Department of Computer Science

BSc (Hons) Computer Science

Academic Year 2023 - 2024

Implementation and Evaluation of Machine Learning Models for Hard Drive Failure Prediction in Resource Constrained Devices

Zahid Sarwar

2108979

A report submitted in partial fulfilment of the requirements for the degree of Bachelor of Science

Brunel University London
Department of Computer Science
Uxbridge
Middlesex
UB8 3PH
United Kingdom
T: +44 1895 203397
F: +44 (0) 1895 251686

Abstract

This dissertation addresses the challenge of selecting the most suitable machine learning model to accurately predict hard drive failures for deployment on resource constrained devices, such as Raspberry Pi's, Le Potato AML as well as the Nvidia Jetson lineup. With the proliferation of such devices in various domains such as home labs, there is a growing demand for efficient and effective machine learning solutions tailored to their resource constraints. The primary objective of this research is to identify and implement a machine learning model that strikes a balance between computational efficiency and predictive performance.

The methodology involves training and testing multiple machine learning models on a common dataset to comprehensively evaluate their performance. Various popular algorithms, including but not limited to Decision Trees, Random Forest, Naïve Bayes, Gradient Boosting and AdaBoost are considered. Performance evaluation metrics such as accuracy, precision, recall, F1-score, and others are utilised to assess the models' effectiveness. Additionally, computational complexity and testing times are considered to ensure suitability for deployment on resource constrained devices.

Findings from the experimentation reveal significant variations in the performance of different machine learning models. While some models exhibit superior predictive accuracy and recall, others demonstrate better computational efficiency. The key challenge lies in identifying a model that offers a good trade-off between predictive performance and computational resource usage. Through extensive evaluation and comparison, a model is chosen that demonstrates the most promising performance characteristics suitable for deployment on resource constrained devices.

The conclusions drawn from this research highlight the importance of tailored machine learning solutions for resource constrained devices. By leveraging performance evaluation metrics and considering resource constraints, practitioners can make informed decisions regarding model selection and deployment strategies.

Acknowledgements

I would like to thank my tutor Dr Tahmina Zebin for providing me with a solid foundation of machine learning as well as addressing my needs without fail and taking the time out every week to advise and overlook my project. This project would not be possible without the help of my tutor, so many thanks Tahmina. I would like to extend additional thanks to my best friend Rahman, my girlfriend Ounsa, my father and my grandparents for motivating me throughout the project as well as being understanding and providing me with the space I needed to successfully complete this project.

I certify that the work presented in the dissertation is my own unless referenced.

Signature ______________ZS_____________________

Date _____________05/04/24_____________________

Total Words: 14079

Table of Contents

List of Tables

List of Figures

1 Introduction

With modern technology evolving at a rapid pace, the form factor of devices have become smaller and more powerful than ever. This expands the market, allowing more people to delve into creating home labs without needing the space and extra expenses of owning a big server. In this dissertation I am looking to solve the problem that comes with unexpected hard drive failures through machine learning, although already explored in depth by others I intend to differentiate my project by catering towards home lab users by focusing on hard drive failure prediction using machine learning for resource constrained devices such as Raspberry Pi's. My motivation for creating this solution is due to most individuals with home labs choosing not to pay for third-party cloud services but instead relying on their own server to act in place of this.

1.1 Aims and Objectives

The aim of the project is to train and deploy a machine learning model that can accurately predict hard drive failures whilst running on resource constrained devices. I will explore, analyse, and compare multiple machine learning models to find a model with a good trade-off between predictive performance and testing time to ensure the chosen model is very lightweight.

The objectives required to support the aim:

1. Explore existing research to identify trends with current machine learning models, existing datasets as well as data preprocessing techniques.
2. Find an up-to-date raw dataset containing rich information on drive statistics showcasing failures and non-failures which can be pre-processed and trained on the chosen model.
3. Evaluate each model's performance on the data by analysing metrics such as precision, recall and testing time.
4. Provide visual evidence with infographics such as matrixes, feature importance graphs and summary plots. Followed by extensive analysis, comparisons and justifications of each models ranking.
5. Develop code to preprocess the input data into an acceptable format for the model, deal with an abundance of input data as well as linking the solution to a front-end web user interface.
6. Explore deployment solutions to find the most user friendly and efficient solution for the project.

1.2 Project Approach

I intend to analyse existing literature in the field to explore trends, existing solutions and evaluate the various methodologies researchers have used to produce comparable results. Based upon this I will explore recent open-source raw datasets containing daily SMART (Self-Monitoring Analysis, and Reporting Technology) hard drive statistics along with the drives operational state highlighting failure if apparent. Preprocess the data adhering to conventional methodologies, providing explanations and justifications on the steps taken. Develop separate instances of code in Python for six different machine learning algorithms, train them on the dataset using common methodologies such as train-test split or cross-validation. Produce a collection of comparable evaluation metrics and infographics, providing evidence where conclusions can be made and justified. The best performing model will be chosen based off an analysis between the computational accuracy and inference times of all models. Further code will be developed to retrain the winning algorithm on the entire dataset, the trained model will then be serialised and saved to a file. Lastly the application's code will be developed to deserialise and import the trained model along with providing SMART data statistics from a connected drive. The SMART data will be pre-processed into an acceptable format for the model, followed by the model returning predictions. This will be connected to a front-end web based graphical user interface, where the user is able to press a button initiating the model to calculate a prediction on a pre-defined drive, returning a result whilst ensuring visibility in the front end. The training and testing stages will be performed on a desktop environment whilst the completed application will be built as a Docker image and transferred onto a Raspberry Pi 4B. The image will be rebuilt on the Pi and deployed as a Docker container offering support for Linux ARM architecture.

1.3 Dissertation Outline

- Chapter 2 delves into the previous background research for the subject area. It involves analysis of techniques. This chapter looks to solve the first objective.
- Chapter 3 explores the proposed techniques I intend to follow to produce a solution satisfactory of the aims. This tackles the aim by supporting a systematic approach for the rest of the chapters and objectives.
- Chapter 4 explains the general data formatting and preprocessing steps I plan to take to ensure the data is ready to be processed by the algorithms. It also briefly touches upon the intentions for the UI (user interface) design as well as the front end and deployment. This chapter looks to solve objective 2.
- Chapter 5 discusses in detail the process of implementing, training, and producing metrics for all the models. The process of training and exporting the best performing model and lastly the implementation and deployment of the final solution. This chapter look to solve objective 5.
- Chapter 6 presents the results acquired from performing the implementation as a part of chapter 5. These will be compared and evaluated whilst providing appropriate justifications. This chapter looks to solve a mixture of objectives including objectives 3, 4 and 6.
- Chapter 7 will involve concluding if the aim and all the objectives have been met. Justifications will be provided to support these claims. This chapter looks to address all objectives and the aim.

2 Background

Being able to predict hard drive failures is becoming essential, as unplanned downtime and data loss can incur a massive cost for big corporations such as server providers. This is especially notable in resource constrained devices where its more common to run the operating system off an SD card which have limited backup storage capacities, making hard drive failures particularly disruptive compared to enterprise systems. If the primary storage fails unexpectedly, it may not be feasible to quickly recover data or restore functionality. This could result in the device being inoperable until the storage is replaced. Predicting hard drive failures allows preventative measures to be put into place in advance, this may include copying data or buying a replacement. With this in mind, there has been significant research on applying machine learning techniques to accurately identify hard drive failures.

2.1 Machine Learning For HDD Failure Prediction

Several studies have explored the use of different machine learning models for predicting hard drive failures. A common approach is to collect sensor data from the SMART (Self-Monitoring, Analysis and Reporting Technology) data statistics of the hard drive, such as spin-up-time, reallocated sectors count, seek error rate, etc. These are then taken and used as input features based on their value to train the predictive models.
One study by (Yazici, Basurra and Gaber, 2018) explored the feasibility of running machine learning algorithms on IoT (Internet of Things) devices. A Raspberry Pi 3 was used to test the performance of three different machine learning algorithms. This included Multi-Layer Perceptron, Support Vector Machine (SVM), and Random Forest. The algorithms were tested on 10 different diverse datasets covering classification and regression problems. The performance metrics gathered were accuracy, speed (training and inference), and power consumption. The key findings from this paper suggest that Random Forest exhibited the highest accuracy amongst all three for both classification and regression tasks. SVM was slightly faster in inference and more efficient in power consumption compared to the other two. Overall, the results showed that running popular machine learning algorithms on resource-constrained devices is feasible, with reasonable accuracy, speed, and power consumption.
In relation to resource-constrained devices the performance evaluation metrics that this paper provides are directly applicable to my use case of predicting hard drive failures. It also demonstrates the feasibility of running advanced machine learning algorithms on embedded

devices like the Raspberry Pi. The methodology of testing multiple machine learning algorithms and comparing their trade-offs are similar to what I aim to do in my research. Another study by (Pinheiro Eduardo, Weber Wolf-Dietrich and Barroso Andre Luiz, 2007) examined failure patterns in a large population of hard drives (over 100,000) in a production internet services deployment at Google. This is one of the largest disk drive population studies, providing insights that go beyond previous studies which were based on smaller populations. An infrastructure was built that collected and analysed detailed health and environmental data from the disk drives over time, including SMART parameters, temperatures, and activity levels. The key findings from this article show that the authors found very little correlation between failure rates and elevated temperature or high activity levels. Several SMART parameters, such as scan errors, reallocation counts, offline reallocation counts and probational counts all showed a strong correlation with higher failure probability. However over 56% of failed drives showed no predictive SMART signals at all, suggesting that models based solely on SMART data are unlikely to be accurate in predicting individual drive failures. They conclude that SMART data can be useful in understanding reliability trends of large disk populations but may not be sufficient for building accurate predictive models of individual drive failures. A limitation of this research is that the study was based on a specific development scenario (Google's internet services infrastructure), and the results may not generalise to other environments or use cases.
A study by (J. Zeng *et al.,* 2022) on the prediction of hard drive failures for data centres based on LightGBM suggested that traditional SMART methods have low fault prediction accuracy (3-10%), motivating the use of more advanced machine learning techniques. The paper uses the Backblaze public dataset (*BackBlaze Hard Drive and Stats.* 2023), which contains SMART data and failure information for over 47,000 hard drives. The authors performed data preprocessing, including handling null values, removing small sample data, and addressing the imbalance between positive (failed) and negative (healthy) samples. Feature engineering was performed selecting the most relevant SMART values. The key findings suggest adding more positive samples (from historical HDD failure data) to the training set improves the model's performance, reducing the number of iterations required to reach the maximum AUC (Area Under the Curve) value. The proposed LightGBM-based approach achieves high accuracy in predicting HDD failures, demonstrating its effectiveness for improving the reliability of data centre storage systems.
In relation to resource constrained devices the data preprocessing techniques, feature engineering and handling of the imbalanced datasets discussed in this paper provide valuable insights that could be applied to my comparative analysis. Whilst the paper focuses on data-centres, the general principles and methodologies presented can be adapted and scaled down

to fit in line with resource-constrained devices. Comparing the performance, efficiency, and suitability of LightGBM against other algorithms (Decision Tree, Random Forest, Neural Networks) in the context of HDD failure prediction on resource-constrained devices would be a natural extension of the work presented in this paper.

A limitation of this paper is the lack of testing in a real-world environment as the paper doesn't address how the model would perform in a true real-world setting with a more extreme class imbalance.

A study by (Đurašević and Đorđević,) investigates the potential of predicting hard drive failures using SMART indicators. The research utilises a large dataset of SMART indicators from a large population of hard drives, operating under similar conditions. The study tests the performance of an anomaly detection algorithm based on statistical distribution for predicting hard drive failures. The algorithm is trained using a set of SMART indicators from Seagate ST3000DM001 disk drives. The performance metrics evaluated were precision, recall, and F1-score. The key findings from this paper suggest that the anomaly detection algorithm achieved a high precision of over 90% in failure detection. The algorithm provided an average of 38.9 days warning prior to the actual failure of the hard drive.

In relation to resource-constrained devices, the performance evaluation metrics that this paper provides are directly applicable to my use case of predicting hard drive failures. It demonstrates the feasibility of running advanced anomaly detection algorithms for predicting hard drive failures. The methodology of testing the anomaly detection algorithm and comparing its trade-offs are similar to what I aim to do in my own research. The study also highlights the importance of SMART indicators in predicting hard drive failures, which could be relevant to my topic of comparing multiple machine learning algorithms for predicting hard drive failures on resource-constrained devices. Overall, the results showed that predicting hard drive failures using SMART indicators is feasible, with reasonable precision, recall and F1-Score.

A study by (G. F. Hughes *et al.,* 2002) aims to improve the SMART failure warning algorithms, due to current SMART failure prediction systems in disk drives having only moderate accuracy at low false-alarm rates. The authors propose two improved SMART algorithms that use the SMART internal drive attribute measures. They replace the current warning-algorithm based on maximum error thresholds with distribution-free statistical hypothesis tests, specifically Wilcoxon rank-sum tests. The new rank-sum based algorithms were tested on 3744 drives of 2 models and gave 3-4 times higher correct prediction accuracy than error thresholds on drives that will fail, at a 0.2% false-alarm rate. The highest accuracies achievable are modest, around 40-60%. The paper argues that rank-sum tests are advantageous because they make no assumptions about the statistical distributions of the data, are robust to outliers, and exploit the known monotonicity of the SMART attributes (larger values imply increased failure risk).

In relation to resource-constrained devices, the paper proposes statistically based, computationally simple algorithms (rank-sum tests) as an alternative to the current thresholding approach in SMART, which could be suitable for resource-constrained devices. The experimental results demonstrate the potential to improve upon the accuracy of current SMART failure warnings, which is an important feature for any new machine learning approaches. The challenges of predicting rare failure events with low false alarms are mentioned, this is a key consideration when applying machine learning to this problem in resource-constrained settings.

A study by (Pitakrat, Van Hoorn and Grunske, 2013) evaluates and compares 21 different machine learning algorithms for proactive hard drive failure detection. It aims to detect failures in advance so that preventative or recovery measures can be planned and implemented. The dataset used is a publicly available dataset of hard drive parameters collected using the SMART system. The dataset contains 68,411 instances from 369 hard drives, 178 of which are good and 191 failed during operation. The authors pre-processed the data to separate the instances into failing and non-failing classes, using a 7-day window before failure as a guideline. The authors used various evaluation metrics to assess the prediction quality of the algorithms, including true positive rate, false positive rate, precision, accuracy, F1-score, training time, prediction time and area under the ROC curve (AUC). The key findings suggest that different algorithms perform better under different constraints and requirements. The top performing algorithms in terms of predictive quality are NNC (Near Neighbour Classifier), Random Forest, C4.5, REPTree, RIPPER, PART, and K-Star. NNC has the highest true positive rate and F1-score, but as the false positive rate increases, it gets outperformed by Random Forest. Algorithms like SVM and SMO (Sequential Minimal Optimisation) have the lowest false, but their true positive rate is much lower compared to the top-performing algorithms. In terms of both training and prediction time the fastest algorithms are instance-based learning and simple classifiers such as K-Star, NNC, whilst SVM, Simple Logistic Regression, and Multilayer Perceptron have the longest training times. The choice of algorithm depends on the specific application requirements and constraints, such as the importance of prediction quality vs prediction time.

In relation to resource-constrained devices, this paper highly relevant to my own research topic on comparing multiple different machine learning algorithms to predict hard drive failures in resource-constrained devices. It provides a comprehensive evaluation of a wide range of machine learning algorithms specifically for the task of hard drive failure detection, which is directly applicable to my own requirements. The findings and recommendations

from this research paper can act as additional guidance during the process of selection algorithms for my own requirements.
The last study by (R. Xu, X. Wang and J. Wu, 2022) explores classification-based methodologies, performance evaluation and comparisons for hard drive disk failure found that the error count attributes SMART 197 (current pending sector count) and SMART 187 (reported uncorrectable errors) were found to be closely related to HDD failures. Several preprocessing techniques were applied to the data, including missing data imputation, discretisation, adding time-series features, and data relabelling. The authors compared the performance of seven popular classification models. This included Decision Tree, SVM, Naïve Bayes, AdaBoost, Random Forest, Multilayer Perceptron, Neural Network, and Long Short-Term Memory (LSTM) model. The key findings suggest that Random Forest performed the best, with a Failure Detection Rate (FDR) of 53.95% and a False Alarm Rate (FAR) of 6.0%. In relation to resource-constrained devices, this paper provides a comprehensive evaluation and comparison of several popular machine learning models for HDD failure prediction. The findings feature selection, data preprocessing, and model performance comparisons can provide valuable insights.

2.2 Challenges on Resource Constrained Devices

Deploying complex machine learning models on resource-constrained devices like embedded systems can be challenging due to the limited memory, RAM, and CPU capabilities. This has resulted in research producing more efficient models and techniques. (Choudhary *et al.,* 2020) Extensive survey on model compression and acceleration techniques. It discusses the challenges of deploying large deep learning models on devices with limited resources. The paper explores the performance of various techniques for model compression and acceleration such as pruning, quantisation, low-rank factorisation, and knowledge distillation. These techniques are tested on a variety of deep learning models, including both CNNs (Convolutional Neural Networks) and RNNs (Recurrent Neural Networks). The performance metrics considered in the paper are model size, computational requirements, and accuracy. The key findings suggest that these techniques can significantly reduce model size and computational requirements without a substantial loss in accuracy. For instance, pruning and quantisation methods are found to be particularly effective in reducing model size while maintaining performance. Knowledge distillation is highlighted as a promising technique for transferring knowledge from larger models to smaller ones, achieving high accuracy with a smaller model size.

In relation to resource-constrained devices, the techniques and findings presented from this paper are directly applicable to the use case of predicting hard drive failures. The methodology of testing various model compression and acceleration techniques and comparing their trade-offs align with the approach of comparing multiple different machine learning algorithms for predicting hard drive failures.

2.3 Conclusion

In conclusion the research reviewed in these documents highlights the significant potential of applying machine learning techniques to predict hard drive failures, especially in the context of resource-constrained devices. Several studies have explored the use of various machine learning models and anomaly detection algorithms, to accurately identify hard drive failures using SMART data as input features. Overall, the research reviewed indicates that with the right combination of feature engineering, model selection and optimisation techniques, it is feasible to develop effective hard drive failure prediction systems that can be deployed on resource-constrained devices. This would enable proactive maintenance and minimise the risk of unexpected downtime and data loss, particularly in critical applications relying on embedded systems. Further research and real-world validation of these approaches are needed to fully realise their potential.

3 Methodology

This chapter outlines the systematic approach adopted to achieve the aim. It details the methodology used for development as well as the processes involved in model selection, training, evaluation, and implementation. This ensures the generation of unbiased results essential for drawing objective conclusions.

3.1 Methodology Selection

The type of methodology used in this project is Agile. The project involves iterating through multiple different stages of model implementation such as model training, testing, evaluation, and implementation. Continuous feedback from the performance metrics allows constant evaluation of model performance and refinement of approach. Flexibility and adaptability with multiple machine learning models as they will be tested against the same dataset. From this the approach will be adapted based on the performance metrics obtained, which could involve model selection criteria or implementation strategies.

3.2 Data Collection

I will be using a publicly accessible dataset from Kaggle (Ghassen Khaled, 2023). I will explain the dataset in depth in chapter four. Furthermore, I was recommended by my second marker to try and look for datasets that contained hard drives or SD cards that were commonly used on Raspberry Pi's but was unfortunately not able to find anything in this area so opted for normal hard drive data.

3.3 Data Formatting and Preprocessing

To ensure effective data formatting and preprocessing exploratory data analysis will be undertaken to gain insights into data distribution, features, and potential challenges. Preprocessing of the dataset to address missing values and data imbalances. Splitting the dataset into training, validation, and test sets to accommodate model development and evaluation. The methodologies listed above will ensure effective data formatting and preprocessing resulting in high reliability and accuracy of machine learning models along with a minimised dataset that enhances the robustness of the subsequent model training and evaluation processes.

3.4 Model Selection and Training

To identify a model that offers the best balance between predictive performance and computational efficiency a variety of machine learning algorithms suitable for the problem domain will be identified. This includes but is not limited to decision trees, boosting algorithms and ensemble methods. Training multiple models with the training dataset, whilst optimising hyperparameters through techniques such as cross validation. Evaluating the model's performance using appropriate metrics, such as accuracy, precision, F1-score, recall, AUC-ROC curve, and testing time. Iteration through the model selection and training process occurs to identify the most promising candidate models. The methodologies listed above will accommodate the successful identification of a model that offers the best balance between predictive performance and computational efficiency, ensuring suitability for resource constrained devices.

3.5 Performance Evaluation

To identify models that offer superior predictive accuracy, computational efficiency, and robustness an assessment regarding the performance of selected models can be undertaken to validate the models' capabilities. Conducting performance evaluation using established metrics to compare model performance. Analysing model interpretability and robustness to confirm the reliability and trustworthiness of predictions. Refining the model selection criteria based on performance evaluation results, iteratively improving the selection process until the most balanced model is established. The methodologies listed above will assist in the successful identification of a model that offers superior predictive accuracy, computational efficiency, and robustness, thus meeting the requirements to accommodate resource constrained devices.

3.6 Model Implementation and Deployment

To validate model performance in real world settings implementation of the selected model using appropriate frameworks or libraries compatible with resource constrained devices must be adhered to. Optimisation of the model's size or inference speed to ensure efficient deployment. Validation of the model's performance in real world scenarios using simulated environments. The methodologies listed above will demonstrate the feasibility and effectiveness of deploying a machine learning model on resource constrained devices.

4 Data Formatting and Preprocessing

This chapter will discuss the interface design, preprocessing techniques undertaken, model testing and comparison to present the general guidelines followed to ensure a non-biased testing environment which allows for fair comparisons to be made. This section will also briefly touch on the process of training and exporting the model as well as how the front end plays a part along with the deployment process of the final solution.

4.1 Interface Design

The idea with the design is to have the user click a button on a web UI and provide either a 0 or a 1 based on if the model thinks the drive will not fail or fail. To know which, drive to make a prediction on the user must specify the hard disk drive path before launching the docker container.

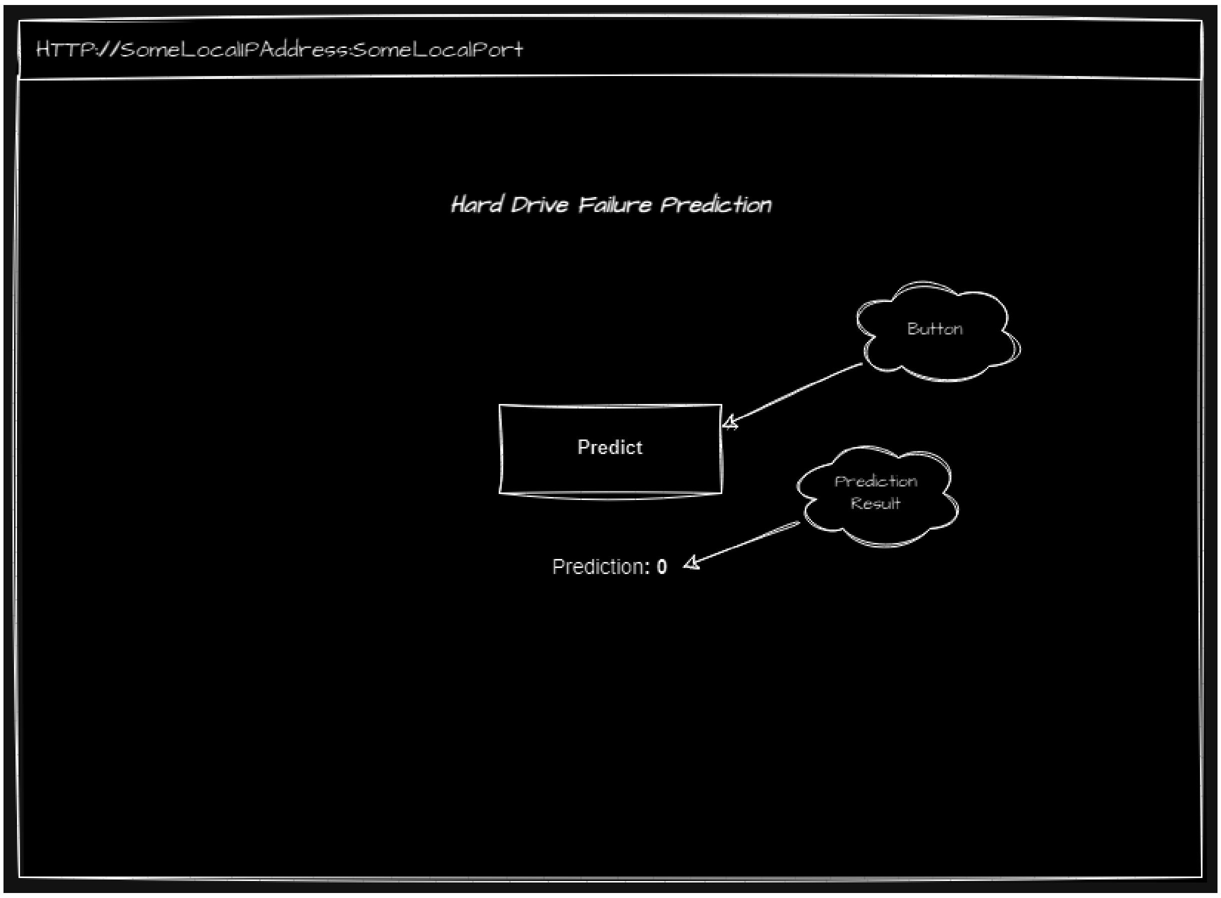

HTTP://SomeLocalIPAddress:SomeLocalPort
Hard Drive Failure Prediction
Button
Predict
Prediction Result
Prediction: 0

Figure 4.1 Hard Drive Failure Prediction Web UI

4.2 Preprocessing Techniques

4.2.1 Merging the data

The original dataset, from Kaggle (Ghassen Khaled, 2023) contains one hundred and eight one CSV (comma separated value) files each referencing the individual day, covering a total period of six months. To work with this very large unsorted raw data, I first must merge all the CSV files together. I intend to do this by utilising a tool called OpenRefine (Open Source Community, 2024) which allows easy and efficient merging of data for large datasets.

4.2.2 Dropping Columns with Missing Values

In Orange3 the merged dataset will be analysed with the feature statistics function and anything that has more than twenty percent of missing values will be dropped afterwards in OpenRefine. The reason I will not drop any of the redundant columns via Orange3 (Demsar J, Curk T, Erjavec A, Gorup C, Hocevar T, Milutinovic M, Mozina M, Polajnar M, Toplak M, Staric A, Stajdohar M, Umek L, Zagar L, Zbontar J, Zitnik M, Zupan B, 2013) is due to the application repeatedly crashing with such large amounts of data from the dataset.

4.2.3 Removal of Redundant Columns

With the feature rank information obtained from Orange3. The merged dataset will now be imported into OpenRefine and the redundant columns removed using OpenRefine' s faceting feature which will allow mass data alteration with a single command. Other than the SMART values being removed I also discarded the

4.2.4 Column Feature Importance

The newly condensed dataset will be imported into Orange3 and the columns will be ranked based on their overall value to the target variable which in our case is the failure column. The main reason behind this is so that the most influential SMART value for predicting disk failure is identified. This ended up being the SMART_5 values.

4.2.5 Removal of Redundant Rows

In the dataset we have various hard drive models which are repeated multiple times across the same day, this pattern is present throughout the entire dataset. It is not necessary to have so much excess repetition present. To solve this the ratio for the number of model numbers to remain against the number of total failures must be decided. On recommendation of my tutor (also find research that also does this to evidence further) I decided to go with a “10:1” ratio, in other words for every ten non-failures we should have one failure to simulate a normal environment as failures aren’t common. We have one thousand and fifty-eight failures present in total; this means that we would only need to discard the non-failures as we have a great minority of failures to begin with, ending up with around ten thousand non-failures to maintain our ratio.

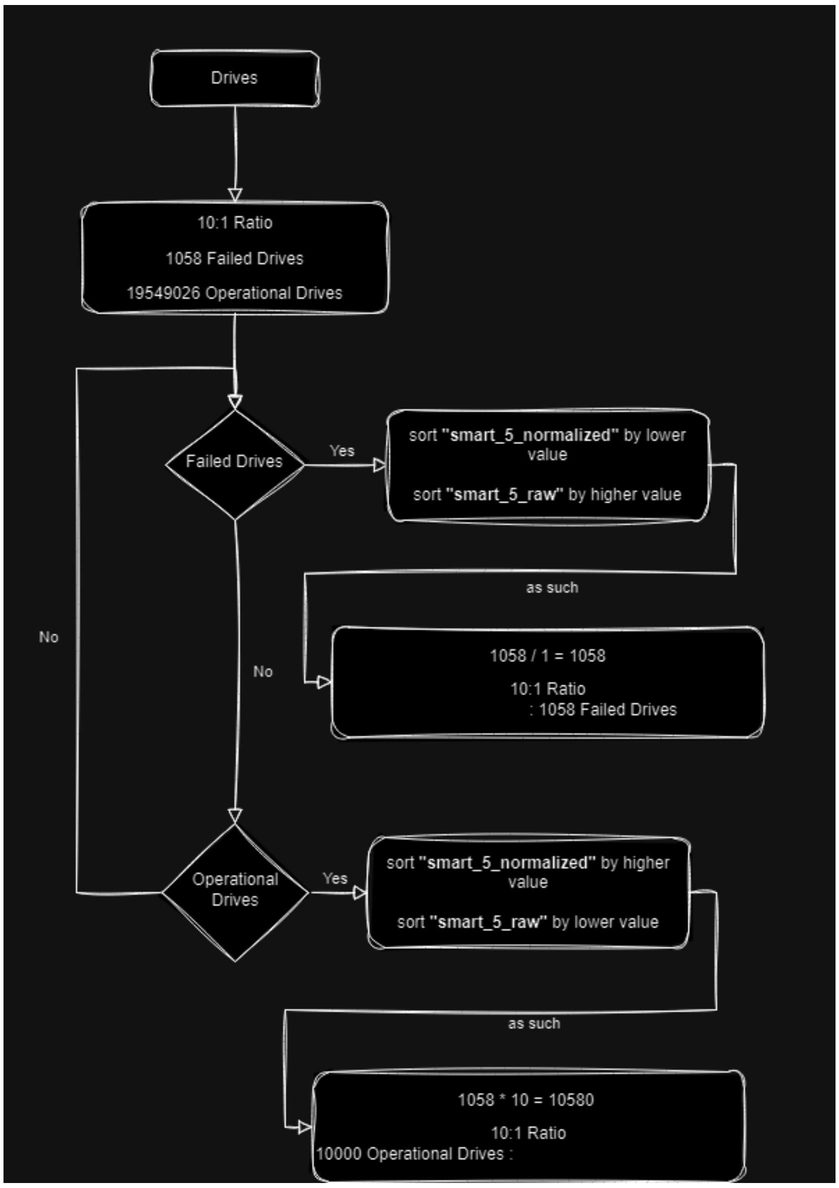

Drives
10:1 Ratio
1058 Failed Drives
19549026 Operational Drives
Failed Drives
Yes
sort "smart_5_normalized" by lower value
sort "smart_5_raw" by higher value
as such
1058 / 1 = 1058
10:1 Ratio
: 1058 Failed Drives
No
No
Operational Drives
Yes
sort "smart_5_normalized" by higher value
sort "smart_5_raw" by lower value
as such
1058 * 10 = 10580
10:1 Ratio
10000 Operational Drives :

Figure 4.2 Flow Chart Showcasing Drive Selection Process

Figure 4.2 above shows the process of how the “10:1” ratio is calculated. The two SMART five values “smart_5_normalized” and “smart_5_raw” are reversed based on if it’s a failed or operational drive. A lower value of “smart_5_normalized” typically indicates a higher likelihood of disk failure. This is because the normalized value is scaled to represent the likelihood of failure, with lower values generally indicating a higher risk. A higher value of "smart_5_raw" indicates a higher count of reallocated sectors, which in turn can indicate a higher likelihood of disk failure. This raw value provides the actual count of reallocated (bad) sectors on the disk, a higher count of reallocated sectors being associated with a higher risk of failure. Let's take failed drive as an example, we sort the "smart_5_normalized" by a lower value and "smart_5_raw" by a higher value to find the drives with the highest chance of failure, in other words the worst performing. Then for the operational drives the "smart_5_normalized" and "smart_5_raw" values are reversed to give the best performing drives, or in other words the drives that are running most optimally.

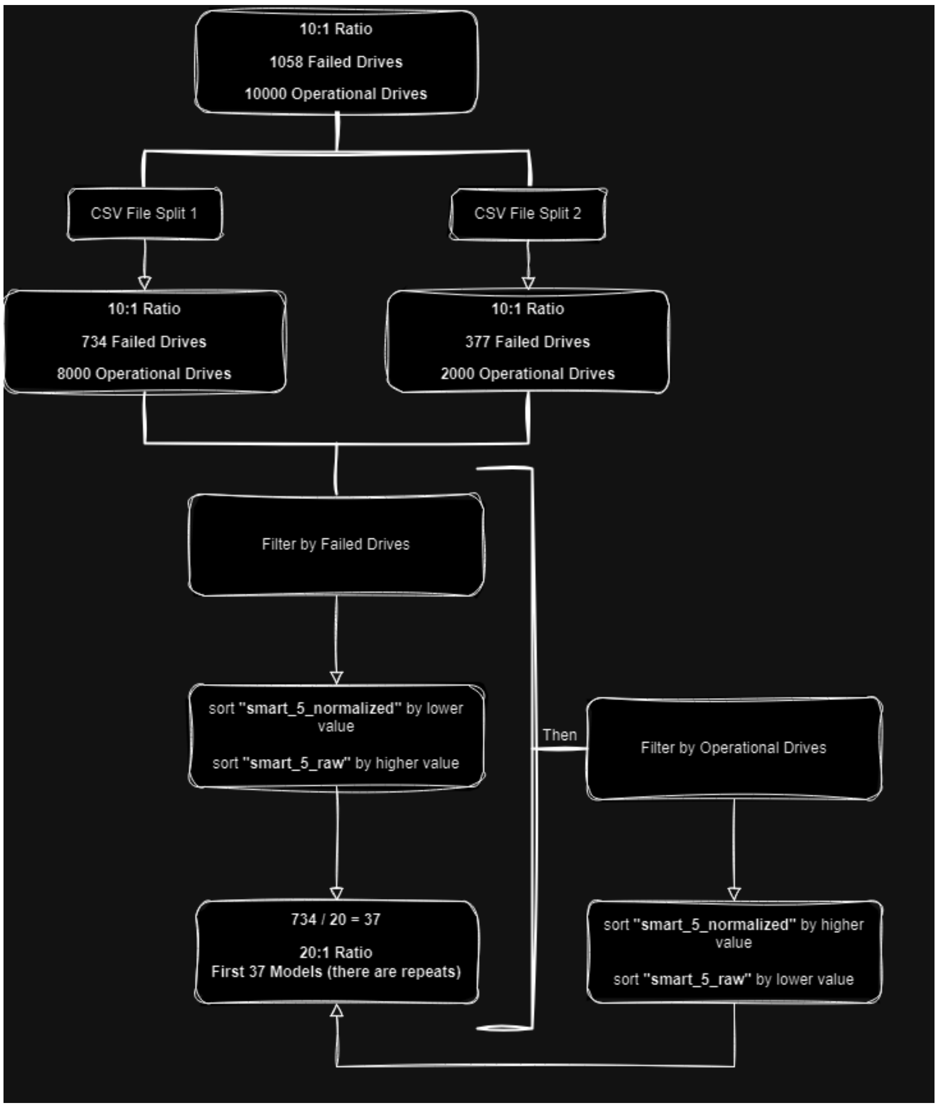


Figure 4.3 Flow Chart Showcasing Methodology Maintained Between Two Split CSV Files

Figure 4.3 shows the methodology used to maintain a "10:1" ratio between two CSV files. The first file contains 8000 operational drives whilst the second contains 2000. The reason they are so unevenly distributed is because the second CSV file covers the last month whilst the first covers a period of three months, to account for that I had to balance the distribution of each file appropriately to ensure a singular month doesn't match multiple. We repeat the same procedure as discussed for Figure 4.2 where we filter by the failed drives first, filtering "smart_5_normalized" by lower values and "smart_5_raw" by higher values. There is seven hundred and thirty-four failed drives in the first CSV file we divide that by twenty as we want to use a "20:1" ratio to give us the number of models (provide justification based on research why 20:1 ratio), leaving us with thirty-seven models including repeats. This can be visually represented below in Figure 4.4.

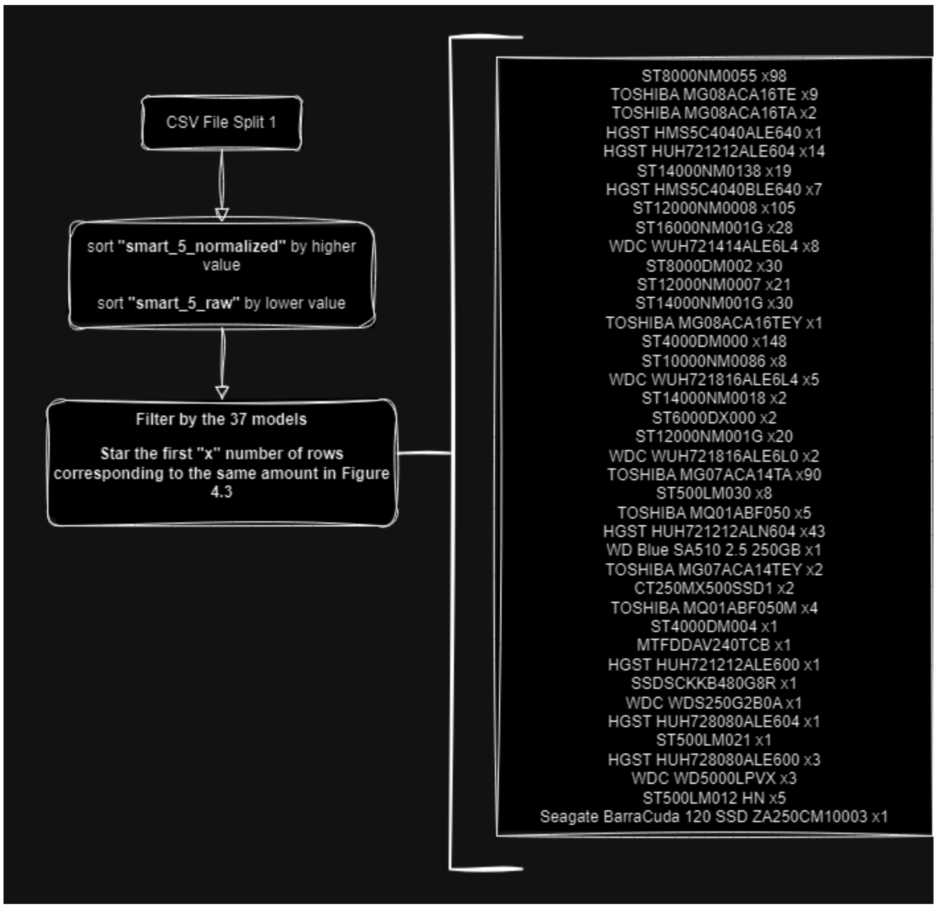


Figure 4.4 Flow Chart that Visually Represents the Model Selection Process CSV File 1

Continuing with the explanation in Figure 4.3 once we have selected the first thirty-seven models within the failure category we filter by operational drives and reverse the "smart_5_normalized" and "smart_5_raw" values to find the best performing drives. Then selecting the first thirty-seven models that match the models identified in Figure 4.4. This procedure is shown below in Figure 4.5.

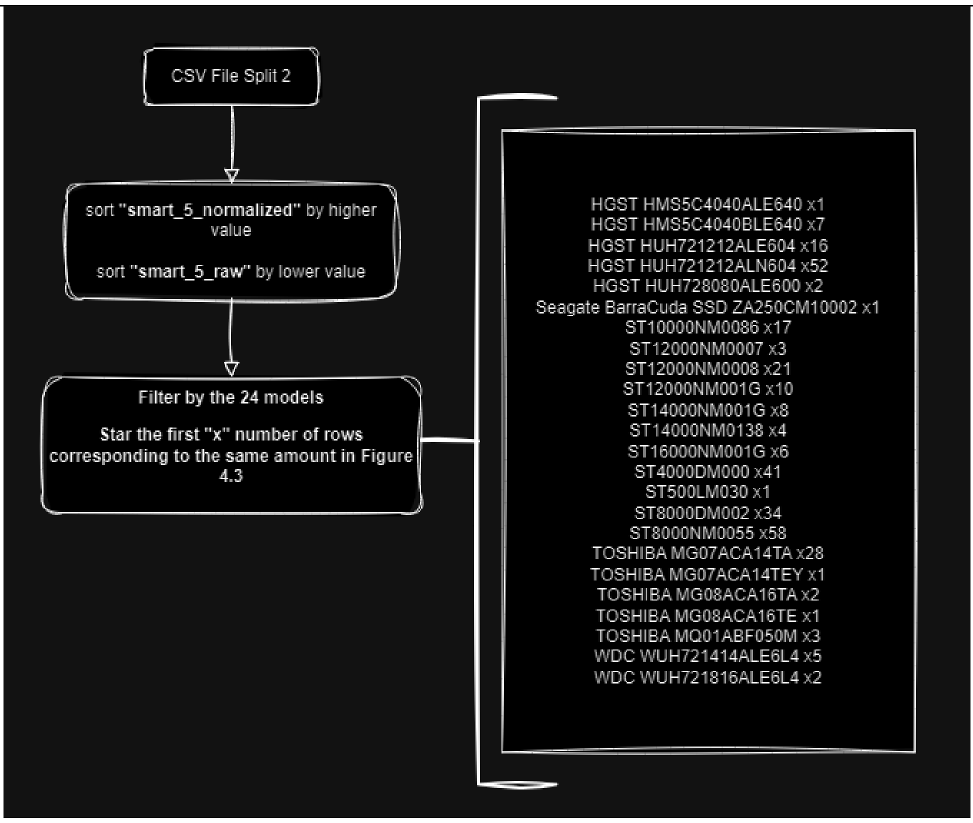


Figure 4.5 Flow Chart that Visually Represents the Model Selection Process CSV File 2

Figure 4.4 and 4.5 should be thought of as subsets of Figure 4.3. In Figure 4.5 the same process applies as discussed in the previous Figure 4.4. The sole difference is there is only twenty-four drives that match the best performing drives in Figure 4.4 as opposed to thirty-seven. This was not intended but was unknown that the raw data would be missing some models in certain months until the point was reached. Regardless the intention was to continue with this amount as it is only a one-month period as opposed to the other three months which includes those thirteen extra models not present in the second CSV file.

4.3 Multiple Model Testing and Comparison

Shown below is Figure 4.6 which showcases the general steps that will be taken to provide a fair testing environment as well as a comparison for each model.

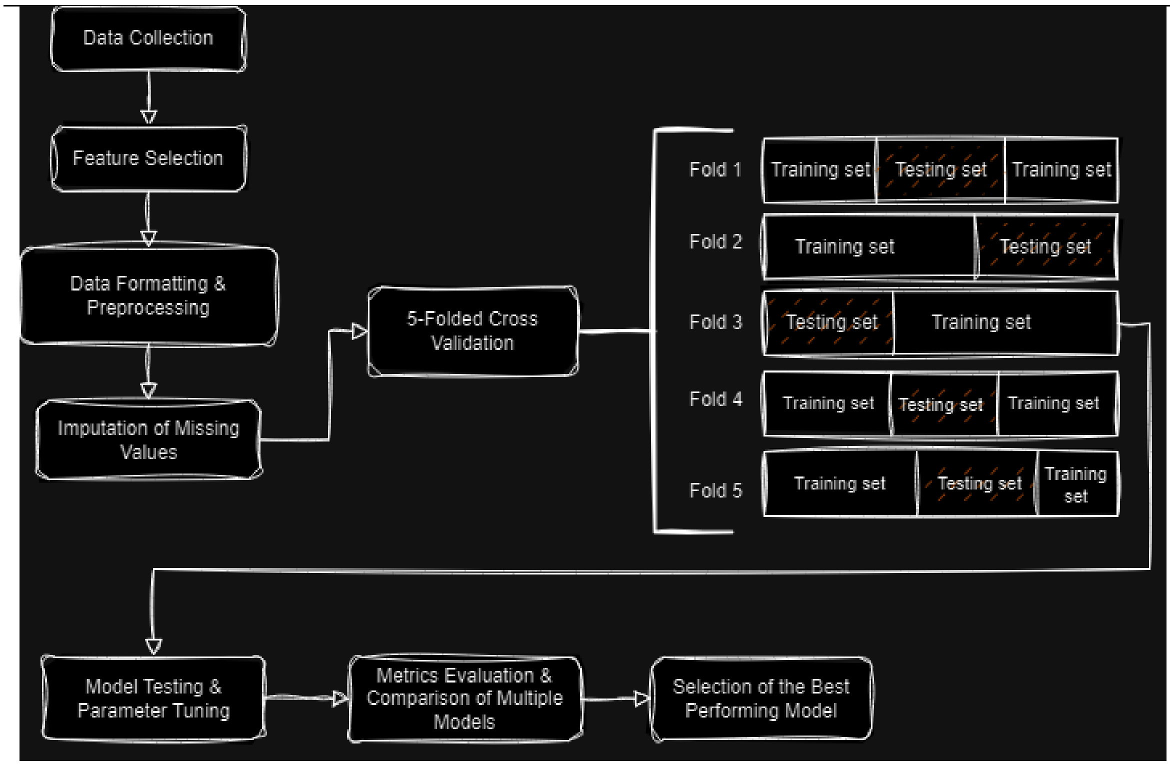


Figure 4.6 Model Testing, Comparison and Selection Process

The basis for this is to find the model with the best trade-off between predictive accuracy and testing time. A lower testing time would indicate that the model is very lightweight and efficient which is what we are aiming towards. On the other hand, a higher predictive accuracy would indicate that the model is very accurate at predicting failures regardless of the computational costs. 5-folded cross validation will be utilised to ensure there is no over-fitting, allowing evidence which provides the most accurate results and allows the best model to be chosen.

Imputation is also utilised due to certain models being unable to handle missing values. Some missing values could not be removed as the same rows also provided very valuable information in other columns. Therefore, removing them would be detrimental to the model's predictive performance.

4.4 Training and Exporting the Model

The simple flowchart provided below in Figure 4.7 showcases the steps that will be taken when training the model.

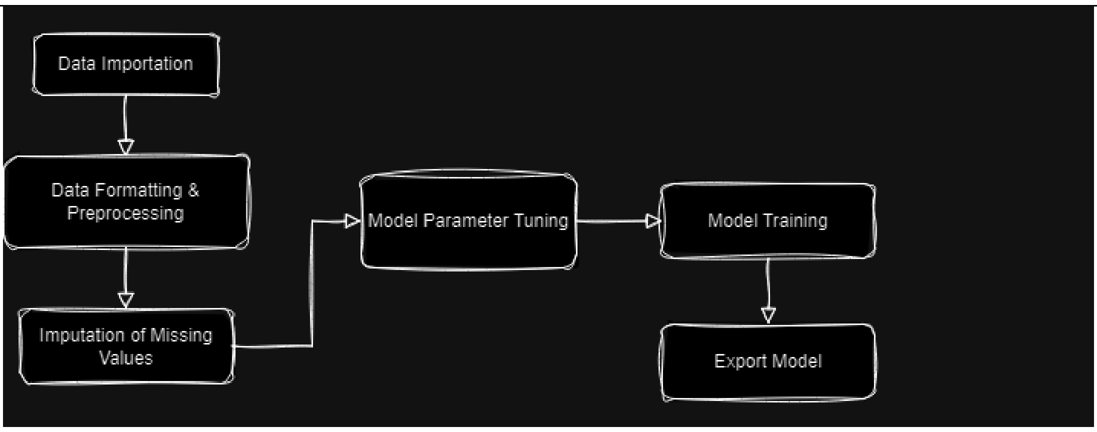


Figure 4.7 Model Training and Exportation Process

4.5 Making Predictions on External Data

The simple flowchart provided below in Figure 4.8 showcases the steps that will be taken to allow the trained model to make predictions on current device statistics.

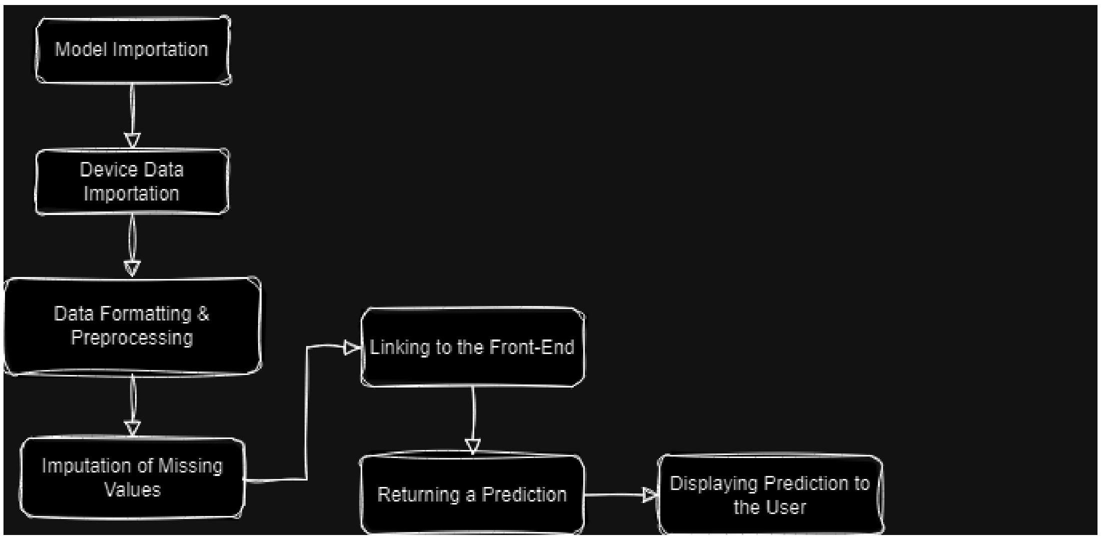

Model Importation
Device Data Importation
Data Formatting & Preprocessing
Imputation of Missing Values
Linking to the Front-End
Returning a Prediction
Displaying Prediction to the User

Figure 4.8 Making Failure Predictions Process

4.6 Front-End

There will be a web UI (user interface) where the user can interact with a button to display the predictions from the model on the initially stated device. The front UI will be very simple as I just want to show basic functionality.

4.7 Deployment

A choice will be made between choosing to deploy the project locally or to package it as a docker container and deploy it on Docker. There will be comparisons and evaluations made between the two, finally a choice will be made as well as appropriate justifications.

5 Implementation

Continuing from the methodology, this chapter will discuss in detail the process of implementing, training, and producing metrics for all the models. The training then exportation of the single best performing model followed by the implementation and deployment of the final solution.

5.1 Multiple Model Implementation

To evaluate the performance of each model I have developed appropriate code in Python which loads the cleaned dataset from a CSV file, proceeds with additional preprocessing techniques including label encoding, data feature extraction, and dropping unnecessary columns. Missing values are handled using the mean strategy. The model is imported and initialised with the best performing hyperparameters. The dataset is split into five folds using cross validation to ensure balanced class distribution for each fold. The evaluation metrics such as accuracy, precision, and recall are calculated for each fold of cross validation. The mean and standard deviation are computed to assess the overall performance of the model. The time taken for model training and testing in each fold is measured using appropriate techniques so that we can evaluate how efficient the model is by taking an average. This is followed by producing various infographics including confusion matrices which are visualised using heatmap plots. A classification report is generated to provide a summary of the precision, recall, F1-score, and support scores for each class. Additionally, SHAP (SHapley Additive exPlanations) values are computed and visualised to explain the model's predictions, summary plots and feature importances. The use of such infographics allows more specific comparisons between certain features.

5.1.2 Balanced Random Forest

Figure 5.1 provides us with the pseudocode that shows steps undertaken to train the model and produce the necessary evaluation metrics to evaluate and compare the model's performance. Below I will go into further detail on the pseudocode and provide the reasons for choosing the methods that were utilised.

```
1. Import necessary libraries

2. Load dataset from file

3. Preprocess data:
   a. Encode categorical variables using LabelEncoder
   b. Convert datetime column to year, month, and day
   c. Drop original datetime column
   d. Splitting data into features and labels
   e. Impute missing values using SimpleImputer

4. Initialize BalancedRandomForestClassifier with specified parameters

5. Initialize StratifiedKFold for cross-validation

6. Initialize lists to store evaluation metrics and timing information

7. Start overall training timer

8. Perform Stratified K-Fold cross-validation:
   a. Loop over each fold
   b. Split data into train and test sets
   c. Start timer for training
   d. Train the classifier on the training data
   e. Start the timer for testing
   f. Make predictions on the testing data
   g. End the timer for testing
   h. Calculate evaluation metrics (accuracy, precision, recall, F1 score, AUC)
   i. End the timer for training

9. End overall training timer

10. Print evaluation metrics:
    a. Cross-validation accuracy scores (mean, standard deviation)
    b. Cross-validation precision scores (mean, standard deviation)
    c. Cross-validation recall scores (mean, standard deviation)
    d. Cross-validation F1 scores (mean, standard deviation)
    e. Cross-validation AUC scores (mean, standard deviation)
    f. Mean training time and standard deviation
    g. Mean testing time and standard deviation

11. Plot overall confusion matrix

12. Generate overall classification report

13. Print overall training time

14. Initialize SHAP explainer

15. Compute SHAP values

16. Get feature names

17. Plot SHAP summary plots with feature names

18. End
```

Figure 5.1 Pseudocode Showcasing the Implementation of Balanced Random Forest

Step 1 involves importing the necessary libraries. I have produced a list below which shows all the modules I have imported and the general functionalities they offer.

- Balanced Random Forest classifier from imblearn. To implement cross validation, I decided to utilise StratifiedKFold from the sklearn library, I decided to utilise sklearn' s model_selection module because it is a very reliable, popular, and respected choice within the machine learning domain.

- To provide metrics for evaluation I used sklearn' s metrics module importing accuracy_score, recall_score, precision_score, f1_score, confusion_matrix, classification_report and lastly roc_auc_score. This allows a fair comparison to be made between different machine learning algorithms to evaluate the best performing model for our use case.
- Imported LabelEncoder from sklearn' s preprocessing module. This will allow me to encode categorical labels as numeric values which then allows the machine learning algorithm to utilise the features contained in a column.
- Imported SimpleImputer from sklearn' s impute module. This allows missing values contained in the dataset to be imputed with simple strategies such as mean, median or mode so that the model can be successfully trained on the data as most models cannot automatically handle missing values. This will provide me with the functionality to easily deal with any missing values.
- Imported matplotlib' s pyplot module which makes matplotlib work like MATLAB for creating various infographics such as plots. This will provide me with the functionality to create richer infographics to allow for an easier understanding of results.
- Imported seaborn which is a data visualisation library built on top of matplotlib and provides extra functionalities. Seaborn simplifies the process of creating complex visualisations such as heatmaps, pair plots, violins plots and provides additional functionality for statistical analysis. This will give me the options to produce more specific graphs such as heatmaps and violin plots which will allow me to break down specific functionalities of the model to draw evidence and provide analysis.
- Imported NumPy which utilises arrays that can execute high level mathematical operations on larger data sets in a more simple and efficient manner compared to Python's built in lists. The benefit of this is not very significant but allows the production of more efficient results which can reduce the code's runtime.
- Imported Pandas which can read data in various formats such as CSV, EXCEL, SQL databases, can manipulate data, handle missing values, and provide descriptive summaries and statistics. This will allow me to import my CSV file and preprocess it by manipulating the data.
- Imported Python's time module which provides various functions for working with time related tasks, such as measuring time intervals and delaying executions. This can

provide me with a way to accurately obtain the testing and training time metrics of the model.

- Imported SHAP which provides explanations regarding the output of machine learning models. It stands for SHapley Additive exPlanations. SHAP values provide a way to interpret the impact of features on model predictions. It is commonly used for understanding model predictions, feature importance analysis and model debugging. This module would give me the option to provide more specific infographic information providing further evidence and analysis to work with.

Step 2 is the importation of the dataset; Pandas was used to read the CSV file and store the data in a DataFrame.

Step 3 involves the preprocessing steps that were done prior to the data being trained on the classifier.

- Step 3a involved fitting the model column within the dataset to the LabelEncoder as this column contained categorical labels of the hard drive names in non-numerical format which prevents the machine learning algorithm from being unable to utilise this column. As such I used LabelEncoder to encode the categorical data into numerical data then saved it to the DataFrame.
- Step 3b involved converting the date column from the dataset into separate year, month, day columns. To do this I converted the values in the date column of the DataFrame to datetime objects using the ‘`pd.to_datetime()`’ function. I then extracted the year, month, and day components from the datetime values in the date column using the ‘`.dt`’ accessor, which provides access to datetime components. This splits the datetime information into separate numerical columns representing the year, month, and day of each date.
- Step 3c involved dropping the original date column as this will remain as part of the DataFrame when no longer required. To do this I utilised the ‘`drop()`’ function. The line of code “`data.drop('date', axis=1, inplace=True)`" where the ‘`axis=1`’ argument indicates that we want to drop a column as opposed to a row, and ‘`inplace=True`’ specifies that the changes should be applied directly to the DataFrame ‘`date`’.
- Step 3d involved splitting the data into features and labels. “`X = data.drop(['failure', 'year', 'month', 'day', 'model'], axis=1)`” and “`y = data['failure']`” lines show that ‘`X`’ represents the features (input variables) used for prediction, whilst ‘`y`’ represents the target variable (the variable to be predicted). Within ‘`X`’ the columns dropped are ‘`failure`’, ‘`year`’, ‘`month`’, ‘`day`’ and ‘`model`’, as they are already processed or irrelevant.

Within ‘`y`’ the ‘`data['failure']`’ selects the ‘`failure`’ column from the DataFrame containing the target variable which is then assigned to ‘`y`’.

- Step 3e involved imputing missing values using SimpleImputer. To do this I defined a strategy for the SimpleImputer followed by assigning SimpleImputer to ‘imputer’ this is shown as “`imputer = SimpleImputer(strategy='mean')`”. Next, I fitted and transformed the features identified as ‘`X`’ to the imputer and assigned the variable as ‘`X_imputed`’. This can be shown as ‘`X_imputed = imputer.fit_transform(X)`’. This variable holds the imputed version of the feature matrix ‘`X`’. I intend to use ‘`X_imputed`’ in all subsequent code for training the model as well as further analysis.

Step 4 is where the initialisation of the Balanced Random Forest classifier takes place as well as defining the specified parameters for the classifier. This is shown as “`clf = BalancedRandomForestClassifier(n_estimators=z, min_samples_split=z, random_state=None)`”, where the ‘`BalancedRandomForestClassifier`’ is assigned to ‘`clf`’, in which ‘`z`’ is a numerical value that will be specified and explained in further detail in section 5.1.2.1 Optimisation.

Step 5 is the process of initialising StratifiedKFold for cross-validation. This is shown as “`skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=None)`”, where ‘`n_splits`’ is defined as ‘`5`’, ‘`shuffle=True`’ and ‘`random_state=42`’. This means the data is split into roughly five equal sized folds, whilst having a random shuffle to ensure that the data is shuffled before splitting, helping to prevent any underlying order in the data affecting the model’s performance. We also define ‘`random_state=42`’ which allows us to set a specific seed value for the random number generator used by StratifiedKFold for shuffling the data before splitting it into folds. Setting ‘`random_state`’ ensures that the data is shuffled in the same way each time you perform cross-validation, resulting in consistent folds across different runs which is essential for reproducibility when comparing multiple different models or tuning hyperparameters reliably. These parameters will ensure no over fitting occurs which is when a model learns the training data too well, capturing random fluctuations or specific patterns that are unique to the training set. This results in high performance on the training set but poor performance to new unseen data. In other words, an overfitted model learns the training data too intricately, to the point of memorising it rather than learning the underlying relationships and patterns that generalise to other data. Performing cross-validation with k folds helps to prevent overfitting by providing multiple estimates of model performance on different subsets of the data. Please refer to *Figure 4.6* for a visual understanding of the cross-validation process.

Step 6 is the process of initialising lists to store the evaluation metrics for each fold. Without these lists only the metrics computed for the last fold would be displayed which is not what we want as it doesn't allow for a proper comparison of the model's generalisation performance. As a result, aggregated metrics such as standard deviation of accuracy, mean, precision or recall wouldn't be able to be computed which drastically limits comparisons. To have a well-established understanding of the model's performance it's imperative to store the evaluation metrics for each fold in lists. In turn this allows for better assessment of model performance, detection of overfitting and a more reliable comparison against different models or hyperparameters.

Step 7 is the initialisation and start of the overall training time timer. This is shown as '`overall_start_time = time.perf_counter()`'. The purpose with this is to produce the overall time taken to perform stratified 5-folded cross validation. This can provide us with insights into the algorithm's efficiency and complexity, a significantly higher overall training time when compared to a different algorithm would indicate that the algorithm is quite heavy. This would suggest this algorithm would not be suitable for resource constrained devices without comprising performance. Comparisons can be made against multiple algorithms granted that the training occurs in the same way and on the same machine specification.

Step 8 involves the process of performing stratified k-fold cross-validation to train the classifier on the training data, make predictions on the testing data, calculate evaluation metrics, and record the training and testing times.

- Step 8a involved creating a for loop that iterates over each fold generated by the '`StratifiedKFold`' object '`skf`'. This is shown as '`for train_index, test_index in skf.split(X_imputed, y):`'. For each iteration, 'train_index' and 'test_index' contain the indices of the training and testing data points for the current fold. The purpose of this is to generate indices for splitting the dataset into training and testing sets for each fold of the cross-validation process. This ensures evaluation is reliably performed across multiple folds.
- Step 8b involved splitting the data into training and testing sets for each fold. This is shown as '`X_train, X_test = X_imputed[train_index], X_imputed[test_index]`' and '`y_train, y_test = y.iloc[train_index], y.iloc[test_index]`'. Using the indices obtained from '`skf.split()`', this code splits the feature matrix '`X_imputed`' and the label vector '`y`' into training and testing sets for the current fold. '`X_train`' and '`X_test`' contain the feature data for training and testing whilst '`y_train`' and '`y_test`' contain the corresponding labels for training and testing. The purpose of this is to maintain class balance whilst allowing for consistent evaluation of the model's performance.

- Step 8c involved initialising and starting the timer for monitoring the training time for each individual fold. The training time for each fold includes timing the total time taken from training the classifier on the training data, making predictions on the testing data, and calculating evaluation metrics.
- Step 8d involved training the classifier on the training data for the current fold. This is shown as '`clf.fit(X_train, y_train)`', where clf (BalancedRandomForestClassifier) is trained on the training data.
- Step 8e involved initialising and starting the timer for monitoring the testing time for each individual fold. The testing time for each fold includes timing the total time taken from making predictions on the testing data. A quicker testing time will generally indicate a less resource intensive and more efficient algorithm, which will be especially useful for drawing comparisons and conclusions in due course.
- Step 8f involved taking the trained classifier and making predictions on the testing data. This shown as '`y_pred = clf.predict(X_test)`', where '`X_test`' is the testing data and '`clf`' is the BalancedRandomForestClassifier. The prediction result is assigned to the '`y_pred`' variable.
- Step 8g involved ending the timer for testing. This is shown as '`testing_time = time.perf_counter() - testing_start_time`' and '`testing_times.append(testing_time)`', where '`testing_time`' is assigned to the current time minus the '`testing_start_time`' to give us the testing time for the current fold. This is then appended into the '`testing_times`' list which can be used for further analysis.
- Step 8h involved calculating the evaluation metrics. This is shown as '`accuracy_scores.append(accuracy_score(y_test, y_pred))`' and '`confusion_matrices.append(confusion_matrix(y_test, y_pred))`', where the various metrics are computed based on the predicted labels and the true labels ('`y_test`') for the current fold. These results are then appended into their respective lists to keep track of the performance across all folds.
- Step 8i involved ending the timer for training. This is shown as '`training_time = time.perf_counter() - start_time`' and '`training_times.append(training_time)`', where '`training_time`' is assigned to the current time minus the '`start_time`' to give us the training time for the current fold. This is then appended into the '`training_times`' list which can be used for further analysis.

Step 9 is the end of the overall training time timer. This is shown as '`overall_training_time = time.perf_counter() - overall_start_time`'. This has already been discussed in detail within *Step 7*.

Step 10 is the process of printing evaluation metrics which takes the various score metrics previously stored in the lists and applies mean and standard deviation onto them to provide us with the end metrics. This is then followed by printing the testing and training times, calculating the overall confusion matrix utilising the '`matplotlib.pyplot`' and seaborn' s '`sns.heatmap()`' library. Generating the overall classification report using 'sklearn.metrics' library, printing the overall training time, initialising the SHAP explainer with '`shap.TreeExplainer()`' and computing the SHAP values using '`.shap_values()`'. Finally, we plot the SHAP summary plot using the feature names, we produce both a bar plot which is the summary plot and a violin plot which is the feature importance graph.

5.1.2.1 Optimisation

I tested a multitude of different parameters to find out what selection provides the best results. I ended up with these parameters "`BalancedRandomForestClassifier(n_estimators=125, min_samples_split=5, random_state=42)`" performing the best in all regards. The number of estimators means the number of trees in the forest, theoretically a larger number of trees would provide more accurate results at the cost of higher performance. In my case anything over 125 estimators produced negligible results where the accuracy remained the same but increased computational costs.

5.1.3 LightGBM

For the results to be reproducible I have taken the exact same steps as described in the *Balanced Random Forest* section. There are no differences in the code apart from the importation of the 'LGBMClassifier' from the 'lightgbm' library as well as the usage of alternative parameters and parameter values.

5.1.3.1 Optimisation

I tested a multitude of different parameters to find out what selection provides the best results. I ended up with these parameters "`LGBMClassifier(n_estimators=80, class_weight='balanced', learning_rate=0.1, random_state=42)`" performing the best in all regards. The same concept mentioned in *5.1.2.1 Optimisation* can be applied here as well regarding the 'n_estimators'. The 'class_weight' was set to balanced which means the weight assigned to each class is inversely proportional to its frequency in the dataset. Classes with fewer samples are giving higher weights, whilst classes with more samples are given lower weights. This adjustment helps to mitigate the effects of class imbalance, ensuring the model is not biased to the majority class, leading to higher performance for minority classes. This is important for our dataset as we have significantly more non-failures as opposed to failures. The '`learning_rate`' is set to 0.1 which means that the model will update its weights by 0.1 times the gradient at each step during training. A lower learning rate makes the model learn slower but can improve generalisation whilst a higher learning rate will update the weights more aggressively but increases the risk of overshooting the optimal solution which would greatly reduce generalisation performance. We need good generalisation performance so we will need to play around with the learning rate appropriately to find the parameter that produces the best results.

5.1.4 CatBoost

For the results to be reproducible I have taken the exact same steps as described in the *Balanced Random Forest* section. There are no differences in the code apart from the importation of the 'CatBoostClassifier' from the 'catboost' library as well as the usage of alternative parameters and parameter values.

5.1.4.1 Optimisation

I tested a multitude of different parameters to find out what selection provides the best results. I ended up with these parameters "`CatBoostClassifier(iterations=250, auto_class_weights='Balanced', learning_rate=0.2, task_type="GPU", devices='0:1', random_state=42)`" performing the best in all regards. The same concept discussed in *5.1.2.1 Optimisation* can be referenced here regarding the '`iterations`' which is another word for trees or '`n_estimators`'. These other concepts can be referenced from *5.1.3.1 Optimisation*, these include '`auto_class_weights`' and '`learning_rate`'. '`task_type="GPU"`' indicates that I have chosen to utilise GPU (graphics processing unit) acceleration for training, this can significantly reduce the training times especially for larger datasets. CatBoost supports GPU accelerated processing unlike most other algorithms I tested. This could be useful when working with the final solution as CatBoost supports GPU utilisation for inference but it's unknown as of right now the benefits this would provide. Typically, resource constrained devices don't have dedicated GPUs, most tend to have integrated GPUs which are not so powerful.

5.1.5 Decision Tree

For the results to be reproducible I have taken the exact same steps as described in the *Balanced Random Forest* section. There are no differences in the code apart from the importation of the 'DecisionTreeClassifier' from the 'sklearn.tree' library as well as the usage of alternative parameters and parameter values.

5.1.5.1 Optimisation

I tested a multitude of different parameters to find out what selection provides the best results. I ended up with these parameters "`DecisionTreeClassifier(min_samples_leaf=2, min_samples_split=5, max_depth=100, class_weight='balanced', random_state=42)`" performing the best in all regards. The concepts for '`class_weight`' have already been discussed, refer to *5.1.2.1 Optimisation*. '`min_samples_leaf=2`' means that if a node has less than 2 samples the algorithm will stop splitting and consider it as a leaf node (end of a branch). Defining a value prevents the tree from splitting nodes that have very few samples, this controls overfitting whilst promoting simpler tree structures. On the other hand, '`min_samples_split=5`' controls the minimum number of samples required to split an internal node as opposed to a leaf node. Both parameters work together to control the growth and complexity of the decision tree whilst also helping to prevent overfitting. '`max_depth=100`' specifies the maximum depth (layers) of the decision tree. Setting the maximum depth helps to prevent overfitting as the tree can become too complex if it's too deep.

5.1.6 Gradient Boosting

For the results to be reproducible I have taken the exact same steps as described in the *Balanced Random Forest* section. There are no differences in the code apart from the importation of the 'HistGradientBoostingClassifier' from the 'sklearn.ensemble' library as well as the usage of alternative parameters and parameter values.

5.1.6.1 Optimisation

I tested a multitude of different parameters to find out what selection provides the best results. I ended up with these parameters "`HistGradientBoostingClassifier(max_iter=40, class_weight='balanced', learning_rate=0.1, random_state=42)`" performing the best in all regards. The above parameters have already been discussed, please refer to *5.1.3.1 Optimisation* for further reference.

5.1.7 AdaBoost

For the results to be reproducible I have taken the exact same steps as described in the *Balanced Random Forest* section. There are no differences in the code apart from the importation of the 'AdaBoostClassifier' from the 'sklearn.ensemble' library as well as the usage of alternative parameters and parameter values.

5.1.7.1 Optimisation

I tested a multitude of different parameters to find out what selection provides the best results. I ended up with these parameters "`AdaBoostClassifier(n_estimators=50, learning_rate=0.8, random_state=42, algorithm='SAMME.R')`" performing the best in all regards. The above parameters '`n_estimators`', '`learning_rate`' and '`random_state`' have already been discussed, please refer to *5.1.3.1 Optimisation* for further reference. Increasing '`n_estimators`' greatly improves the performance of the AdaBoost ensemble up to a certain point. More estimators usually lead to better performance but increases the computational cost. The key here lies in finding the most efficient median. '`algorithm='SAMME.R'`' refers to the AdaBoost algorithm to use it stands for Stagewise Additive Modelling using a Multiclass Exponential loss function with Real-valued predictions. 'SAMME.R' is an improved version of 'SAMME' that allows for real-valued predictions as opposed to class labels. It also typically converges faster and has the potential to achieve a higher accuracy over its predecessor.

5.1.8 Naïve Bayes

For the results to be reproducible I have taken the exact same steps as described in the *Balanced Random Forest* section. There are no differences in the code apart from the importation of the 'GaussianNB' from the 'sklearn.naive_bayes' library.

5.1.8.1 Optimisation

There isn't any optimisation in terms of parameters for naïve bayes as it's an extremely simple model.

5.2 Chosen Model Training and Exportation

I had decided to choose the LightGBM (Light Gradient Boosting Machine) algorithm as this produced one of the fastest inference times whilst showing very consistent and high accuracy, precision and recall scores amongst the group. This will be further explained in *Chapter 6 Testing and Evaluation*. To export the chosen model for usage in our application, code has been developed which is shown below in Figure 5.2.

```
1. Import necessary libraries

2. Load dataset from file

3. Preprocess data:
   a. Encode categorical variables using LabelEncoder
   b. Convert datetime column to year, month, and day
   c. Drop original datetime column
   d. Splitting data into features and labels
   d. Impute missing values using SimpleImputer

4. Initialize LightGBMClassifier with specified parameters

5. Train the classifier on the training data

6. Save the trained model to a file

7. End
```

Figure 5.2 Pseudocode Showing the Process of Saving LightGBM Trained Model

- Step 1 I imported all the necessary libraries. This included LGBMClassifier from lightgbm, LabelEncoder from sklearn.preprocessing, SimpleImputer from sklearn.impute, Joblib and Pandas.
- Step 2 I loaded the dataset from the CSV file using Pandas to read the CSV file and store the data in a DataFrame.
- Step 3 involved the preprocessing steps undertaken to ensure that the data is in the same format as it was during the multiple model implementation stages. This followed the exact same procedure as described in *Step 3 from Figure 5.1*.
- Step 4 involved initialising the LightGBM Classifier using the parameters listed and explained in *5.1.3.1 Optimisation*.
- Step 5 involved training the classifier on the entire training data. We do not need to perform cross-validation anymore since the data is now the training data in its entirety as opposed to having five folds being split between training and testing data.
- Step 6 involved saving the trained model to a pickle file which is a way to save the trained model object as a file. "`joblib.dump(clf, 'C:/Users/zahid/Documents/csv/lightgbm_model.pkl')`" this is done by using '`joblib.dump()`' to serialise the model object along with metadata such as parameters, into a binary format. The serialised data is then written to a file specified by the user which in our case is saved to as '`lightgbm_model.pkl`'. This pickle file can then be utilised with 'joblib.load()' to reconstruct the trained model and make predictions from new data.

5.3 Application Development and Returning Predictions

There would be many stages involved when producing a working application that can return predictions of hard drive failure for a specified drive using it's SMART values, whilst ensuring that an accurate prediction can be made and shown to the user. The pseudocode listed below in

Figure 5.3 shows the steps taken to successfully develop an application which returns predictions.

```
1. Import necessary libraries

2. Initialise Flask application

3. Load the machine learning model

4. Initialise SimpleImputer to handle missing values

5. Define a function to get SMART data from a device:
    a. Run smartctl command to get SMART data
    b. Save the SMART data to a JSON file
    c. return the SMART data

6. Define a function to convert SMART data from JSON to CSV format:
    a. Load JSON data from a file
    b. Prepare CSV data by extracting specific attributes
    c. Convert the data into a DataFrame
    d. Handle any missing columns with NaN values
    e. Return DataFrame

7. Define route for the homepage:
    a. Return the index.html file

8. Define route for prediction:
    a. Set a predefined device path
    b. Get SMART data and save it to a file
    c. Convert SMART data to DataFrame
    d. Use SimpleImputer to fill missing values in the DataFrame
    e. Sort the DataFrame columns
    f. Set the display options
    g. Extract numbers from column names, convert to integer, and sort
    h. Make prediction using the loaded model
    i. Return prediction in JSON format

9. Run the Flask application
```

Figure 5.3 Pseudocode of Flask Application that Makes Predictions on Drive Failures

Step 1 involved importing the necessary libraries. I have produced a list below which shows all the modules I have imported and the general functionalities they offer.

- ‘`from flask import Flask`’. This import utilises Flask which is a lightweight web application framework. It provides the basic functionality needed to handle HTTP requests and responses as well as being able to route URLs (uniform resource locator) to functions. In my case I have utilised Flask to define routes with corresponding functions, one to handle requests to the root URL “(‘/’)” and serves the ‘`index.html`’ file using the ‘`send_file`’ function. This can be seen as ‘`return send_file('/app/index.html')`’. The other to handle HTTP POST requests used to make predictions, so when the Flask application receives a POST request to the ‘/predict’ endpoint it triggers the ‘predict’ function. This can be seen as "`@app.route('/predict', methods=['POST'])`" followed with '`def predict():`'. The reason I have chosen to use Flask is because it was the most popular solution, I could find which provided the most information regarding the setup as I wanted something quick and efficient.

- '`from flask import jsonify`'. Jsonify is a function provided by Flask which can generate JSON-formatted HTTP responses. It does this by taking in arguments and converts them to JSON strings, whilst setting the appropriate header in the HTTP response to indicate that the response body contains JSON data. I have utilised jsonify in the predict function of my code to convert the prediction result to a JSON-formatted response, returning it as a HTTP response.
- '`from flask import send_file`'. 'send_file' is a way to serve static files from my Flask application to users' browsers. When a user makes a HTTP request to the corresponding URL, Flask sends the file as the response. The user's browser then handles displaying of the contents. This provides the functionality to serve HTML files, text files, etc directly to the user's browser. In my case I have used 'send_file' to serve the 'index.html' so that when a user navigates to the root URL of the flask application Flask triggers the 'index()' function.
- '`import subprocess`'. 'subprocess' is a Python module that provides functionality to create new processes, connect to them and capture the output. In other words, it allows the user to execute system commands or external programs as if you were running them from the command line. I utilised a subprocess in my code to execute a command which would return the current SMART statistics for a predefined drive.
- '`import json`'. 'json' is a Python module that provides functionality for encoding Python objects as JSON strings and decoding them back into Python objects. In my code I have utilised 'json' to decode JSON formatted data from a JSON file into a Python object to work with the data.

Step 2 involved initialising the flask application which is done by simply passing '__name__' as an argument to 'Flask()'. This can be seen as '`app = Flask(__name__)`', where app is the name of the Flask application. This is necessary for Flask to know where to look for static files and other resources associated with the application. For example, further on I define the 'index.html' file as '/app/index.html'.

Step 3 involved loading the machine learning model which we had previously trained and saved. This is done using '`model = joblib.load("lightgbm_model.pkl")`' which reconstructs and assigns the model to the model variable.

Step 4 involved initialising SimpleImputer to handle missing values. As done in our previous code the same mean strategy was utilised.

Step 5 involved the steps undertaken to define a function able to get SMART data from a device.

- In step 5a I had to discover a way to return current SMART values for a specific drive which could be done with a command in the command line interface, so that I can automate this process with code as opposed to having to work with a GUI interface. The solution found was from a package called smartmontools which contained a command called smartctl that can be executed with specified device parameters to produce the current SMART values for the drive. It also offers options to output the file in JSON format which provides flexibility to save the file as a JSON, which is what I ended up utilising as it made the process a lot easier. The function created was '`def get_smart_data(device_path):`', where 'device_path' is the current user defined drive mount point which is declared in the last function of the code, I will explain this further on.
- Step 5b involved capturing the saved results containing all the SMART data and writing it into a JSON file, whilst specifying the save location as '/predictt/smart_data.json'. This save path was assigned to a variable called 'json_file_path' to simplify the process of calling the file for reading during the preprocessing stage.

Step 6 involved the steps undertaken to define a function that can convert SMART data from JSON format into the CSV format my model was trained on. The problem with this is different drives report different SMART statistics, they are not all the same. For example, an SSD (solid-state drive) won't return any SMART 10 values (spin retry count) because it doesn't have a spin up disk mechanism like hard drives do. For my model to return a prediction on an input it must contain the exact same number (in the exact same order) of SMART values it was trained on. The trained model selection of SMART values don't follow any particular order. To work around this, after loading in the JSON data I created a for loop that iterates over multiple if statements specifying the id numbers of the SMART values needed. For example, '`if id == 1:`
`csv_data["Column_" + str(1)] = item["value"]`
`csv_data["Column_" + str(2)] = item["raw"]["value"]`'. This means that new columns are created with a string value specified which is the corresponding column number that the model was trained on and then that column is assigned the value that is named "value" in the JSON file. Based on the example this means for SMART_1 values (id correlates to the SMART value), column_1 is created and assigned the normalised value and column_2 is created and assigned the raw value. This totalled 28 SMART values including SMART 1, 3, 4, 5, 7, 9, 10, 12, 192, 193, 194, 197, 198, 199, where 14 of these were the raw values and the remaining 14 were the corresponding normalised values. if the ID of the appropriate SMART value was present in the JSON file, then it would be assigned to the 'csv_data' list I created under the corresponding index of the column matching it. I created another for loop that checks if all columns are present, if missing values are present, they are filled with NaN (not a number) as LightGBM model accepts NaN values but not blank values. the DataFrame is then returned for later usage.

Step 7 involved defining the route for the homepage which handles requests to the root URL serving the 'index.html' file. This was discussed in more detail within Step 1.

Step 8 involved the steps taken to define the route for prediction. When the user clicks the predict button on the frontend a POST request is sent to the Flask application and the predict function is triggered. Once triggered it starts a chain activating the two functions 'get_smart_data' followed by 'convert_to_csv_format' which were both described previously in steps 5 and 6. As such the SMART data is saved to a file and is pre-processed appropriately. The extracted SMART data returned from the 'convert_to_csv_format' is converted into a DataFrame and SimpleImputer is used to fill any values missing from within the columns. Previously in the 'convert_to_csv_format' function we only filled with NAN values; we can now use SimpleImputer to put estimated values instead as this can sometimes lead to better performance depending on the algorithm. I initially planned to implement multiple models which is why I've used NAN values and SimpleImputer one after the other, as some algorithms cannot handle NAN values natively unlike LightGBM. For simplicity purposes I retained the code this way as I had already built the application. After sorting the column names in ascending order, converting the second part (index 1) into an integer, and then rearranging the columns based on the order. I returned a prediction on the imputed data and returned the prediction as a JSON file using 'jsonify'. The reason I returned the prediction in JSON format is because I used AJAX in the frontend 'index.html' file, AJAX expects the prediction as a JSON object so in order to avoid modifying the client-side code to handle a different format the data was returned as JSON.

Lastly in Step 9 the if statement specifies that if the app is being run directly as opposed to being imported as a module, then start the Flask development server in debug mode for development purposes.

5.4 Application Deployment

The final solution was deployed as a docker container built for Linux/ARM64 devices. This was due to LightGBM unfortunately not supporting 32-bit architecture. To containerise my solution, I would have to build a Docker Image which first requires the creation of a DockerFile specifying the system dependencies that would need to be installed for the application to function. This included installing smartmontools and libgomp1. I would also need to provide a requirements.txt file listing all the necessary packages for the application to function that should also be installed when the user deploys the docker container. This includes flask, joblib, lightgbm, pandas, scikit-learn and schedule. Below I have produced some pseudocode that lists the process of creating the DockerFile.

```
1. Use an official Python runtime as a parent image for ARM architecture

2. Install system dependancies (sudo, libgomp1, smartmontools)
```

```
3. Set the working directory in the container to /app

4. Copy the current directory contents into the container at /app

5. Install any needed packages specified in requirements.txt using pip

6. Make port 5001 available to the world outside the container

7. Define an environment variable FLASK_APP with the value app.py

8. Run the command "flask run --host=0.0.0.0" when the container launches
```
Figure 5.4 Pseudocode of a DockerFile that containerises the Flask application

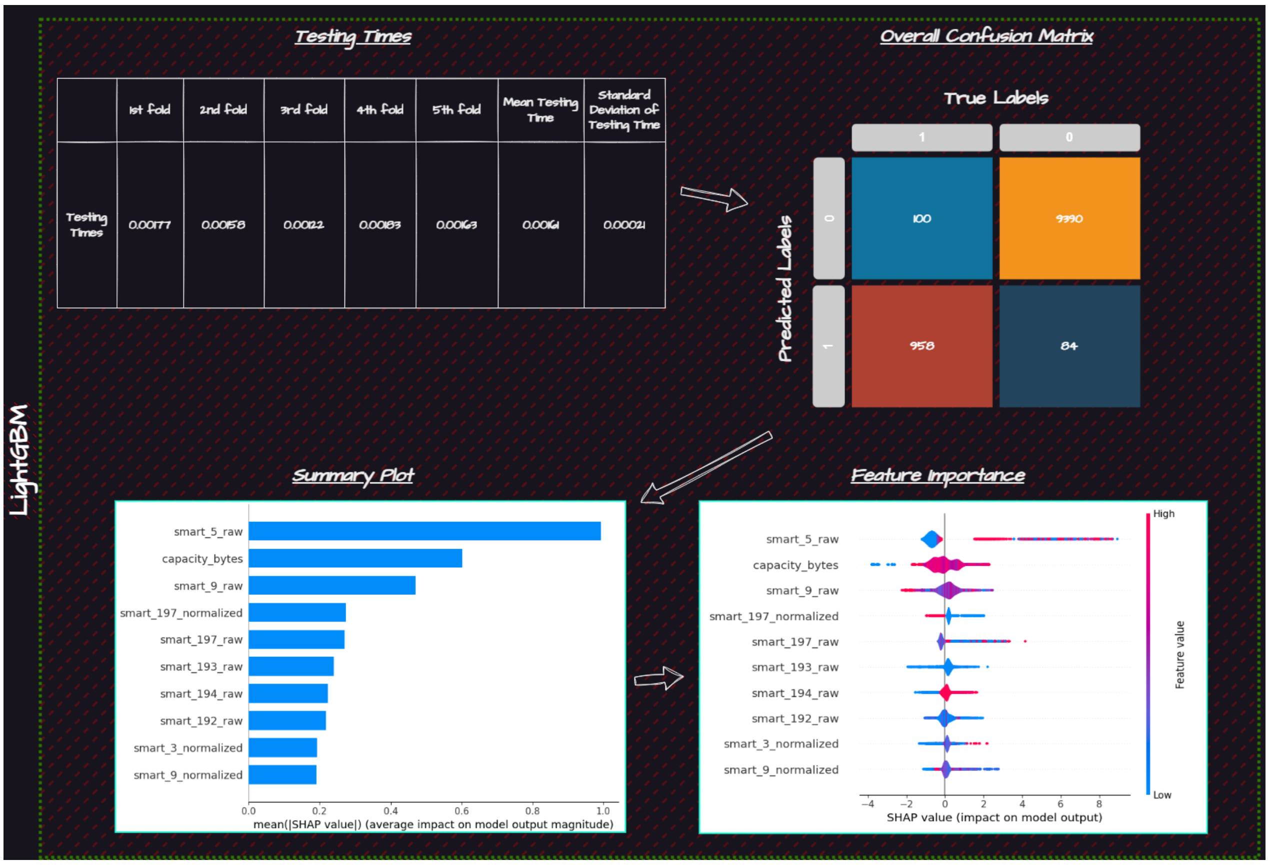
With the DockerFile completed it was time to build the image. This was done by running 'docker build -t disk_failure_prediction .' which builds the Docker image with the ARM architecture tag. I then transferred the Docker image over to my Raspberry Pi 4B using SFTP (Secure File Transfer Protocol) and built the image locally using 'docker load -i disk_failure_prediction.tar'. I then ran the solution using 'docker run --privileged -v hddpredict:/predictt -i --device=/dev/sda -p 5001:5000 disk_failure_prediction:latest'. In this Docker run command I created a new docker volume called /predictt which is where the code stores the JSON file. The reason for this is because there tends to be permission issues when storing it in a normal filesystem location, instead making a Docker volume requires no permissions setup so it's the most straightforward solution. I then specified access to the device which I would like the container to have access to. This now launches the application on the web interface port 5001 which can be accessed locally via the Raspberry Pi's local IP address.

6 Testing and Evaluation

In this chapter I will be presenting the results from the 5.1 Multi Model Implementation section, comparing, and evaluating the results, justifying why I chose LightGBM as the best model. This will be done by comparing the evaluation metrics from all the models and analysing infographics such as summary plots, feature importance graphs and confusion matrices. Finally, I will compare the differences between containerising the solution as a Docker container against native deployment, providing justifications for why I containerised my solution.

6.1 Algorithm Comparisons

All of the metrics generated, model training and docker image building were performed on a desktop computer running Windows 11, AMD Ryzen 7 7800X3D CPU, Nvidia GeForce RTX 3060 GPU, 32GB DDR5 RAM. The final solution was built as a Docker image for Linux/ARM64 on the desktop then was transferred to the Raspberry Pi 4B via SFTP. All the diagrams were produced using draw.io software (JGraph Ltd, 2024, Draw.io, 24.0.7). Algorithms are listed in order from best to worst.



	1st fold	2nd fold	3rd fold	4th fold	5th fold	Mean Testing Time	Standard Deviation of Testing Time
Testing Times	0.00177	0.00158	0.00122	0.00183	0.00163	0.00161	0.00021

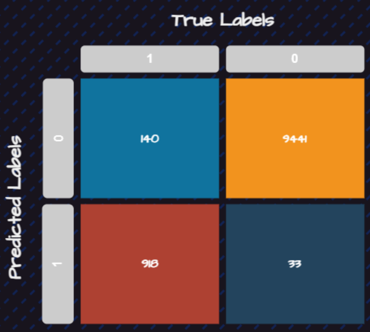
$$\begin{array}{c|cc} & 1 & 0 \\ \hline 0 & 100 & 9390 \\ 1 & 958 & 84 \end{array}$$

Gradient Boosting

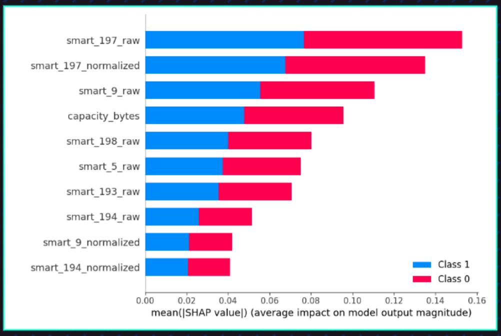
Testing Times

	1st fold	2nd fold	3rd fold	4th fold	5th fold	Mean Testing Time	Standard Deviation of Testing Time
Testing Times	0.00444	0.00470	0.00472	0.00413	0.00408	0.00441	0.00027

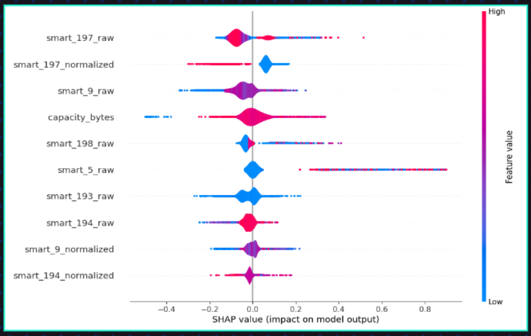
Overall Confusion Matrix



Summary Plot



Feature Importance

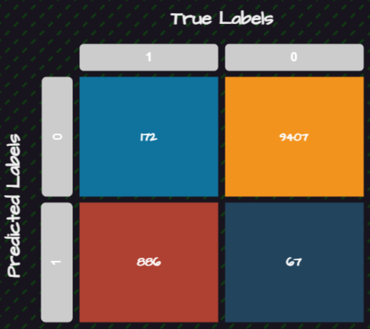


Decision Tree

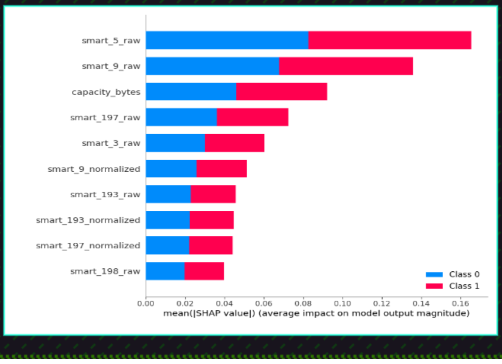
Testing Times

	1st fold	2nd fold	3rd fold	4th fold	5th fold	Mean Testing Time	Standard Deviation of Testing Time
Testing Times	0.00021	0.00021	0.00021	0.00026	0.00022	0.00022	1.92102

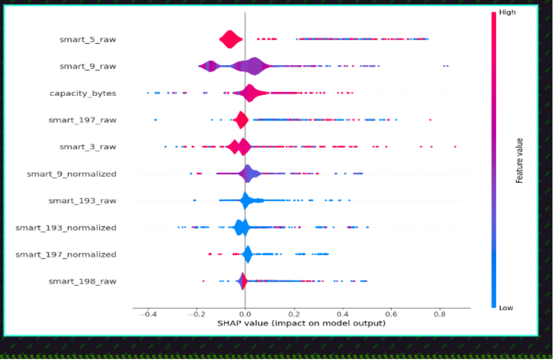
Overall Confusion Matrix

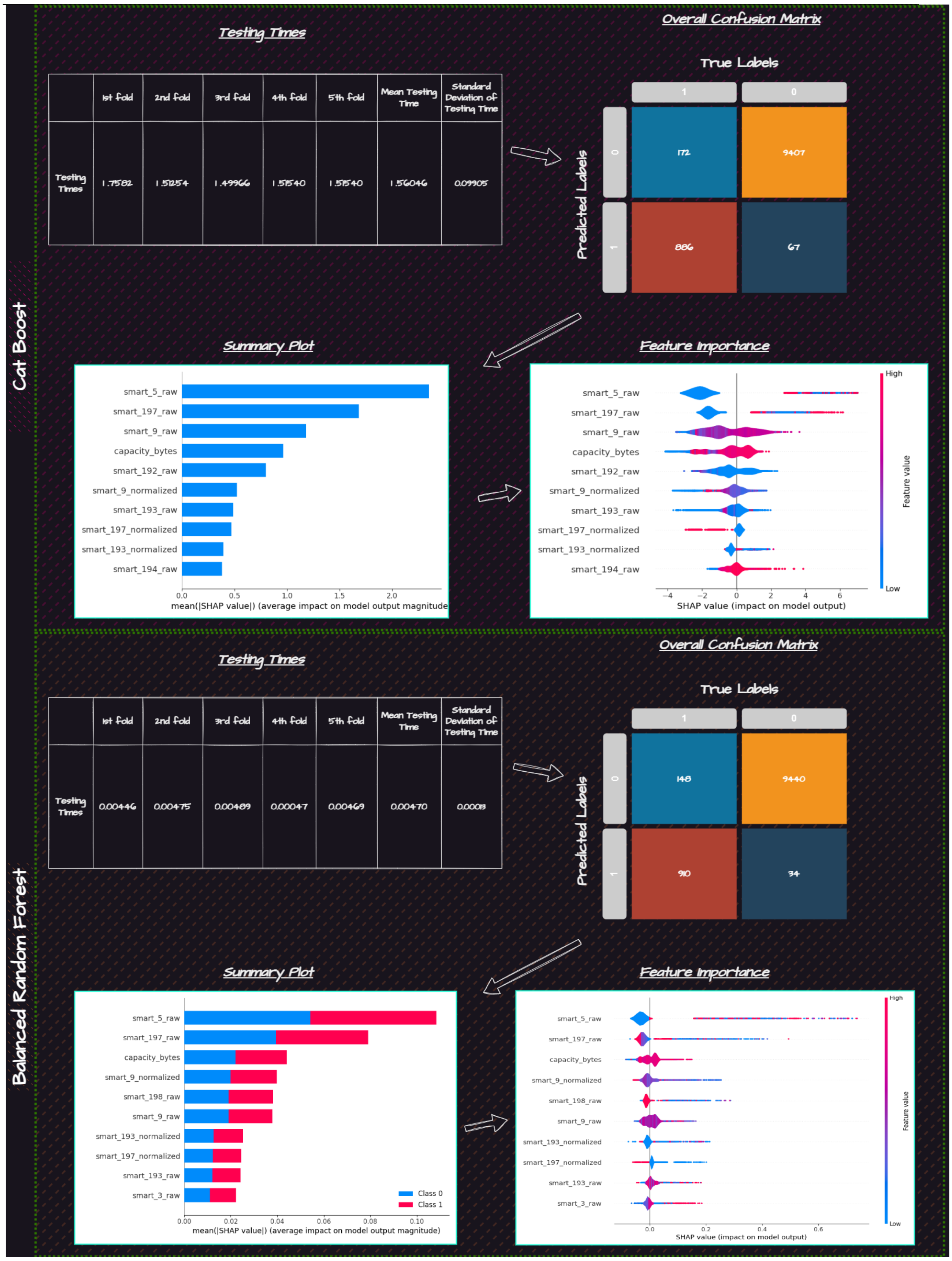


Summary Plot



Feature Importance





Cat Boost

Testing Times

	1st fold	2nd fold	3rd fold	4th fold	5th fold	Mean Testing Time	Standard Deviation of Testing Time
Testing Times	1.7582	1.51254	1.49966	1.51540	1.51540	1.56046	0.09905

Overall Confusion Matrix

	1	0
0	172	9407
1	886	67

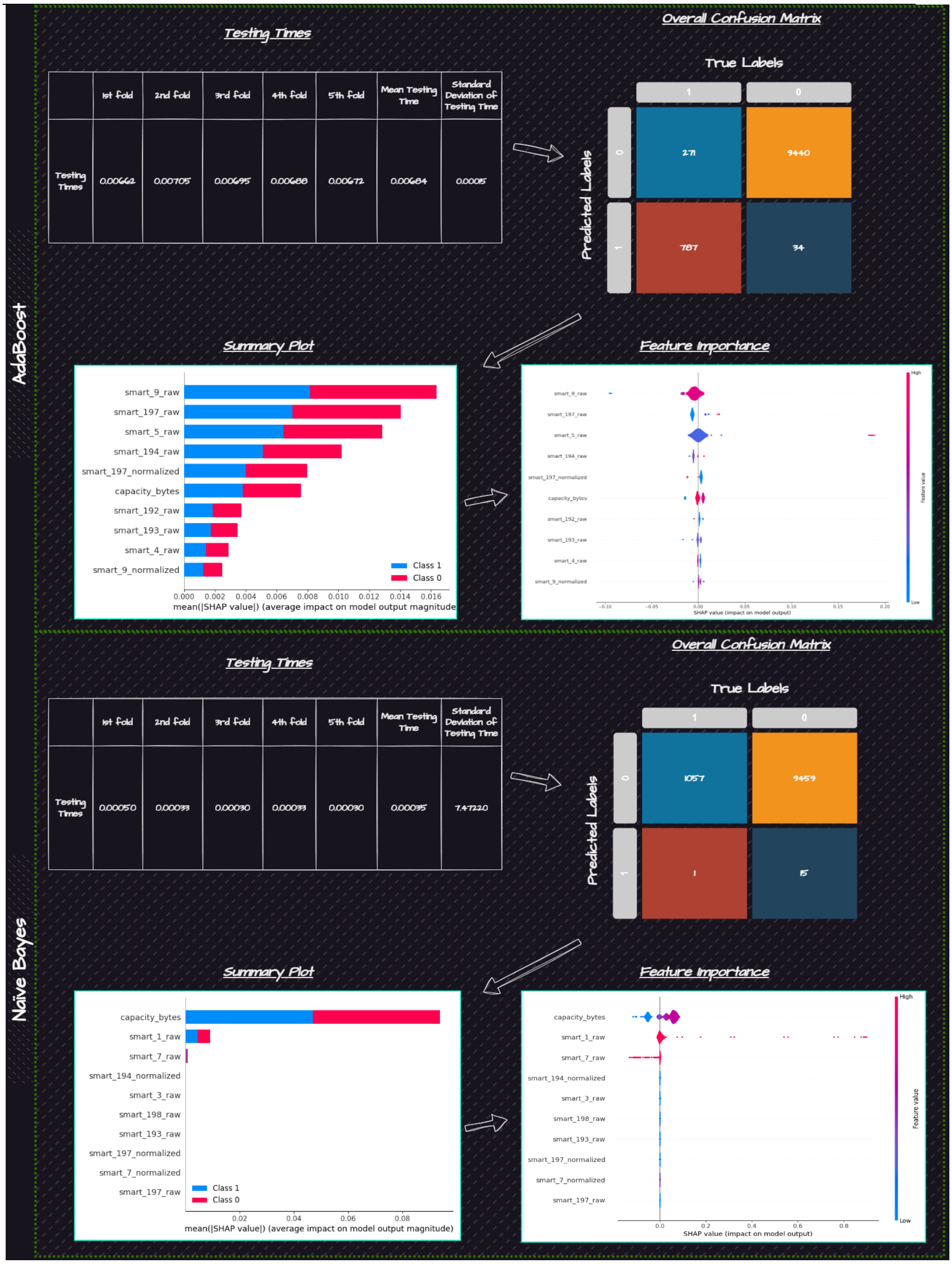
Balanced Random Forest

Testing Times

	1st fold	2nd fold	3rd fold	4th fold	5th fold	Mean Testing Time	Standard Deviation of Testing Time
Testing Times	0.00446	0.00475	0.00489	0.00047	0.00469	0.00470	0.00013

Overall Confusion Matrix

	1	0
0	148	9440
1	910	34



AdaBoost

Testing Times

	1st fold	2nd fold	3rd fold	4th fold	5th fold	Mean Testing Time	Standard Deviation of Testing Time
Testing Times	0.00662	0.00705	0.00695	0.00688	0.00672	0.00684	0.00015

Overall Confusion Matrix

Predicted Labels \ True Labels	1	0
0	271	9440
1	787	34

Naïve Bayes

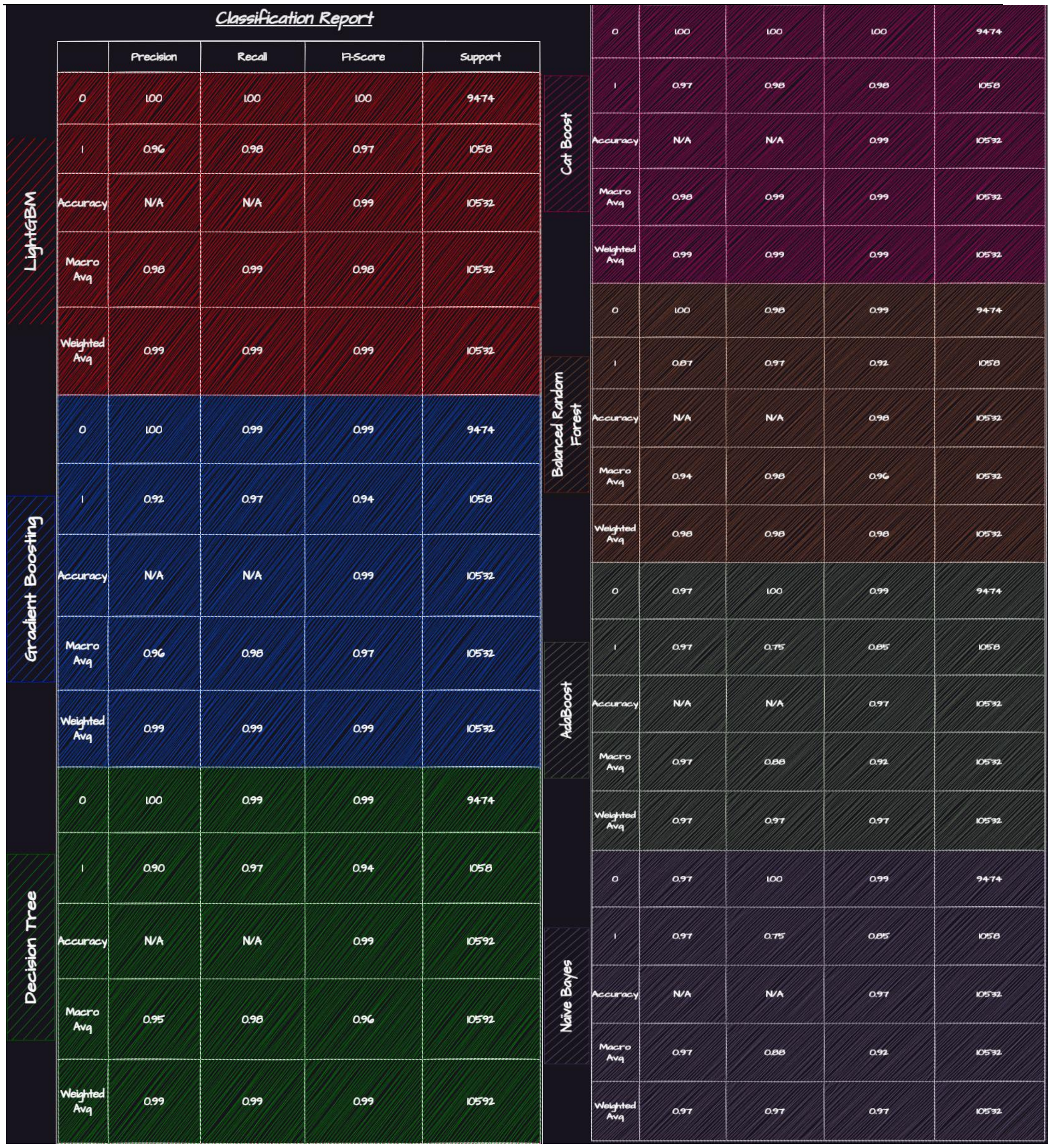
Testing Times

	1st fold	2nd fold	3rd fold	4th fold	5th fold	Mean Testing Time	Standard Deviation of Testing Time
Testing Times	0.00050	0.00033	0.00030	0.00033	0.00030	0.00035	7.47220

Overall Confusion Matrix

Predicted Labels \ True Labels	1	0
0	1057	9459
1	1	15

Figure 6.1 Comparison Between Model Performance Metrics



Classification Report

Model		Precision	Recall	F1-Score	Support
LightGBM	0	1.00	1.00	1.00	9474
	1	0.96	0.98	0.97	1058
	Accuracy	N/A	N/A	0.99	10532
	Macro Avg	0.98	0.99	0.98	10532
	Weighted Avg	0.99	0.99	0.99	10532
Gradient Boosting	0	1.00	0.99	0.99	9474
	1	0.92	0.97	0.94	1058
	Accuracy	N/A	N/A	0.99	10532
	Macro Avg	0.96	0.98	0.97	10532
	Weighted Avg	0.99	0.99	0.99	10532
Decision Tree	0	1.00	0.99	0.99	9474
	1	0.90	0.97	0.94	1058
	Accuracy	N/A	N/A	0.99	10592
	Macro Avg	0.95	0.98	0.96	10592
	Weighted Avg	0.99	0.99	0.99	10592
Cat Boost	0	1.00	1.00	1.00	9474
	1	0.97	0.98	0.98	1058
	Accuracy	N/A	N/A	0.99	10532
	Macro Avg	0.98	0.99	0.99	10532
	Weighted Avg	0.99	0.99	0.99	10532
Balanced Random Forest	0	1.00	0.98	0.99	9474
	1	0.87	0.97	0.92	1058
	Accuracy	N/A	N/A	0.98	10532
	Macro Avg	0.94	0.98	0.96	10532
	Weighted Avg	0.98	0.98	0.98	10532
AdaBoost	0	0.97	1.00	0.99	9474
	1	0.97	0.75	0.85	1058
	Accuracy	N/A	N/A	0.97	10532
	Macro Avg	0.97	0.88	0.92	10532
	Weighted Avg	0.97	0.97	0.97	10532
Naïve Bayes	0	0.97	1.00	0.99	9474
	1	0.97	0.75	0.85	1058
	Accuracy	N/A	N/A	0.97	10532
	Macro Avg	0.97	0.88	0.92	10532
	Weighted Avg	0.97	0.97	0.97	10532

Figure 6.2 Model Classification Report

In Figure 6.1 LightGBM and CatBoost have only positive contributions (blue bar charts) unlike the rest of the algorithms, the reason for this is because of the way these two algorithms work. These algorithms build trees that optimise and improve predictions iteratively, meaning all features either contribute positively or don't contribute.

Based on the information provided from Figure 6.1 LightGBM gets outclassed in inference speed by only Decision Tree and Naïve Bayes. Decision Tree looks to be better than LightGBM from Figure 6.1, but when we consider Figure 6.2, we can clearly see there are some performance differences between the two. LightGBM has 3x 100 scores when predicting non failure values (0). When predicting failure values (1) it has 0.96 for precision, 0.98 for recall and 0.97 for F1-score. On the other hand, when predicting non failure values Decision Tree only achieves 1x 100 score, granted the other two are 0.99 which is a very insignificant difference. When predicting failures (1) which is what our intended use case for the model is, Decision Tree falls short of LightGBM achieving 0.90 for precision, 0.97 for recall and 0.94 for F1-Score. These are still very good results but I believe sacrificing 0.00139s of inference speed is worthwhile as LightGBM makes up for it by achieving 0.06 better precision, 0.01 better recall and 0.03 better F1-score. Naïve bayes performs very poorly overall with the only thing going for it is its inference time which is expected since it's such a simple algorithm. There is not much purpose in discussing this algorithm any further since LightGBM outclasses it significantly. Gradient Boosting is a solid performing algorithm being comparable in performance to Decision Tree but falls short in its inference time, having an average inference time of 0.00441 which is 0.0028s slower than LightGBM whilst also performing slightly worse in precision, recall and F1-scores. CatBoost mostly outperforms LightGBM in all the tests but has significantly inferior inference times which are 1.55885s slower, this trade-off is not justifiable at all. Balanced random forest falls short of LightGBM in all aspects. AdaBoost has 0.01 better precision than LightGBM for predicting failed drives and performs the same as naïve bayes but with an inference time 0.00523s slower than LightGBM. Both algorithms lack recall and F1-scores to be considered.

6.2 Docker Versus Native Deployment

Docker containerisation provides many advantages and disadvantages over native deployment and greatly depends on what you plan to implement. In my case containerising my application would provide an easier to manage instance of the application due to the way Docker works. Docker containers run in isolation from each other. This means they don't require a full operating system to run off, they instead share the host system's OS kernel, providing potential to be less resource intensive. For example, when running the application as a docker container all you would need to do is pull the image and build it with the correct docker compose arguments relative to your needs, the only disadvantage with this is someone with no experience of Docker would potentially struggle to deploy the solution correctly. Native deployment on the other hand you would need all the appropriate files with an identical folder structure for the solution to not throw any errors and if you wanted to change the folder structure you would need to modify the code appropriately. There are more ways that the setup can go wrong and less flexibility when deploying natively compared to containerising the solution. In my case the performance between both instances were identical so I cannot make any additional evaluations regarding docker's performance other than it simply doesn't reduce performance any further over native deployment.

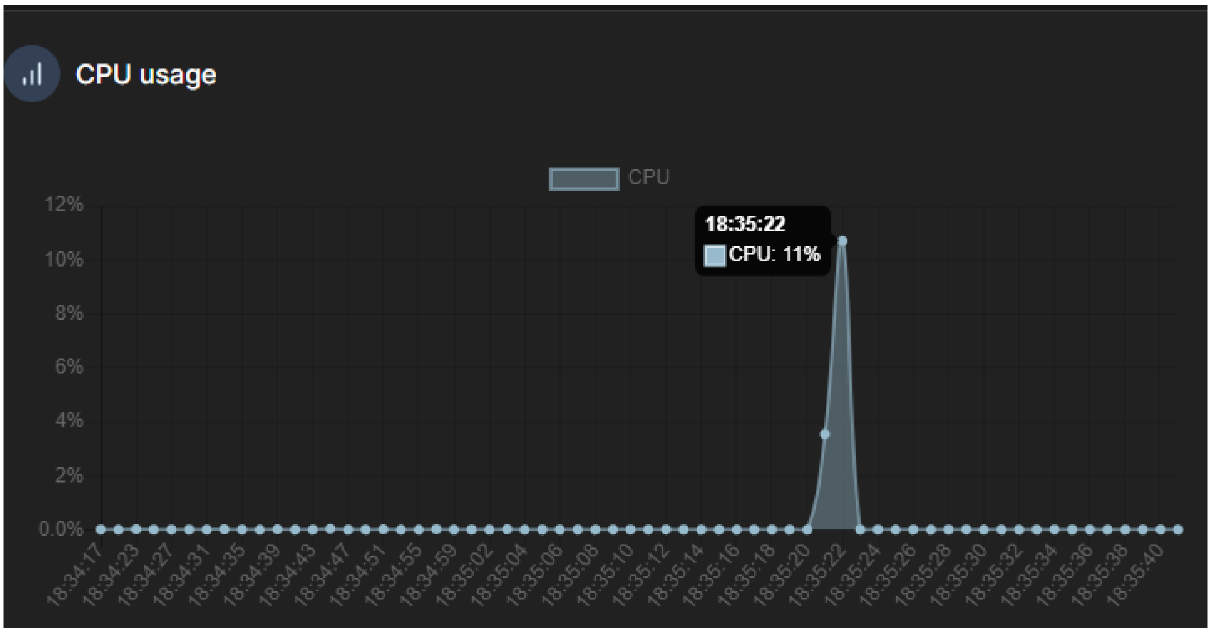
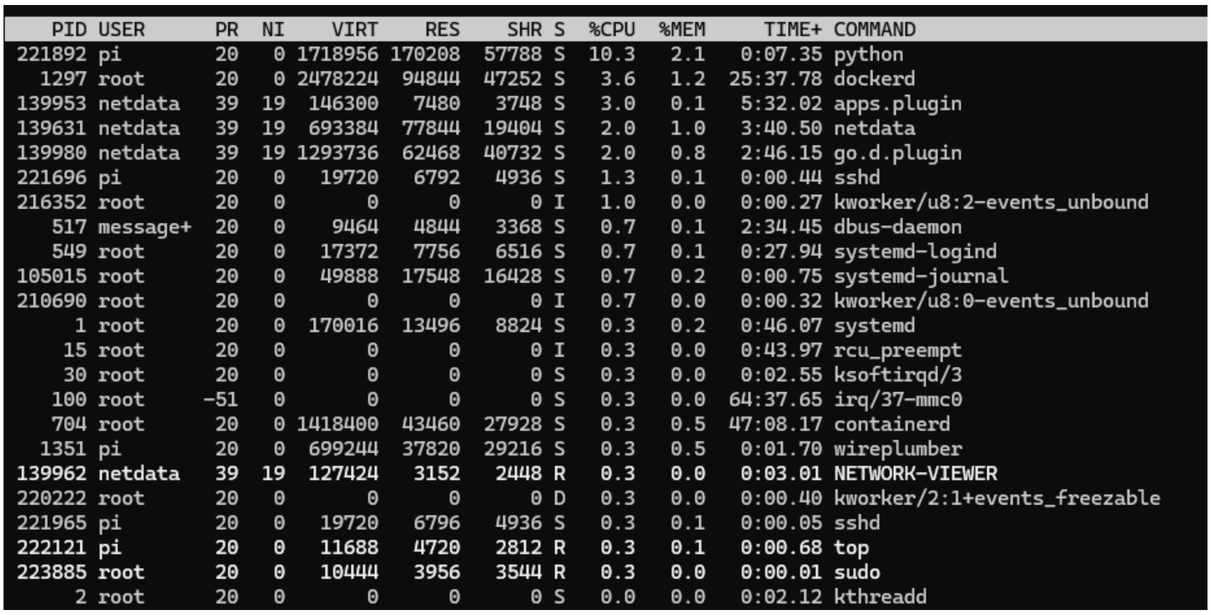


Figure 6.3 Docker Deployment Resource Usage Graph



```
    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 221892 pi        20   0 1718956 170208  57788 S  10.3   2.1   0:07.35 python
   1297 root      20   0 2478224  94844  47252 S   3.6   1.2  25:37.78 dockerd
 139953 netdata   39  19  146300   7480   3748 S   3.0   0.1   5:32.02 apps.plugin
 139631 netdata   39  19  693384  77844  19404 S   2.0   1.0   3:40.50 netdata
 139980 netdata   39  19 1293736  62468  40732 S   2.0   0.8   2:46.15 go.d.plugin
 221696 pi        20   0   19720   6792   4936 S   1.3   0.1   0:00.44 sshd
 216352 root      20   0       0      0      0 I   1.0   0.0   0:00.27 kworker/u8:2-events_unbound
    517 message+  20   0    9464   4844   3368 S   0.7   0.1   2:34.45 dbus-daemon
    549 root      20   0   17372   7756   6516 S   0.7   0.1   0:27.94 systemd-logind
 105015 root      20   0   49888  17548  16428 S   0.7   0.2   0:00.75 systemd-journal
 210690 root      20   0       0      0      0 I   0.7   0.0   0:00.32 kworker/u8:0-events_unbound
      1 root      20   0  170016  13496   8824 S   0.3   0.2   0:46.07 systemd
     15 root      20   0       0      0      0 I   0.3   0.0   0:43.97 rcu_preempt
     30 root      20   0       0      0      0 S   0.3   0.0   0:02.55 ksoftirqd/3
    100 root     -51   0       0      0      0 S   0.3   0.0  64:37.65 irq/37-mmc0
    704 root      20   0 1418400  43460  27928 S   0.3   0.5  47:08.17 containerd
   1351 pi        20   0  699244  37820  29216 S   0.3   0.5   0:01.70 wireplumber
 139962 netdata   39  19  127424   3152   2448 R   0.3   0.0   0:03.01 NETWORK-VIEWER
 220222 root      20   0       0      0      0 D   0.3   0.0   0:00.40 kworker/2:1+events_freezable
 221965 pi        20   0   19720   6796   4936 S   0.3   0.1   0:00.05 sshd
 222121 pi        20   0   11688   4720   2812 R   0.3   0.1   0:00.68 top
 223885 root      20   0   10444   3956   3544 R   0.3   0.0   0:00.01 sudo
      2 root      20   0       0      0      0 S   0.0   0.0   0:02.12 kthreadd
```

Figure 6.4 Native Deployment Linux Resource Usage List

7 Conclusions

In this section I will be summarising how I have met each objective and concluding if I have met my overall aim. Justifications will be provided.

Objective 1 has been met as I explored a lot of up to date and older literature in great depth, I identified the main points of the research, what results they discovered and how that relates to my topic. Any common trends can easily be identified from the research, existing datasets as well as some preprocessing techniques.

Objective 2 was met as I found a recent extremely raw dataset which contained data on hard drive failures and non-failures for the same drives every day over a six-month period.

Objectives 3 and 4 were both met as I was able to conduct a fair comparison and evaluation between six different models in a controlled environment whilst providing the same infographics allowing for a comparable analysis between different models.

Objective 5 was met as I successfully built an application that utilises the trained model to make predictions on SMART statistics from connected data storage devices then displayed the predictions in a simple web based graphical user interface.

Objective 6 was met as I explored both native and Docker solutions and compared some of the advantages and disadvantages of the two, justifying why Docker was more user friendly and efficient.

In conclusion I have successfully met my overall aim to an extent but there are numerous weaknesses in my approach which could be refined further. This can be justified as I have successfully found the most efficient algorithm for resource constrained devices, have trained the model to be able to predict hard drive failures and have successfully containerised the solution as a docker container for further optimisation.

7.1 Future Work

Further work can be done to get a more efficient solution such as pruning and compression. Key recommendations include further exploration of model optimisation techniques, such as pruning and quantisation, to enhance computational efficiency without compromising predictive performance. Storing the json output into memory rather than to an actual file on the filesystem would be faster and more efficient. Removing either NaN values or imputation but not both, since LightGBM can handle missing values as opposed to blank values.

References

G. F. Hughes, J. F. Murray, K. Kreutz-Delgado and C. Elkan (2002) 'Improved disk-drive failure warnings', *IEEE Transactions on Reliability,* 51(3), pp. 350-357 Available at: 10.1109/TR.2002.802886

Open-Source Community (2024) OpenRefine (3.7.1) [Computer program]. Available at: https://openrefine.org (Accessed: 20/01/24).

JGraph Ltd (2024) Draw.io (24.0.7) [Computer program]. Available at: https://www.drawio.com (Accessed: 19/03/24).

J. Zeng, R. Ba, Q. Chen, L. Wu, H. Wang and Y. Xiong (2022) *Prediction of Hard Drive Failures for Data Center Based on LightGBM.* pp. 105

BackBlaze Hard Drive and Stats. (2023) Available at: https://www.backblaze.com/cloud-storage/resources/hard-drive-test-data (Accessed:

R. Xu, X. Wang and J. Wu (2022) *Classification Based Hard Disk Drive Failure Prediction: Methodologies, Performance Evaluation and Comparison.* pp. 189

Yazici, M.T., Basurra, S. and Gaber, M.M. (2018) 'Edge Machine Learning: Enabling Smart Internet of Things Applications', *Big Data and Cognitive Computing,* 2(3) Available at: 10.3390/bdcc2030026

Choudhary, T., Mishra, V., Goswami, A. and Sarangapani, J. (2020) 'A comprehensive survey on model compression and acceleration', *Artificial Intelligence Review,* 53(7), pp. 5113-5155 Available at: 10.1007/s10462-020-09816-7

Đurašević, S. and Đorđević, B. 'Anomaly detection using SMART indicators for hard disk drive failure prediction',

Pitakrat, T., Van Hoorn, A. and Grunske, L. (2013) *A comparison of machine learning algorithms for proactive hard disk drive failure detection.* pp. 1

Ghassen Khaled (2023) *Hard Drive Data Kaggle.* Available at: https://www.kaggle.com/datasets/ghassenkhaled/hard-drive-data-2023/data (Accessed:

Open Source Community. (2024) *OpenRefine* [0]. Available at: https://openrefine.org (Downloaded: 04/02/24)

Demsar J, Curk T, Erjavec A, Gorup C, Hocevar T, Milutinovic M, Mozina M, Polajnar M, Toplak M, Staric A, Stajdohar M, Umek L, Zagar L, Zbontar J, Zitnik M, Zupan B (2013) 'Orange: Data Mining Toolbox in Python', , pp. 2349-2353 Available at: 10.5555/2567709.2567736

Pinheiro Eduardo, Weber Wolf-Dietrich and Barroso Andre Luiz (2007) 'Failure Trends in a Large Disk Drive Population',

Appendix A Personal Reflection

A.1 Reflection on Project

Coming into this project I had no previous experience with any form of artificial intelligence other than knowing how a few popular algorithms worked such as Decision Trees. If I had my time again, I would look to prioritise the data preprocessing stages as these took significantly longer than expected. From my discussions with my tutor and based off my own personal research I knew the preprocessing was one of the toughest and longest stages but didn’t truly realise the extent of it until I reached this point. This would grant me additional time to dedicate to potentially being able to prune or compress my algorithm to be less resource intensive but still retaining the same performance.

A.2 Personal Reflection

If I had my time again, I would look to start organising, researching, and working on my project much earlier instead of working longer hours all at once even though I prefer this. For this magnitude of a project, it’s not ideal and I would’ve most likely produced a higher quality report and results.

Appendix B Ethics Documentation

B.1 Ethics Confirmation

PDF

breo application approval.pdf

Appendix C Other Appendices