluded that with the right procedure of research and evaluation, along with the continuous monitoring and maintenance of the selected model, either one of the three researched tree-based models have to potential to be effective and perform at an acceptable level. The results imply for a certainty, that with the proper analysis of the models' results, old practices like reactive maintenance and base-learner models will be outperformed and will predict failures to an acceptable extent, deeming these models worthy assuming the willingness to invest is present. Under the circumstances of this research, the most probable model to deploy would have been the XGB despite its simplicity compared to its peers. Whether deployment becomes the fruition will depend on further managerial cost-benefit analyses, as initial investments and maintenance costs vary highly in the manufacturing, logistics and transportation industries.

Recommendations for future research would be to firstly, implement the same classification models onto the new datasets from different machinery to gain a deeper understanding into the versatility of such model. Secondly, to broaden the business relevance of the topic at hand, the integration of such models with financial tools could provide even more feasibility in making managerial decisions regarding the investment into PdM through tree-based models.

Bibliography

S. S. Azmi and S. Baliga. An overview of boosting decision tree algorithms utilizing adaboost and xgboost boosting strategies. *International Research Journal of Engineering and Technology*, 2020. ISSN 2395-0072.

P. Billet. Predictive maintenance, stacking and importance. 2022. URL https://www.kaggle.com/code/philippebillet/predictive-maintenance-stacking-and-importance.

A. Binding, N. Dykeman, and S. Pang. Machine learning predictive maintenance on data in the wild. 2019. doi: 10.1109/WF-IoT.2019.8767312.

M. Calabrese, M. Cimmino, F. Fiume, M. Manfrin, L. Romeo, S. Ceccacci, M. Paolanti, G. Toscano, G. Ciandrini, A. Carrotta, M. Mengoni, E. Frontoni, and D. Kapetis. Sophia: An event-based iot and machine learning architecture for predictive maintenance in industry 4.0. *Information (Switzerland)*, 11, 2020. ISSN 20782489. doi: 10.3390/INFO11040202.

M. Canizo, E. Onieva, A. Conde, S. Charramendieta, and S. Trujillo. Real-time predictive maintenance for wind turbines using big data frameworks. 2017. doi: 10.1109/ICPHM.2017.7998308.

T. P. Carvalho, F. A. Soares, R. Vita, R. da P. Francisco, J. P. Basto, and S. G. Alcalá. A systematic literature review of machine learning methods applied to predictive maintenance. *Computers and Industrial Engineering*, 137, 2019. ISSN 03608352. doi: 10.1016/j.cie.2019.106024.

P. Chapman, J. Clinton, R. Kerber, T. khabaza, T. Reinartz, C. Shearer, and R. Wirth. *CRISP-DM: Step-by-step data mining guide*. NCR Systems Engineering Copenhagen and DaimlerChrysler AG and SPSS Inc. and Ohra Verzekeringen en Bank Groep B.V, USA and Denmark and Germany and The Netherlands, 1999.

T. Chen and T. He. xgboost: Extreme gradient boosting. *R Lecture*, 2014. ISSN 0371-5515.

Z. Chen, M. Wu, R. Zhao, F. Guretno, R. Yan, and X. Li. Machine remaining useful life prediction via an attention-based deep learning approach. *IEEE Transactions on Industrial Electronics*, 68, 2021. ISSN 15579948. doi: 10.1109/TIE.2020.2972443.

L. CMMS. Defining and implementing predictive maintenance with examples, Oct 2022. URL https://limblecmms.com/predictive-maintenance/.

J. Dalzochio, R. Kunst, E. Pignaton, A. Binotto, S. Sanyal, J. Favilla, and J. Barbosa. Machine learning and reasoning for predictive maintenance in industry 4.0: Current status and challenges. *Computers in Industry*, 123, 2020. ISSN 01663615. doi: 10.1016/j.compind.2020.103298.

D. Dua and C. Graff. Uci: Machine learning repository, 2017. URL http://archive.ics.uci.edu/ml.

Fiix. What is predictive maintenance?, Sep 2022. URL https://www.fiixsoftware.com/maintenance-strategies/predictive-maintenance/.

E. Florian, F. Sgarbossa, and I. Zennaro. Machine learning-based predictive maintenance: A cost-oriented model for implementation. *International Journal of Production Economics*, 236, 2021. ISSN 09255273. doi: 10.1016/j.ijpe.2021.108114.

J. Friedman, T. Hastie, S. Rosset, R. Tibshirani, and J. Zhu. Discussion of boosting papers. *Ann Stat*, 28, 2004.

J. H. Friedman. Recent advances in predictive (machine) learning. *Journal of Classification*, 23, 2006. ISSN 01764268. doi: 10.1007/s00357-006-0012-4.

H. A. Gohel, H. Upadhyay, L. Lagos, K. Cooper, and A. Sanzetenea. Predictive maintenance architecture development for nuclear infrastructure using machine learning. *Nuclear Engineering and Technology*, 52, 2020. ISSN 2234358X. doi: 10.1016/j.net.2019.12.029.

Infraspeak. Predictive maintenance: what is it, tools, and applications [complete guide 2022], 2022. URL https://blog.infraspeak.com/what-is-predictive-maintenance/#15.

G. James, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning: with Applications in R*. Springer, 2013. URL https://faculty.marshall.usc.edu/gareth-james/ISL/.

A. Jamwal, R. Agrawal, M. Sharma, and A. Giallanza. Industry 4.0 technologies for manufacturing sustainability: A systematic review and future research directions, 2021. ISSN 20763417.

E. L. Kaplan and P. Meier. Nonparametric estimation from incomplete observations. *Journal of the American Statistical Association*, 53, 1958. ISSN 1537274X. doi: 10.1080/01621459.1958.10501452.

R. Khelif, B. Chebel-Morello, S. Malinowski, E. Laajili, F. Fnaiech, and N. Zerhouni. Direct remaining useful life estimation based on support vector regression. *IEEE Transactions on Industrial Electronics*, 64, 2017. ISSN 02780046. doi: 10.1109/TIE.2016.2623260.

C. Kirstein. Predictive maintenance - milling machine 98.6%. URL https://www.kaggle.com/code/carlkirstein/predictive-maintenance-milling-machine-98-6/.

K. Kulkarni, U. Devi, A. Sirighee, J. Hazra, and P. Rao. Predictive maintenance for supermarket refrigeration systems using only case temperature data. volume 2018-June, 2018. doi: 10.23919/ACC.2018.8431901.

S. Lim and S. Chi. Xgboost application on bridge management systems for proactive damage estimation. *Advanced Engineering Informatics*, 41, 2019. ISSN 14740346. doi: 10.1016/j.aei.2019.100922.

S. Matzka. Explainable artificial intelligence for predictive maintenance applications, 2020. URL https://archive.ics.uci.edu/ml/datasets/AI4I+2020+Predictive+Maintenance+Dataset.

M. Nacchia, F. Fruggiero, A. Lambiase, and K. Bruton. A systematic mapping of the advancing use of machine learning techniques for predictive maintenance in the manufacturing sector. *Applied Sciences (Switzerland)*, 11, 2021. ISSN 20763417. doi: 10.3390/app11062546.

M. Paolanti, L. Romeo, A. Felicetti, A. Mancini, E. Frontoni, and J. Loncarski. Machine learning approach for predictive maintenance in industry 4.0. 2018. doi: 10.1109/MESA.2018.8449150.

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

F. Provost and T. Fawcett. Data science and its relationship to big data and data-driven decision making. *Big Data*, 1, 2013. ISSN 2167647X. doi: 10.1089/big.2013.1508.

H. U. Rehman, M. Asif, and M. Ahmad. Future applications and research challenges of iot. volume 2017-December, 2018. doi: 10.1109/ICICT.2017.8320166.

J. T. Rich, J. G. Neely, R. C. Paniello, C. C. Voelker, B. Nussenbaum, and E. W. Wang. A practical guide to understanding kaplan-meier curves. *Otolaryngology - Head and Neck Surgery*, 143, 2010. ISSN 01945998. doi: 10.1016/j.otohns.2010.05.007.

T. Shin. What is bootstrap sampling in machine learning and why is it important? *Medium*, Jul 2020.

Smart-Vision-Europe. What is the crisp-dm methodology?, 2015. URL `https://www.sv-europe.com/crisp-dm-methodology/`.

M. Taddy. *Business Data Science: Combining Machine Learning and Economics to Optimize, Automate, and Accelerate Business Decisions*. McGraw-Hill Education, 2019.

F. Tao, Y. Cheng, L. D. Xu, L. Zhang, and B. H. Li. Cciot-cmfg: Cloud computing and internet of things-based cloud manufacturing service system. *IEEE Transactions on Industrial Informatics*, 10, 2014. ISSN 15513203. doi: 10.1109/TII.2014.2306383.

W. Udo and Y. Muhammad. Data-driven predictive maintenance of wind turbine based on scada data. *IEEE Access*, 9, 2021. ISSN 21693536. doi: 10.1109/ACCESS.2021.3132684.

R. van Dinter, B. Tekinerdogan, and C. Catal. Predictive maintenance using digital twins: A systematic literature review. *Information and Software Technology*, 151:107008, 2022.

ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2022.107008. URL https://www.sciencedirect.com/science/article/pii/S0950584922001331.

Y. Wen, M. F. Rahman, H. Xu, and T. L. B. Tseng. Recent advances and trends of predictive maintenance from data-driven machine prognostics perspective. *Measurement: Journal of the International Measurement Confederation*, 187, 2022. ISSN 02632241. doi: 10.1016/j.measurement.2021.110276.

D. Wu, C. Jennings, J. Terpenny, R. X. Gao, and S. Kumara. A comparative study on machine learning algorithms for smart manufacturing: Tool wear prediction using random forests. *Journal of Manufacturing Science and Engineering, Transactions of the ASME*, 139, 2017. ISSN 15288935. doi: 10.1115/1.4036350.

C. Zhang, Y. Chen, H. Chen, and D. Chong. Industry 4.0 and its implementation: a review. *Information Systems Frontiers*, 2021. ISSN 15729419. doi: 10.1007/s10796-021-10153-5.

T. Zheng, M. Ardolino, A. Bacchetti, and M. Perona. The applications of industry 4.0 technologies in manufacturing context: a systematic literature review. *International Journal of Production Research*, 59, 2021. ISSN 1366588X. doi: 10.1080/00207543.2020.1824085.

R. Y. Zhong, X. Xu, E. Klotz, and S. T. Newman. Intelligent manufacturing in the context of industry 4.0: A review. *Engineering*, 3, 2017. ISSN 20958099. doi: 10.1016/J.ENG.2017.05.015.

Z. M. Çinar, A. A. Nuhu, Q. Zeeshan, O. Korhan, M. Asmael, and B. Safaei. Machine learning in predictive maintenance towards sustainable smart manufacturing in industry 4.0. *Sustainability (Switzerland)*, 12, 2020. ISSN 20711050. doi: 10.3390/su12198211.

A Code

```
#import relevant modules
import os
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

Data Exploration

```
ai = pd.read_csv('/content/drive/MyDrive/Dissertation/ai4i2020_dataset/ai4i2020.csv')
```

```
ai.shape
```

```
(10000, 14)
```

```
ai.head()
```

	UDI	Product ID	Type	Air temperature [K]	Process temperature [K]	Rotational speed [rpm]	Torque [Nm]	Tool wear [min]	Machine failure
0	1	M14860	M	298.1	308.6	1551	42.8	0	0
1	2	L47181	L	298.2	308.7	1408	46.3	3	0
2	3	L47182	L	298.1	308.5	1498	49.4	5	0
3	4	L47183	L	298.2	308.6	1433	39.5	7	0

```
ai.tail()
```

	UDI	Product ID	Type	Air temperature [K]	Process temperature [K]	Rotational speed [rpm]	Torque [Nm]	Tool wear [min]	Ma fa
9995	9996	M24855	M	298.8	308.4	1604	29.5	14	
9996	9997	H39410	H	298.9	308.4	1632	31.8	17	
9997	9998	M24857	M	299.0	308.6	1645	33.4	22	
9998	9999	H39412	H	299.0	308.7	1408	48.5	25	

```
ai.dtypes
```

```
UDI                        int64
Product ID                object
Type                      object
Air temperature [K]      float64
Process temperature [K]  float64
Rotational speed [rpm]     int64
Torque [Nm]              float64
Tool wear [min]            int64
Machine failure            int64
TWF                        int64
HDF                        int64
PWF                        int64
OSF                        int64
RNF                        int64
dtype: object
```

```
ai.describe()
```

```
ai.describe()
```

	UDI	Air temperature [K]	Process temperature [K]	Rotational speed [rpm]	Torque [Nm]	Tool wear [min]	Machine failure	
count	10000.00000	10000.000000	10000.000000	10000.000000	10000.000000	10000.000000	10000.000000	10000.000
mean	5000.50000	300.004930	310.005560	1538.776100	39.986910	107.951000	0.033900	0.004
std	2886.89568	2.000259	1.483734	179.284096	9.968934	63.654147	0.180981	0.067
min	1.00000	295.300000	305.700000	1168.000000	3.800000	0.000000	0.000000	0.000
25%	2500.75000	298.300000	308.800000	1423.000000	33.200000	53.000000	0.000000	0.000
50%	5000.50000	300.100000	310.100000	1503.000000	40.100000	108.000000	0.000000	0.000
75%	7500.25000	301.500000	311.100000	1612.000000	46.800000	162.000000	0.000000	0.000

```
ai.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 14 columns):
 #   Column                   Non-Null Count  Dtype  
---  ------                   --------------  -----  
 0   UDI                      10000 non-null  int64  
 1   Product ID               10000 non-null  object 
 2   Type                     10000 non-null  object 
 3   Air temperature [K]      10000 non-null  float64
 4   Process temperature [K]  10000 non-null  float64
 5   Rotational speed [rpm]   10000 non-null  int64  
 6   Torque [Nm]              10000 non-null  float64
 7   Tool wear [min]          10000 non-null  int64  
 8   Machine failure          10000 non-null  int64  
 9   TWF                      10000 non-null  int64  
 10  HDF                      10000 non-null  int64  
 11  PWF                      10000 non-null  int64  
 12  OSF                      10000 non-null  int64  
 13  RNF                      10000 non-null  int64  
dtypes: float64(3), int64(9), object(2)
memory usage: 1.1+ MB
```

```
ai.isna().sum()
```

```
UDI                        0
Product ID                 0
Type                       0
Air temperature [K]        0
Process temperature [K]    0
Rotational speed [rpm]     0
Torque [Nm]                0
Tool wear [min]            0
Machine failure            0
TWF                        0
HDF                        0
PWF                        0
OSF                        0
RNF                        0
dtype: int64
```

```
print(ai['Machine failure'].value_counts())
```

```
0    9661
1     339
Name: Machine failure, dtype: int64
```

```
ai.nunique()
```

```
UDI                        10000
Product ID                 10000
```

```
Type                          3
Air temperature [K]          93
Process temperature [K]      82
Rotational speed [rpm]      941
Torque [Nm]                 577
Tool wear [min]             246
Machine failure               2
TWF                           2
HDF                           2
PWF                           2
OSF                           2
RNF                           2
dtype: int64
```

```
ai.corr()
```

	UDI	Air temperature [K]	Process temperature [K]	Rotational speed [rpm]	Torque [Nm]	Tool wear [min]	Machine failure	TWF	HDF
UDI	1.000000	0.117428	0.324428	-0.006615	0.003207	-0.010702	-0.022892	0.009154	-0.022215
Air temperature [K]	0.117428	1.000000	0.876107	0.022670	-0.013778	0.013853	0.082556	0.009955	0.137831
Process temperature [K]	0.324428	0.876107	1.000000	0.019277	-0.014061	0.013488	0.035946	0.007315	0.056933
Rotational speed [rpm]	-0.006615	0.022670	0.019277	1.000000	-0.875027	0.000223	-0.044188	0.010389	-0.121241
Torque [Nm]	0.003207	-0.013778	-0.014061	-0.875027	1.000000	-0.003093	0.191321	-0.014662	0.142610
Tool wear [min]	-0.010702	0.013853	0.013488	0.000223	-0.003093	1.000000	0.105448	0.115792	-0.001287
Machine	-0.022892	0.082556	0.035946	-0.044188	0.191321	0.105448	1.000000	0.362904	0.575800

```
#remove the Unique ID, product ID. failure types from the dataset, they cannot be predictor variables
ai2 = ai.drop(columns=['UDI', 'Product ID', 'TWF', 'HDF', 'PWF', 'OSF', 'RNF'], axis=1)

#Change the 'type' column into dummies
ai2 = pd.get_dummies(ai2, columns=['Type'], drop_first=True) #dropping the whole type H column because if L and M a

#For some reason XGBoost cant have square brackets in the column names...
ai2.rename(columns = {'Air temperature [K]':'Air temperature', 'Process temperature [K]':'Process temperature',
                      'Rotational speed [rpm]':'Rotational speed', 'Torque [Nm]':'Torque', 'Tool wear [min]
```

```
ai2.head()
```

	Air temperature	Process temperature	Rotational speed	Torque	Tool wear	Machine failure	Type_L	Type_M
0	298.1	308.6	1551	42.8	0	0	0	1
1	298.2	308.7	1408	46.3	3	0	1	0
2	298.1	308.5	1498	49.4	5	0	1	0
3	298.2	308.6	1433	39.5	7	0	1	0
4	298.2	308.7	1408	40.0	9	0	1	0

```
#explore with seaborn pairplot

sns.pairplot(ai2, hue='Machine failure')
```

```
<seaborn.axisgrid.PairGrid at 0x7fc03f3c21f0>
```



```
#get a figure of type pairplot
ai2_types = ai2[['Type_L', 'Type_M', 'Machine failure']]
print(ai2_types)
```

```
      Type_L  Type_M  Machine failure
0          0       1                0
1          1       0                0
2          1       0                0
3          1       0                0
4          1       0                0
...      ...     ...              ...
9995       0       1                0
9996       0       0                0
9997       0       1                0
9998       0       0                0
9999       0       1                0

[10000 rows x 3 columns]
```

```
#the type columns might be disproportionate according the the plot. lets check this out.
ai2_failure_sum = ai2[ai2['Machine failure'] == 1]
print(ai2_failure_sum['Type_L'].value_counts())
print(ai2_failure_sum['Type_M'].value_counts())

#this confirms that a lot of the failed machines are type 1 which creates an imbalance as the tree models may favor
```

```
1    235
0    104
Name: Type_L, dtype: int64
0    256
1     83
Name: Type_M, dtype: int64
```

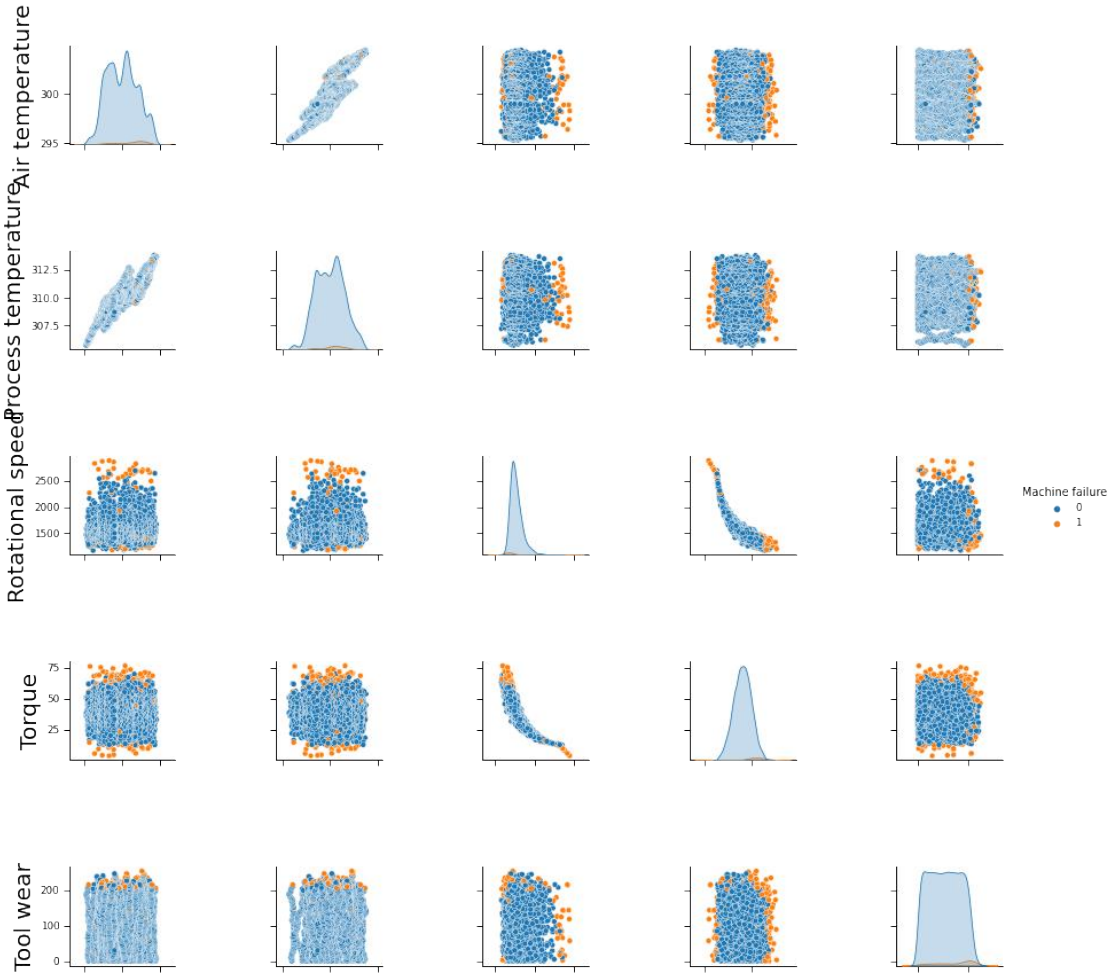
```
#lets remove the binary type columns!
ai3 = ai2.drop(columns=['Type_L', 'Type_M'], axis=1)
```

```
ai3.head()
```

	Air temperature	Process temperature	Rotational speed	Torque	Tool wear	Machine failure
0	298.1	308.6	1551	42.8	0	0
1	298.2	308.7	1408	46.3	3	0
2	298.1	308.5	1498	49.4	5	0
3	298.2	308.6	1433	39.5	7	0
4	298.2	308.7	1408	40.0	9	0

```
#explore with seaborn pairplot with types dropped

sns.pairplot(ai3, hue='Machine failure')
#plt.savefig('pairplot.png')
```

```
<seaborn.axisgrid.PairGrid at 0x7f9d95940730>
```




```
ai3.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 6 columns):
 #   Column               Non-Null Count  Dtype  
---  ------               --------------  -----  
 0   Air temperature      10000 non-null  float64
 1   Process temperature  10000 non-null  float64
 2   Rotational speed     10000 non-null  int64  
 3   Torque               10000 non-null  float64
 4   Tool wear            10000 non-null  int64  
 5   Machine failure      10000 non-null  int64  
dtypes: float64(3), int64(3)
memory usage: 468.9 KB
```

```
#separte the predictor variables from the target variable
predictors = ai3.drop(columns=['Machine failure'])
target = ai3['Machine failure']
```

```
target.value_counts()
```

```
0    9661
1     339
Name: Machine failure, dtype: int64
```

```
#split the data into train and test
#import train test split module
from sklearn.model_selection import train_test_split
```

```
#define object to specify a 80/20 split
test_size = 0.2

#initiate split
X_train, X_test, y_train, y_test = train_test_split(predictors, target, test_size=test_size, shuffle=True, stratify
```

```
#just to check to splits
print(y_train.value_counts(),"\n",
      y_test.value_counts())
```

```
0    7729
1     271
Name: Machine failure, dtype: int64
 0    1932
1      68
Name: Machine failure, dtype: int64
```

Logistic Regression

```
#For logistic regression, standardize the data. Easily done with sklearn's standardscaler
#from sklearn import preprocessing

#scaler = preprocessing.StandardScaler()
#X_train_scaler = scaler.fit_transform(X_train)
#X_test_scaler = scaler.fit_transform(X_test)
```

```
#import logreg package and gridsearch
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV

#set the parameters to hypertune
#import numpy to use logspace, that returns a specified range in log scale
import numpy as np

logreg_params = {'penalty' : ['l1','l2', 'none', 'elasticnet'], 'C': np.logspace(-3,3,5),
    'solver' : ['newton-cg', 'lbfgs', 'liblinear']}

#initiate model
logreg = LogisticRegression()
logreg_grid = GridSearchCV(logreg, param_grid=logreg_params, scoring='accuracy', cv=10)
```

```
#fit the logreg model to the training data now
logreg_grid.fit(X_train, y_train)
```

```
/usr/local/lib/python3.8/dist-packages/scipy/optimize/linesearch.py:478: LineSearchWarning: The line search
  warn('The line search algorithm did not converge', LineSearchWarning)
/usr/local/lib/python3.8/dist-packages/scipy/optimize/linesearch.py:327: LineSearchWarning: The line search
  warn('The line search algorithm did not converge', LineSearchWarning)
/usr/local/lib/python3.8/dist-packages/scipy/optimize/linesearch.py:478: LineSearchWarning: The line search
  warn('The line search algorithm did not converge', LineSearchWarning)
/usr/local/lib/python3.8/dist-packages/scipy/optimize/linesearch.py:327: LineSearchWarning: The line search
  warn('The line search algorithm did not converge', LineSearchWarning)
/usr/local/lib/python3.8/dist-packages/scipy/optimize/linesearch.py:478: LineSearchWarning: The line search
  warn('The line search algorithm did not converge', LineSearchWarning)
/usr/local/lib/python3.8/dist-packages/scipy/optimize/linesearch.py:327: LineSearchWarning: The line search
  warn('The line search algorithm did not converge', LineSearchWarning)
/usr/local/lib/python3.8/dist-packages/scipy/optimize/linesearch.py:478: LineSearchWarning: The line search
  warn('The line search algorithm did not converge', LineSearchWarning)
/usr/local/lib/python3.8/dist-packages/scipy/optimize/linesearch.py:327: LineSearchWarning: The line search
  warn('The line search algorithm did not converge', LineSearchWarning)
/usr/local/lib/python3.8/dist-packages/scipy/optimize/linesearch.py:478: LineSearchWarning: The line search
  warn('The line search algorithm did not converge', LineSearchWarning)
/usr/local/lib/python3.8/dist-packages/scipy/optimize/linesearch.py:327: LineSearchWarning: The line search
  warn('The line search algorithm did not converge', LineSearchWarning)
```

```
/usr/local/lib/python3.8/dist-packages/scipy/optimize/linesearch.py:478: LineSearchWarning: The line search
  warn('The line search algorithm did not converge', LineSearchWarning)
/usr/local/lib/python3.8/dist-packages/scipy/optimize/linesearch.py:327: LineSearchWarning: The line search
  warn('The line search algorithm did not converge', LineSearchWarning)
/usr/local/lib/python3.8/dist-packages/scipy/optimize/linesearch.py:478: LineSearchWarning: The line search
  warn('The line search algorithm did not converge', LineSearchWarning)
/usr/local/lib/python3.8/dist-packages/scipy/optimize/linesearch.py:327: LineSearchWarning: The line search
  warn('The line search algorithm did not converge', LineSearchWarning)
/usr/local/lib/python3.8/dist-packages/scipy/optimize/linesearch.py:478: LineSearchWarning: The line search
  warn('The line search algorithm did not converge', LineSearchWarning)
/usr/local/lib/python3.8/dist-packages/scipy/optimize/linesearch.py:327: LineSearchWarning: The line search
  warn('The line search algorithm did not converge', LineSearchWarning)
/usr/local/lib/python3.8/dist-packages/scipy/optimize/linesearch.py:478: LineSearchWarning: The line search
  warn('The line search algorithm did not converge', LineSearchWarning)
/usr/local/lib/python3.8/dist-packages/scipy/optimize/linesearch.py:327: LineSearchWarning: The line search
  warn('The line search algorithm did not converge', LineSearchWarning)
/usr/local/lib/python3.8/dist-packages/scipy/optimize/linesearch.py:478: LineSearchWarning: The line search
  warn('The line search algorithm did not converge', LineSearchWarning)
/usr/local/lib/python3.8/dist-packages/scipy/optimize/linesearch.py:327: LineSearchWarning: The line search
  warn('The line search algorithm did not converge', LineSearchWarning)
/usr/local/lib/python3.8/dist-packages/scipy/optimize/linesearch.py:478: LineSearchWarning: The line search
  warn('The line search algorithm did not converge', LineSearchWarning)
/usr/local/lib/python3.8/dist-packages/scipy/optimize/linesearch.py:327: LineSearchWarning: The line search
  warn('The line search algorithm did not converge', LineSearchWarning)
/usr/local/lib/python3.8/dist-packages/scipy/optimize/linesearch.py:478: LineSearchWarning: The line search
  warn('The line search algorithm did not converge', LineSearchWarning)
/usr/local/lib/python3.8/dist-packages/scipy/optimize/linesearch.py:327: LineSearchWarning: The line search
  warn('The line search algorithm did not converge', LineSearchWarning)
/usr/local/lib/python3.8/dist-packages/scipy/optimize/linesearch.py:478: LineSearchWarning: The line search
  warn('The line search algorithm did not converge', LineSearchWarning)
/usr/local/lib/python3.8/dist-packages/scipy/optimize/linesearch.py:327: LineSearchWarning: The line search
  warn('The line search algorithm did not converge', LineSearchWarning)
/usr/local/lib/python3.8/dist-packages/scipy/optimize/linesearch.py:478: LineSearchWarning: The line search
  warn('The line search algorithm did not converge', LineSearchWarning)
/usr/local/lib/python3.8/dist-packages/scipy/optimize/linesearch.py:327: LineSearchWarning: The line search
```

```
print("Best Hyperparameters for logreg model:", logreg_grid.best_params_)
print("Accuracy :", logreg_grid.best_score_)
```

```
Best Hyperparameters for logreg model: {'C': 31.622776601683793, 'penalty': 'l1', 'solver': 'liblinear'}
Accuracy : 0.9716249999999998
```

```
#now we know the optimal parameters so we fit it into the model again
logreg_opt = LogisticRegression(C=31.622776601683793, penalty='l1', solver='liblinear')
logreg_opt.fit(X_train, y_train) #now we check how well it performs to the outcome for training

logreg_y_pred = logreg_opt.predict(X_test) #now that it has learnt, we give it the unseen dataset to predict on
print("Model Accuracy is: ", logreg_opt.score(X_test, y_test))
```

```
Model Accuracy is:  0.9725
/usr/local/lib/python3.8/dist-packages/sklearn/svm/_base.py:1206: ConvergenceWarning: Liblinear failed to con
  warnings.warn(
```

Logistic Regression Evaluation

Classification Report

```
from sklearn.metrics import classification_report

logreg_classreport = classification_report(y_test, logreg_y_pred)
print(logreg_classreport)
```

```
              precision    recall  f1-score   support

           0       0.97      1.00      0.99      1932
           1       0.93      0.21      0.34        68

    accuracy                           0.97      2000
   macro avg       0.95      0.60      0.66      2000
```

```
weighted avg       0.97      0.97      0.96      2000
```

Confusion Matrix

```
from sklearn.metrics import confusion_matrix

logreg_matrix = confusion_matrix(y_test, logreg_y_pred)
print(logreg_matrix)
```

```
[[1931    1]
 [  54   14]]
```

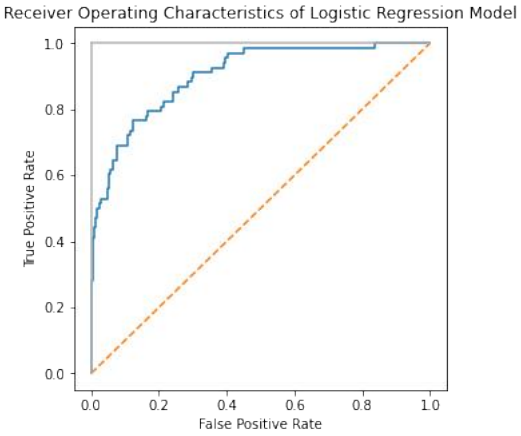
AUC-ROC

```
from sklearn.metrics import roc_curve, roc_auc_score

#return the probability estimates of the predictions
logreg_y_proba = logreg_opt.predict_proba(X_test)[:,1]

#create false positive rate, true positive rate, threshold with roc-curve function
logreg_fpr, logreg_tpr, logreg_threshold = roc_curve(y_test, logreg_y_proba)

print('roc_auc_score for Logistic Regression is: ', roc_auc_score(y_test, logreg_y_proba))
```

```
roc_auc_score for Logistic Regression is:  0.9025621118012422
```

Plotting AUC ROC

```
import matplotlib.pyplot as plt

plt.subplots(1, figsize=(5,5))
plt.title('Receiver Operating Characteristics of Logistic Regression Model')
plt.plot(logreg_fpr, logreg_tpr)
plt.plot([0, 1], ls="--")
plt.plot([0, 0], [1, 0], c=".7"), plt.plot([1, 1], c=".7")
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```



Decision Tree

```
from sklearn.tree import DecisionTreeClassifier

dt_params = {'criterion' : ['gini','entropy'], 'max_depth': np.arange(3,21,3)
    }
```

```
dt = DecisionTreeClassifier()
dt_grid = GridSearchCV(dt, param_grid=dt_params, scoring='accuracy', cv=10)
```

```
dt_grid.fit(X_train, y_train)
```

```
GridSearchCV(cv=10, estimator=DecisionTreeClassifier(),
             param_grid={'criterion': ['gini', 'entropy'],
                         'max_depth': array([ 3,  6,  9, 12, 15, 18])},
             scoring='accuracy')
```

```
print("Best Hyperparameters for Decision Tree model:", dt_grid.best_params_)
print("Accuracy :", dt_grid.best_score_)
```

```
Best Hyperparameters for Decision Tree model: {'criterion': 'gini', 'max_depth': 9}
Accuracy : 0.9828750000000002
```

```
#now we know the optimal decision tree parameters so we fit it into the model again
dt_opt = DecisionTreeClassifier(criterion='gini' , max_depth=10)
dt_opt.fit(X_train, y_train)

dt_y_pred = dt_opt.predict(X_test)
print("Model Accuracy is: ", dt_opt.score(X_test, y_test))
```

```
Model Accuracy is:  0.983
```

Decision Tree Evaluation

Classification Report

```
#from sklearn.metrics import classification_report

dt_classreport = classification_report(y_test, dt_y_pred)
print(dt_classreport)
```

```
                  precision    recall  f1-score   support

               0       0.99      0.99      0.99      1932
               1       0.80      0.66      0.73        68

        accuracy                           0.98      2000
       macro avg       0.90      0.83      0.86      2000
    weighted avg       0.98      0.98      0.98      2000
```

Confusion Matrix

```
#from sklearn.metrics import confusion_matrix

dt_matrix = confusion_matrix(y_test, dt_y_pred)
print(dt_matrix)
```

```
[[1921   11]
 [  23   45]]
```

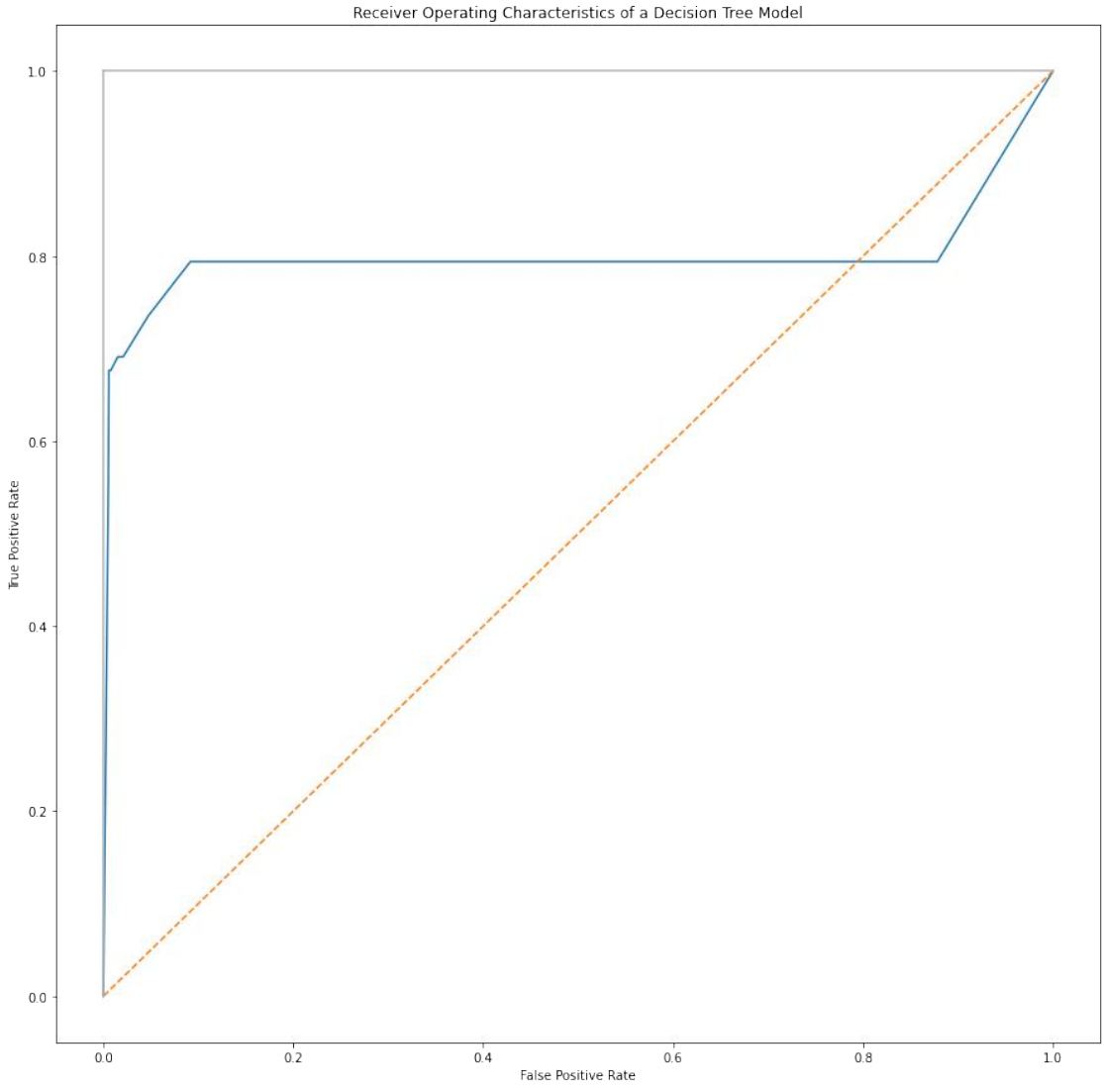
AUC ROC

```
#from sklearn.metrics import roc_curve, roc_auc_score

#return the probability estimates of the predictions
dt_y_proba = dt_opt.predict_proba(X_test)[:,1]

#create false positive rate, true positive rate, threshold with roc-curve function
dt_fpr, dt_tpr, dt_threshold = roc_curve(y_test, dt_y_proba)

print('roc_auc_score for the Decision Tree is: ', roc_auc_score(y_test, dt_y_proba))
```

```
roc_auc_score for the Decision Tree is:  0.7988064791133845
```

Plotting AUC ROC

```
#import matplotlib.pyplot as plt

plt.subplots(1, figsize=(15,15))
plt.title('Receiver Operating Characteristics of a Decision Tree Model')
plt.plot(dt_fpr, dt_tpr)
plt.plot([0, 1], ls="--")
plt.plot([0, 0], [1, 0], c=".7"), plt.plot([1, 1], c=".7")
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```



Random Forest

```
from sklearn.ensemble import RandomForestClassifier

rf_params = {'n_estimators' : np.arange(2, 20, 2), 'max_features': ['auto', 'sqrt'],
    'max_depth' : np.arange(3, 21, 3), 'min_samples_split': np.arange(2,5,1), 'min_samples_leaf': np.arange(1,3,1)

rf = RandomForestClassifier()
rf_grid = GridSearchCV(rf, param_grid=rf_params, scoring='accuracy', cv=10)
```

```
rf_grid.fit(X_train, y_train)
```

```
GridSearchCV(cv=10, estimator=RandomForestClassifier(),
             param_grid={'max_depth': array([ 3,  6,  9, 12, 15, 18]),
                         'max_features': ['auto', 'sqrt'],
                         'min_samples_leaf': array([1, 2]),
                         'min_samples_split': array([2, 3, 4]),
                         'n_estimators': array([ 2,  4,  6,  8, 10, 12, 14, 16, 18])},
             scoring='accuracy')
```

```
print("Best Hyperparameters for the Random Forest model:", rf_grid.best_params_)
print("Accuracy :", rf_grid.best_score_)
```

```
Best Hyperparameters for the Random Forest model: {'max_depth': 15, 'max_features': 'sqrt', 'min_samples_leaf
Accuracy : 0.9827499999999999
```

```
#now we know the optimal random forest parameters so we fit it into the model again
rf_opt = RandomForestClassifier(max_depth=15, max_features='sqrt', min_samples_leaf=1, min_samples_split=3, n_estim
                                )
rf_opt.fit(X_train, y_train)

rf_y_pred = rf_opt.predict(X_test)
print("Model Accuracy is: ", rf_opt.score(X_test, y_test))
```

```
Model Accuracy is:  0.982
```

Random Forest Evaluation

[ ] ↳ *8 cells hidden*

XGBoost

```
from xgboost import XGBClassifier

#gotta reduce the params, took 2 hours...

xgb_params = {'n_estimators' : np.arange(2, 20, 5), 'learning_rate': [0.001, 0.01, 0.1],
    'max_depth' : np.arange(3, 21, 7), 'subsample': [0.7, 0.8, 0.9], 'colsample_bytree': [ 0.7, 0.9, 1], 'gamma': [

xgb = XGBClassifier()
xgb_grid = GridSearchCV(xgb, param_grid=xgb_params, scoring='accuracy', cv=10)
```

```
xgb_grid.fit(X_train, y_train)
```

```
GridSearchCV(cv=10, estimator=XGBClassifier(),
             param_grid={'colsample_bytree': [0.7, 0.9, 1], 'gamma': [0, 1, 5],
                         'learning_rate': [0.001, 0.01, 0.1],
                         'max_depth': array([ 3, 10, 17]),
                         'n_estimators': array([ 2,  7, 12, 17]),
                         'subsample': [0.7, 0.8, 0.9]},
             scoring='accuracy')
```

```
print("Best Hyperparameters for the XGBoost model:", xgb_grid.best_params_)
print("Accuracy :", xgb_grid.best_score_)
```

```
Best Hyperparameters for the XGBoost model: {'colsample_bytree': 1, 'gamma': 1, 'learning_rate': 0.1, 'max_de
Accuracy : 0.986375
```

```
#now we know the optimal random forest parameters so we fit it into the model again
xgb_opt = XGBClassifier(colsample_bytree=1, gamma=1, learning_rate=0.1, max_depth=10, n_estimators=17, subsample=0.
xgb_opt.fit(X_train, y_train)

xgb_y_pred = xgb_opt.predict(X_test)
print("Model Accuracy is: ", xgb_opt.score(X_test, y_test))
```

```
Model Accuracy is:  0.984
```

XGBoost Evaluation

Classification Report

```
#from sklearn.metrics import classification_report

xgb_classreport = classification_report(y_test, xgb_y_pred)
print(xgb_classreport)
```

```
                   precision    recall  f1-score   support

               0       0.99      0.99      0.99      1932
               1       0.82      0.68      0.74        68

        accuracy                           0.98      2000
       macro avg       0.91      0.84      0.87      2000
    weighted avg       0.98      0.98      0.98      2000
```

Confusion Matrix

```
#from sklearn.metrics import confusion_matrix

xgb_matrix = confusion_matrix(y_test, xgb_y_pred)
print(xgb_matrix)
```

```
[[1922   10]
 [  22   46]]
```

AUC ROC

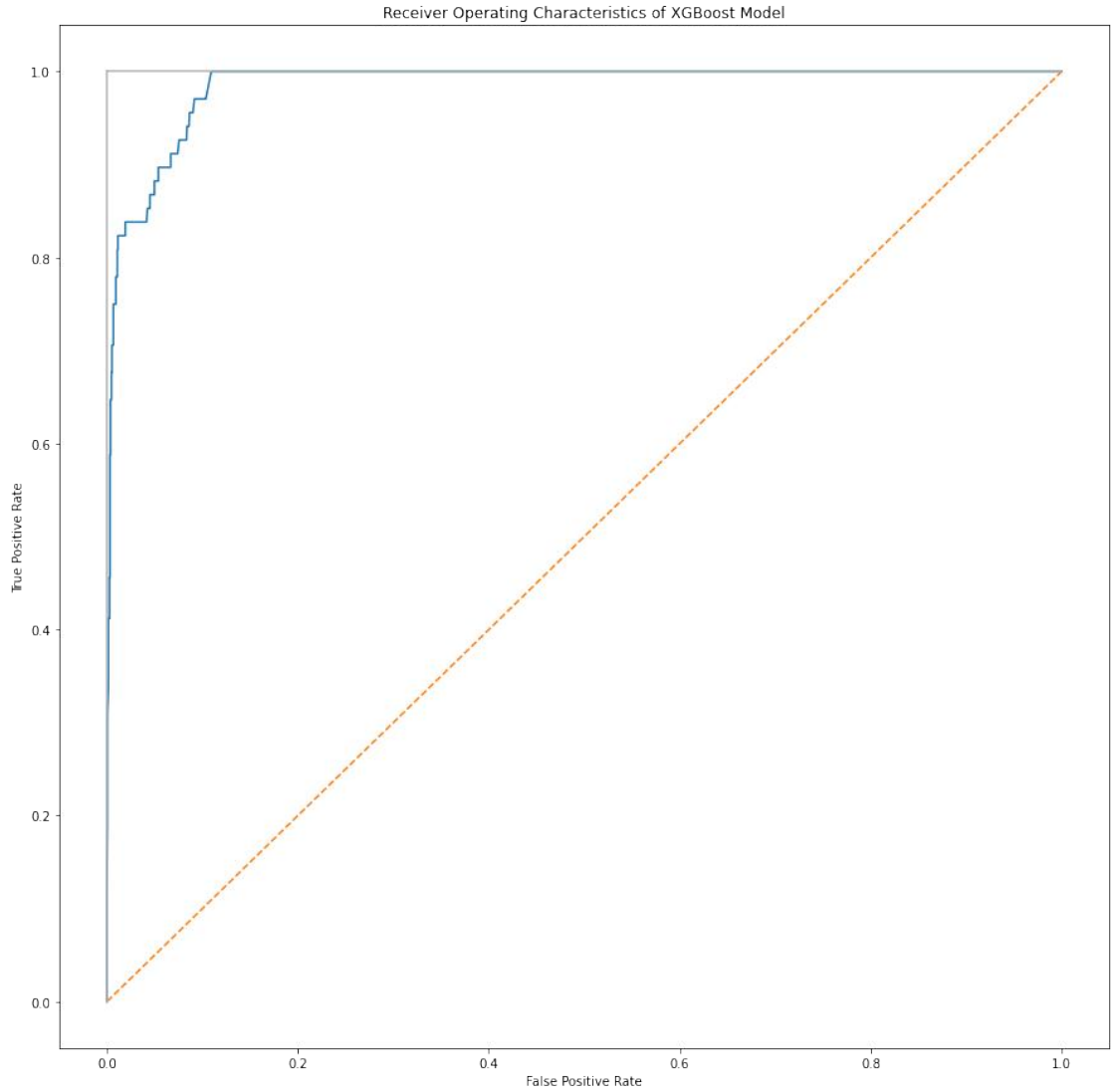
```
#from sklearn.metrics import roc_curve, roc_auc_score

#return the probability estimates of the predictions
xgb_y_proba = xgb_opt.predict_proba(X_test)[:,1]

#create false positive rate, true positive rate, threshold with roc-curve function
xgb_fpr, xgb_tpr, xgb_threshold = roc_curve(y_test, xgb_y_proba)

print('roc_auc_score for the XGBoost is: ', roc_auc_score(y_test, xgb_y_proba))
```

```
roc_auc_score for the XGBoost is:  0.9855643344294239
```

Plotting AUC ROC

```
#import matplotlib.pyplot as plt

plt.subplots(1, figsize=(15,15))
plt.title('Receiver Operating Characteristics of XGBoost Model')
plt.plot(xgb_fpr, xgb_tpr)
```

```
plt.plot([0, 1], ls="--")
plt.plot([0, 0], [1, 0], c=".7"), plt.plot([1, 1], c=".7")
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```



Figures

```
#re-initiate all classreports into dictionaries using the output_dict parameter in classification_report
logreg_classreport_dict = classification_report(y_test, logreg_y_pred, output_dict=True)
dt_classreport_dict = classification_report(y_test, dt_y_pred, output_dict=True)
rf_classreport_dict = classification_report(y_test, rf_y_pred, output_dict=True)
xgb_classreport_dict = classification_report(y_test, xgb_y_pred, output_dict=True)


#now turn them into pandas DataFrames

logreg_classreport_df = pd.DataFrame(logreg_classreport_dict).transpose()
dt_classreport_df = pd.DataFrame(dt_classreport_dict).transpose()
rf_classreport_df = pd.DataFrame(rf_classreport_dict).transpose()
xgb_classreport_df = pd.DataFrame(xgb_classreport_dict).transpose()

#add a column to each dataframe that specifies from what model it came from
logreg_classreport_df['Model'] = "Logistic Regression"
dt_classreport_df['Model'] = "Decision Tree"
```

```
rf_classreport_df['Model'] = "Random Forest"
xgb_classreport_df['Model'] = "XGBoost"



#concat all the dataframes together
all_classreport_df = pd.concat([logreg_classreport_df, dt_classreport_df, rf_classreport_df, xgb_classreport_df])



#turn the original index into a column
all_classreport_df = all_classreport_df.reset_index(level=0)

#set the index
all_classreport_df = all_classreport_df.set_index(['Model'])

#now use pivot table to turn the Model and index column into a multiindex
all_classreport_df = pd.pivot_table(all_classreport_df, index=['Model', 'index'], )

all_classreport_df = all_classreport_df[['precision', 'recall', 'f1-score', 'support']]

all_classreport_df

#now make the table  latex format

#print(all_classreport_df.to_latex(index=True))
```

Model	index	precision	recall	f1-score	support
Decision Tree	**0**	0.988169	0.994306	0.991228	1932.0000
	1	0.803571	0.661765	0.725806	68.0000
	accuracy	0.983000	0.983000	0.983000	0.9830
	macro avg	0.895870	0.828036	0.858517	2000.0000
	weighted avg	0.981892	0.983000	0.982204	2000.0000
Logistic Regression	**0**	0.972796	0.999482	0.985959	1932.0000
	1	0.933333	0.205882	0.337349	68.0000
	accuracy	0.972500	0.972500	0.972500	0.9725
	macro avg	0.953065	0.602682	0.661654	2000.0000
	weighted avg	0.971454	0.972500	0.963906	2000.0000
Random Forest	**0**	0.987153	0.994306	0.990717	1932.0000
	1	0.796296	0.632353	0.704918	68.0000
	accuracy	0.982000	0.982000	0.982000	0.9820
	macro avg	0.891725	0.813330	0.847817	2000.0000
	weighted avg	0.980664	0.982000	0.981000	2000.0000
XGBoost	**0**	0.988683	0.994824	0.991744	1932.0000
	1	0.821429	0.676471	0.741935	68.0000
	accuracy	0.984000	0.984000	0.984000	0.9840
	macro avg	0.905056	0.835647	0.866840	2000.0000
	weighted avg	0.982996	0.984000	0.983251	2000.0000

```
#Plot of all ROCs

#Create the Subplot
plt.subplots(1, figsize=(5,5))
plt.title('Receiver Operating Characteristics of All Models')
```

```
#LogReg
plt.plot(logreg_fpr, logreg_tpr, label='LogReg')


#DT
plt.plot(dt_fpr, dt_tpr, label='DT')

#RF
plt.plot(rf_fpr, rf_tpr, label='RF')


#XGB
plt.plot(xgb_fpr, xgb_tpr, label='XGB')

#formatting
plt.plot([0, 1], ls="--")
plt.plot([0, 0], [1, 0], c=".7"), plt.plot([1, 1], c=".7")
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.legend(loc='lower right')

#save figure
#plt.savefig('ROC_allmodels.png')
```