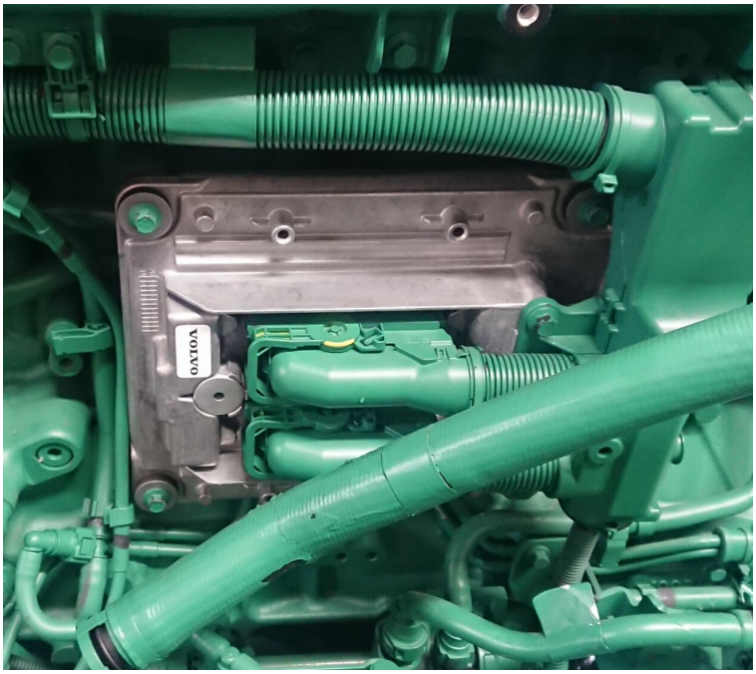

Remaining useful life classification of ECUs in trucks using a transformer encoder model

Master's thesis in Computer Science and Engineering

FREDRIK NYSTRÖM
AXEL SIWMARK

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

Remaining useful life classification of ECUs in trucks using a transformer encoder model

Fredrik Nyström & Axel Siwmark

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Remaining useful life classification of ECUs in trucks using a transformer encoder model

Fredrik Nyström Axel Siwmark

Supervisor: Oana Geman, Department of Computer Science and Engineering
Advisors: Gilberto Hishida, Caio Alves and Shima Saadatimolaee, Volvo Group
Examiner: Robin Adams, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: An electronic control unit component for a Volvo combustion engine.

Typeset in LaTeX
Gothenburg, Sweden 2025

Remaining useful life classification of ECUs in trucks using a transformer encoder model
Fredrik Nyström & Axel Siwmark
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Originally developed for natural language processing, transformer models have achieved state-of-the-art results in tasks such as machine translation and text classification. This has led to increasing interest in applying the transformer architecture to sequential data across multiple other domains.

This thesis takes a binary classification approach to investigate whether a transformer encoder model can be used to classify the remaining useful life of electronic control units (ECUs) in Volvo trucks. The model is trained on operational data and faults related to the ECU, to predict whether an ECU is likely to fail within the following three years.

The performance of the transformer model is evaluated against traditional machine learning classifiers, including logistic regression, LGBM, Extra Trees, and Random Forest. In addition to standard metrics, a custom cost metric is introduced to reflect the real-world impact of false positives and false negatives.

Results show that the transformer encoder outperforms traditional models across all evaluation metrics, particularly when used with ensemble methods. However, the transformer encoder still underperformed compared to a naive classifier on the custom cost metric. This work serves as a starting point for improving the decision-making process in ECU refurbishment.

Keywords: electronic control unit, machine learning, remaining useful life, transformer, Volvo.

Acknowledgements

We are grateful to our supervisor, Oana Geman, for her guidance throughout the thesis. We would also like to thank our supervisors at Volvo, Gilberto Hishida, Caio Alves, and Shima Saadatimolaee for their valuable support. Additionally, we extend our thanks to everyone at E&E for their friendliness and the enjoyable fikas.

Finally, we acknowledge ChatGPT and Grammarly which was used for proofreading and grammar correction during the writing process.

Fredrik Nyström & Axel Siwmark

Contents

List of Figures

List of Tables

List of Abbreviations

C-MAPSS - Commercial Modular Aero-Propulsion System Simulation

CNN - Convolutional Neural Network

ECU - Electronic Control Unit

GRU - Gated Recurrent Unit

LGBM - Light Gradient Boosting Machine

LSTM - Long Short-Term Memory

MLP - Multi Layer Perceptron

MICE - Multiple Imputation by Chained Equations

RNN - Recurrent Neural Network

RUL - Remaining Useful Life

RVR - Relevance Vector Regression

SVR - Support Vector Regression

1

Introduction

Remaining Useful Life (RUL) is a key metric in data-driven methods used to evaluate the health of components. It is applied in various domains, including battery health monitoring, fleet vehicle maintenance, and aircraft engine diagnostics [1][2][3][4][5]. In recent years, interest in data-driven approaches for RUL prediction has been steadily increasing. This trend is largely driven by the advancements of sensor and measurement technologies along with wireless data collection [6].

The automotive industry is an industry that generates large amounts of data due to its connected vehicles. Volvo, a Swedish truck manufacturer with substantial data resources, is exploring data-driven solutions for RUL prediction of truck components. Modern Volvo trucks consist of many interconnected components. One of the most critical components is the engine Electronic Control Unit (ECU), also known as the Engine Control Module (ECM), which will be referred to as the ECU for the remainder of this thesis. The ECU controls various functions, including the electronic input and output of the engine.

When an unknown error occurs in a truck, the ECU is removed for assessment. This assessment involves performing multiple tests on the ECU, such as sending current through it. If the ECU passes these tests it may be refurbished and reinstalled in a truck, otherwise it is discarded. Volvo is not only interested in determining whether a refurbished ECU is reusable but also interested in predicting its RUL. Accurate RUL prediction helps determine whether reinstalling the ECU is a reasonable choice, leading to cost savings through reduced maintenance and fewer breakdowns resulting in less downtime.

1.1 Related Work

Numerous studies have been conducted on predicting the RUL of components and systems, typically categorized into physics-based, experimental, and data-driven methods. While physics-based methods can perform well, they are often difficult to develop for complex environments. Experimental methods are costly and may have scaling issues, limiting their real-world relevance. Therefore, this work focuses on data-driven methods, which are flexible and can use the available information more completely [7].

Data-driven methods for RUL prediction have been mainly oriented around deep learning methods such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) since they work well on time series data. Multiple papers and models for RUL prediction have been evaluated on the Commercial Modular Aero-Propulsion System Simulation (C-MAPSS) datasets which are simulated datasets created by NASA [8]. One example is the paper by Sateesh Babu et al. [2], which showed that a novel CNN could outperform other models such as Multi-Layer Perceptron (MLP), Support Vector Regression (SVR) and Relevance Vector Regression (RVR). In a study by Li et al. [3] a CNN with four convolutional layers and dropout was also evaluated on the C-MAPSS datasets and outperformed several other models, including Long Short-Term Memory (LSTM) and Random Forest. Multiple studies have explored the application of RNNs and their variants such as LSTM and Gated Recurrent Units (GRU), for RUL prediction. In the paper by Cheng et al. [9] an ensemble learning approach of LSTMs was evaluated on the C-MAPSS datasets. Their ensemble learning LSTM model outperformed both simple models, such as SVR, and RVR, but also more advanced architectures, including deep CNNs.

Models like RNNs and LSTM networks have commonly been used for time series tasks such as RUL prediction. In 2017, a new architecture called the Transformer was introduced by Vaswani et al. [10]. Originally developed for natural language processing, the Transformer has achieved great success in that domain. This has led to growing interest in applying transformers to other areas, including regression problems like RUL prediction. In the paper by Chen et al. [4] a transformer model was used to predict the RUL of lithium-ion batteries. The transformer outperformed several other state-of-the-art models such as LSTM and GRU. In a recent study by Ogunfowora and Najjaran [5] it is shown that an encoder-transformer model with an expanding window method as data preparation also outperforms state-of-the-art models on the C-MAPSS datasets. In the paper by Zerveas et al. [11], a transformer encoder architecture was applied to several multivariate time series datasets from different domains. They demonstrated that it often outperformed other methods for both regression and classification tasks, even with datasets containing only a few hundred training samples. For classification tasks, the model showed an average accuracy improvement of 2-25 percentage points over other models across eleven datasets.

In 2024, Scania, a Swedish truck company, released SCANIA Component X Dataset [1] which is a real-world multivariate time series dataset for predictive maintenance and RUL prediction. This dataset was released to establish a benchmark for predictive maintenance and support reproducible research. The dataset used in this work was developed based on inspiration from the SCANIA Component X Dataset.

Recent studies have shown that the encoder-transformer performs well on RUL prediction tasks [4][5][11], making it the main focus of this work. This study takes a classification approach to predicting the RUL of ECUs using time-sequential vehicle data from Volvo. To evaluate the performance of the encoder-transformer model,

traditional machine learning methods such as Logistic Regression and tree-based ensemble models will also be used for comparison.

1.2 Purpose

The purpose of this thesis is to develop and evaluate a transformer-based machine learning model for predicting the RUL of ECUs. Along the way to this objective, three subgoals will be addressed.

- **SG1**: Evaluate and compare the transformer encoder model against traditional machine learning models.
- **SG2**: Optimize model performance by fine-tuning the hyperparameters of a transformer encoder model.
- **SG3**: Explore if ensemble methods can improve the transformer encoder model's performance.

1.3 Scope

In order to make the data comparable, only data from the European region were considered. This also ensured similar regulatory standards for the trucks. To further narrow the scope, the data used were from heavy-duty trucks with the same ECU family. Additionally, due to time constraints, the number of models explored was limited to primarily focus on the transformer encoder model. To provide a benchmark, four traditional machine learning models were included for baseline comparison.

1.4 Outline of Report

The structure of the remainder of this thesis is organized as follows: Chapter 2 presents the relevant theory. Chapter 3 describes the approach and methodology. The results are presented in Chapter 4. Finally, Chapter 5 provides a discussion and conclusion of the results, along with suggestions for future work.

2

Theory

This chapter provides information about the theoretical foundations and models used throughout this thesis. It begins by describing several baseline classifiers, including Logistic Regression, Random Forest, Extra Trees, and Light Gradient Boosting Machine. After that, the Transformer architecture is introduced. This is followed by an overview of regularization techniques commonly used in neural networks to prevent overfitting and improve generalization. Additionally, a description of the Multiple Imputation by Chained Equations method used for handling missing data is presented. Finally, the evaluation metrics used to assess model performance are defined.

2.1 Baseline Models

The baseline models in this thesis are standard machine learning classifiers that serve as reference points for evaluating the performance of the more advanced transformer model. All the baseline models except logistic regression use decision trees. A decision tree is a structure where, at each node, a decision is made based on the value of a specific feature. This decision determines which branch to follow next as the input vector moves down the tree. The class is assigned to the input vector when it reaches the leaf node. All the tree-based models are also ensemble models, meaning that they build and combine multiple trees.

2.1.1 Logistic Regression

When the dependent variable is binary, as in this thesis, binary logistic regression is employed. The primary objective in binary logistic regression is to estimate the probability that the dependent variable Y equals 1, given a set of independent input features. The input features form a linear combination (see Equation 2.1), where each feature x has a β that controls its impact, additionally, a bias term β_0 is added. This linear combination is then fed into the Sigmoid function (see Equation 2.2), which transforms the linear combination of the input features into a probability between 0 and 1, see Figure 2.1.

$$z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n \tag{2.1}$$

$$f(z) = \frac{1}{1 + e^{-z}} \tag{2.2}$$

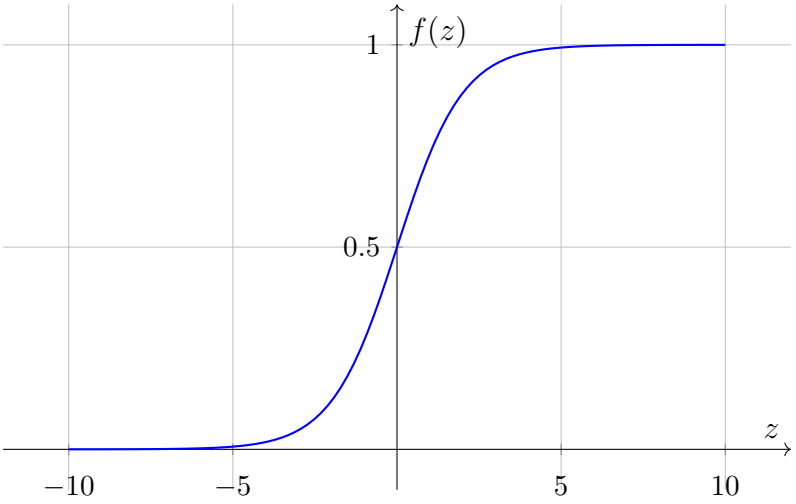

Figure 2.1: Plot of the sigmoid function $f(z) = \frac{1}{1+e^{-z}}$

The parameters $\beta_0, \beta_1, \ldots, \beta_n$ are learned from the training data. They are typically initialized randomly or with small values and then updated through an optimization process that seeks to maximize the likelihood of the observed outcomes. Finally, applying the sigmoid function to z yields the estimated probability that the dependent variable Y equals 1, as shown in Equation 2.3. This probability is then what determines the predicted class.

$$P(Y = 1 \mid x_1, x_2, \ldots, x_n) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n)}} \tag{2.3}$$

2.1.2 Random Forest Classifier

Random Forest is an ensemble learning method that builds multiple decision trees and combines their outputs to make a final prediction [12]. It introduces two sources of randomness to increase the diversity of the trees. The first source of randomness is that it uses bootstrap sampling which means that each tree is trained on a randomly selected subset of the entire dataset, as shown in Figure 2.2. The second source of randomness is that a random subset of features, rather than all the features, is considered when splitting nodes during tree construction. The split is determined by selecting the feature and threshold that best separates the data, minimizing impurity within the resulting child nodes. Impurity is a measure of how mixed the class labels are within a node. A node is pure if all samples in it belong to a single class and considered impure if it contains samples of multiple classes. A common impurity measure for classification is Gini impurity [13]. Given C distinct classes, and $p(i)$ representing the probability of selecting a data point at a child node belonging to class i, the Gini impurity is defined as shown in Equation 2.4.

$$G = \sum_{i=1}^{C} p(i) \cdot (1 - p(i)) \tag{2.4}$$

By minimizing impurity at each split, the decision tree becomes more effective at separating the data into homogeneous groups. Splitting typically continues until a predefined maximum depth of the tree is reached or other stopping criteria are met, such as a minimum number of samples per node.

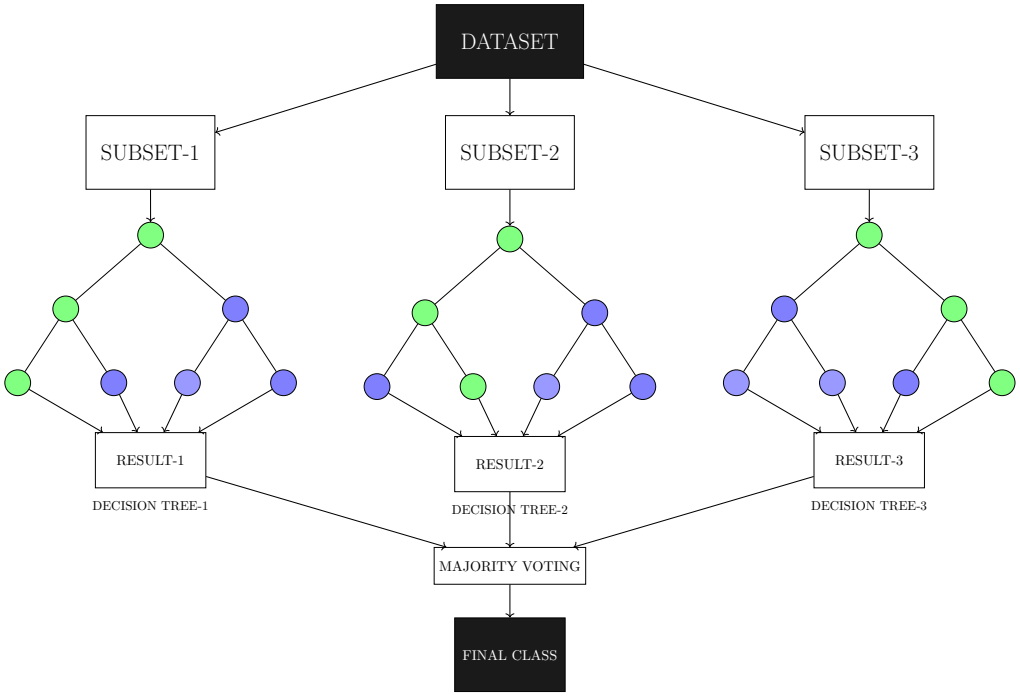

Figure 2.2: Random forest architecture with three decision trees were each tree's result is combined with majority voting resulting in the final class prediction.

2.1.3 Extremely Randomized Trees Classifier

Extremely Randomized Trees (Extra Trees) shown in Figure 2.3 is also an ensemble learning method based on decision trees [14]. Similarly to Random Forest it builds multiple trees and aggregates their predictions, for example through majority voting. Extra Trees introduces additional randomness into the tree-building process and differs from traditional decision tree ensembles in two key ways. Firstly, unlike Random Forest which uses random subsets of the data to grow each tree (as shown in Figure 2.2), this method uses the entire original dataset for each tree. Secondly, during splitting Extra Trees selects the splitting threshold at random compared to Random Forest which determines the optimal threshold to minimize impurity. This randomness increases diversity among the trees and can lead to better performance.

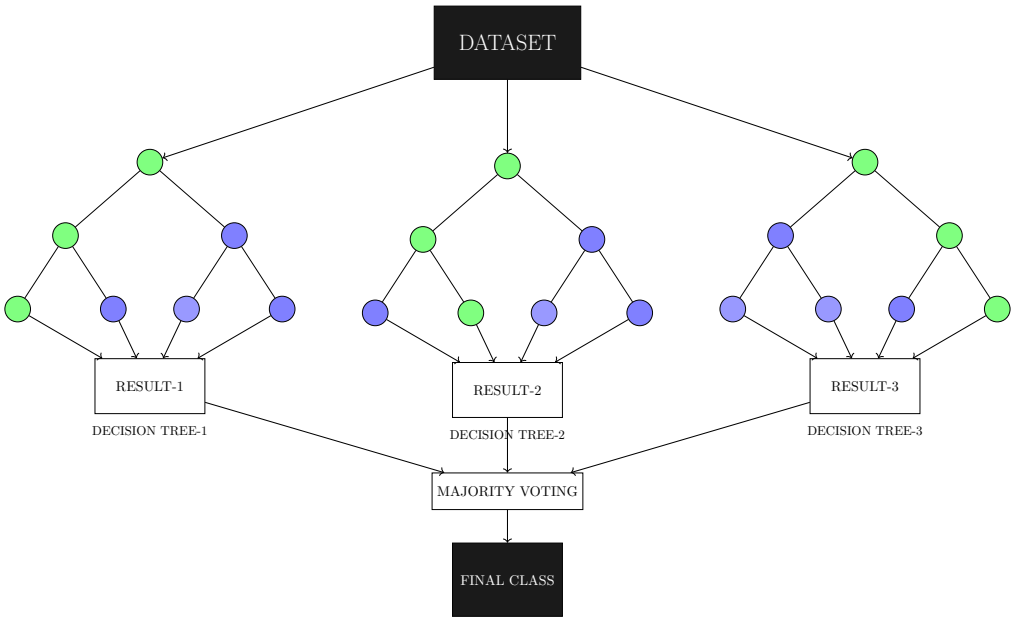

Figure 2.3: Extra trees architecture with three decision trees using the entire dataset, and predictions aggregated through majority voting.

2.1.4 Light Gradient Boosting Machine Classifier

Light Gradient Boosting Machine (LGBM) is a Gradient Boosting Decision Tree model developed by Microsoft [15]. A Gradient Boosting Decision Tree is an ensemble model that builds trees sequentially. For LGBM, the trees are grown leaf-wise, as shown in Figure 2.4. Each new tree attempts to predict and correct the errors (residuals) of its predecessor. The tree is then added to the ensemble with the previous trees to improve overall performance. This process is repeated iteratively until the model reaches a predefined number of trees.

When the final prediction is made, each tree contributes a fixed score based on which leaf node a sample falls into. These scores are scaled by the learning rate and summed together to produce the final score. For binary classification, the final score is passed through the sigmoid function to generate a probability, which then determines the predicted class. Each leaf score represents the optimal constant value that minimizes the loss function for the data samples assigned to that leaf. In other words, if a tree were used on its own the leaf score would be the best possible prediction for all samples that reach that leaf. This is based on the residual errors at that point in training. The boosting process then adds up these contributions from many such trees to build a strong overall model.

LGBM introduces two key techniques, Gradient-Based One Side Sampling, and Exclusive Feature Bundling, to achieve faster training and better memory efficiency compared to traditional gradient boosting methods. Gradient-based One-Side Sampling speeds up training by selecting a subset of the data based on gradient values. It keeps all data instances with large gradients which are those the model is currently performing poorly on and randomly samples from instances with small gradients.

This lets the model focus on the most informative data points while reducing computational need.

Exclusive Feature Bundling reduces the number of features by combining mutually exclusive features that never take non-zero values at the same time into a single feature. This is possible since these features do not overlap in terms of their non-zero values so only information about which original feature corresponds to a zero value is lost. This reduces the feature dimensionality without significant information loss which leads to faster training and reduced memory usage.

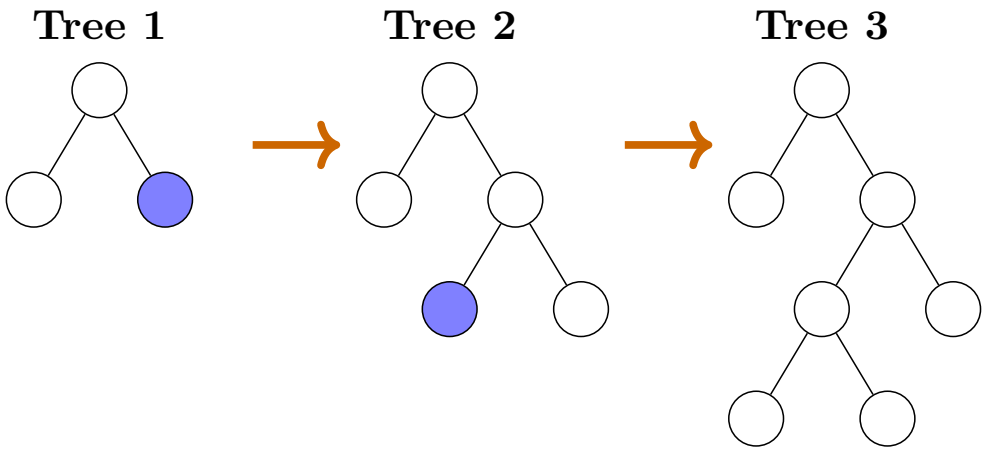

Figure 2.4: Illustration of LGBM training process for 3 iterations. Each tree is trained sequentially on residuals from the previous tree, and they grow leaf-wise as indicated by the blue leaves.

2.2 Transformer Model

The Transformer was originally introduced by Vaswani et al. [10]. It is an encoder-decoder architecture based on self-attention that takes a sequence as input and outputs a sequence. The encoder turns the input sequence into a high-dimensional vector that the decoder uses to produce an output sequence. This transformer architecture produced state-of-the-art results in natural language processing tasks such as machine translation and general language understanding. Unlike recurrent neural networks and convolutional neural networks, transformers do not rely on recurrence or convolution to capture sequential or spatial dependencies in data. Instead, transformers use self-attention to capture relationships between feature vectors at each timestep in the sequence. This means that each feature vector attends to all other feature vectors in the sequence simultaneously. As a result, transformers handle long-range dependencies more effectively, allowing for parallelization that speeds up training.

To perform classification and produce a single prediction rather than a sequence, only the encoder of the Transformer is used, similar to the approach of Zerveas et al. [11]. This encoder-based setup consists of three main components: input em-

beddings combined with positional embeddings, the encoder layers, and the output layer, see Figure 2.5.

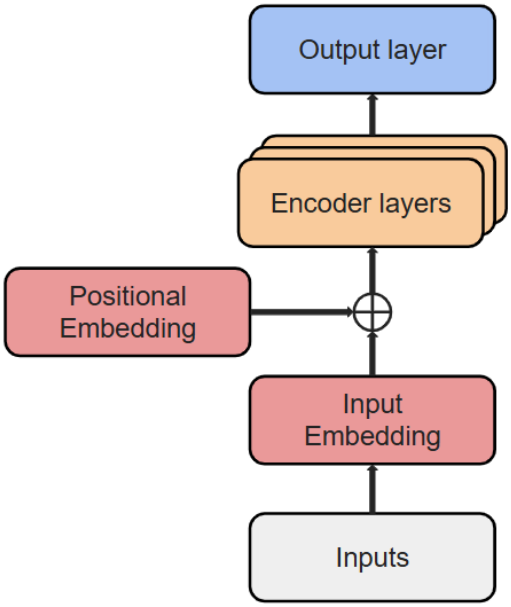

Figure 2.5: Structure of a general transformer encoder architecture.

2.2.1 Input Embedding and Positional Encoding

The input to the encoder is first passed through an embedding layer which projects the input to a higher-dimensional space. This gives the model more room to express relationships between features. Since the Transformer does not process the inputs sequentially, positional embeddings are added to the input embeddings to preserve the order of elements in the input sequence, see Figure 2.5. One approach to selecting positional embeddings is to use sinusoidal positional encodings [10], while another is to use fully learnable positional encodings [11].

2.2.2 Encoder Architecture

The encoder consists of N stacked, identical encoder layers. Each layer (see Figure 2.6) contains two main components: a multi-head self-attention mechanism (explained in Section 2.2.3) and a position-wise feed-forward network described in Section 2.2.4. Each of these components is surrounded by a residual connection, followed by layer normalization as represented in the formula:

$$\text{LayerNorm}(x + \text{Sublayer}(x))$$

Here, x is the input to the sublayer (either the attention mechanism or the feed-forward network), and $\text{Sublayer}(x)$ is the output produced by applying that sublayer to the input. Residual connections help gradient flow through deep networks [16], and layer normalization improves convergence. To speed up the residual connections, all layers in the model and the embeddings have the same dimension [10].

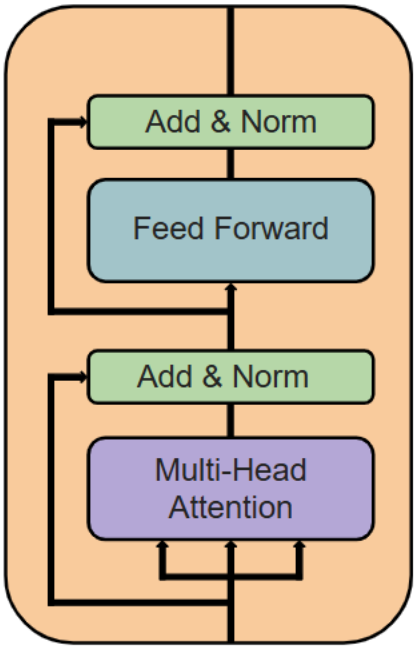

Figure 2.6: Structure of a general encoder layer.

2.2.3 Self-Attention and Multi-Head Attention

The self-attention mechanism is a fundamental building block of the Transformer architecture. It enables the model to compute relationships and similarities between embedding vectors in the input sequence. This is accomplished by transforming the input sequence into three matrices: queries (Q), keys (K), and values (V).

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

The input sequence $X \in \mathbb{R}^{n \times d_{\text{model}}}$ consists of n embedding vectors with an embedding size d_{model}. The weight matrices $W^Q, W^K, W^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$ are learnable parameters that are initialized with the Xavier Glorot Initialization [17] and used to project the input into query, key, and value spaces. For a single head attention each of the matrices Q, K, and V has dimensions $\mathbb{R}^{n \times d_k}$, where $d_k = \frac{d_{\text{model}}}{h} = \frac{d_{\text{model}}}{1} = d_{\text{model}}$. The attention output is then computed using the scaled dot-product attention mechanism introduced by Vaswani et al. [10], as shown in Equation 2.5. This operation produces a new representation of each embedding vector as a weighted sum of all embedding vectors in the sequence with their learned weights.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{2.5}$$

In multi-head attention, the attention function is applied independently and in parallel across h separate heads, each creating and using its own linear projections of the queries, keys, and values. Each attention head (see Equation 2.6) focuses on different aspects of the input, allowing the model to capture a wider variety of dependencies

across the sequence. These independent attention outputs are then concatenated and projected back to the original dimensions of $\mathbb{R}^{n \times d_{model}}$, see Equation 2.7.

$$\text{head}_i = \text{Attention}(Q_i, K_i, V_i) \quad \text{for } i = 1, \ldots, h \tag{2.6}$$

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{2.7}$$

Where $Q_i, K_i, V_i \in \mathbb{R}^{n \times d_k}$, where n is the sequence length, and $d_k = \frac{d_{\text{model}}}{h}$ is the dimensionality of each attention head. The learnable weight matrix $W^O \in \mathbb{R}^{hd_k \times d_{\text{model}}}$ projects the concatenated result back to the original embedding size of d_{model}. The concatenated output of all attention heads in Equation 2.7 has dimensions $\mathbb{R}^{n \times hd_k}$.

2.2.4 Feed-Forward Network

After the multi-head attention layer, each encoder layer also contains a Feed-Forward Network (FFN). The FFN is applied independently to each attention embedding in the input sequence. It consists of two fully connected linear layers with an activation function (σ) in between, as shown in Equation 2.8. The first layer projects the input from d_{model} to a higher-dimensional space d_{ff}, and the second projects it back to d_{model}. Specifically, W_1 and b_1 are the weights and bias for the first linear layer, while W_2 and b_2 are the weights and bias for the second. The activation function σ introduces non-linearity between the two layers.

$$\text{FFN}(x) = \sigma(xW_1 + b_1)W_2 + b_2 \tag{2.8}$$

2.2.5 Final Layer for Classification

The final step in the transformer encoder model is to make a prediction based on the output from the final encoder layer. The encoder layer output, which is a vector representing the final processed representation of the input sequence, is then passed through a final layer with dimensions $d_{\text{model}} \times 1$, where d_{model} is the embedding size. This final layer produces a scalar value, which represents the model's raw prediction. For binary classification, this raw prediction is then passed through a sigmoid function to convert it into a probability between 0 and 1. A threshold of 0.5 is used to decide the class, with probabilities above and equal to 0.5 classified as class 1 and probabilities below 0.5 classified as class 0.

2.3 Regularization Techniques in Neural Networks

Overfitting is a common issue in neural networks, where a model performs well on the training data but poorly on unseen data. This happens when the network becomes too specialized to the training set, learning noise and patterns that do not generalize. Overfitting often occurs when the model is too complex or trained for too many epochs. To reduce the risk of overfitting and improve generalization, techniques like dropout, weight decay, early stopping, and ensemble learning are often used.

2.3.1 Dropout

Dropout is a regularization technique used to reduce overfitting in neural networks. It works by randomly removing a subset of neurons during each forward pass in training. This means that selected neurons and their incoming and outgoing connections are temporarily removed from the network during training. The idea is that this forces the network to learn general features from the data since it cannot only rely on the presence of specific neurons. An example of dropout applied to a neural network with two hidden layers is shown in Figure 2.7.

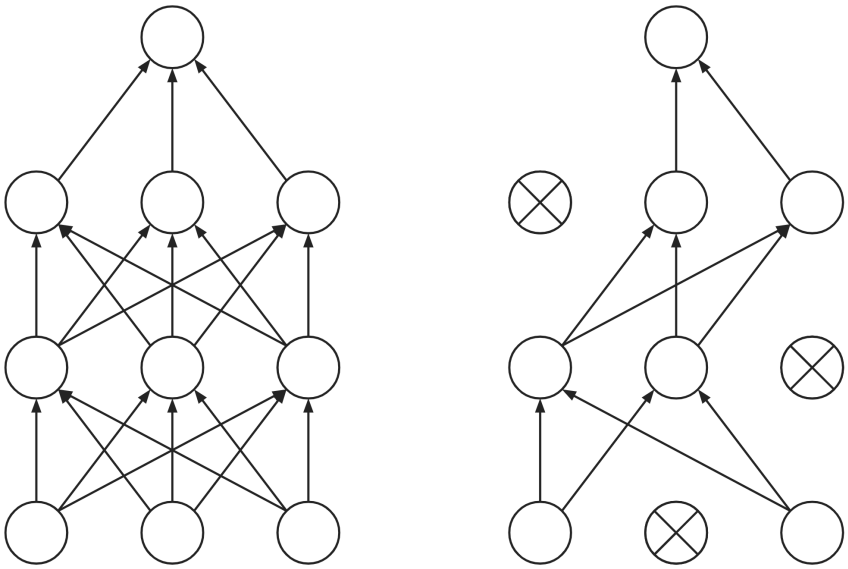

Figure 2.7: Dropout example. **Left:** A neural network with two hidden layers. **Right:** The same neural network with dropout applied, where the neurons marked with crosses have been dropped.

The dropout process is stochastic, meaning that each neuron is set to zero with an independent probability p. For example, a dropout rate of 0.4 means that each neuron has a 40% chance of being dropped in a given training step. This randomness effectively results in the training of multiple smaller subnetworks, each of which sees a slightly different view of the data.

At test time, all neurons are active and dropout is not used, but the outputs of neurons are scaled down by the same probability p as used during training. This compensates for the increased number of active units during inference and ensures that the expected output remains consistent between training and testing.

Dropout has been shown to improve performance across a wide range of tasks, including image classification, speech recognition, and translation, demonstrating that it is a general-purpose technique effective in many domains [10] [18] [19].

2.3.2 Weight Decay

The L^2 norm penalty commonly known as weight decay, is one of the simplest and most widely used regularization techniques in neural networks [20]. It works by

penalizing large weights, encouraging the model to keep them as small as possible without forcing them to zero. This is done by adding the penalty term in Equation 2.9 to the objective function, where $\mathbf{w}$ represents the model weights and λ is the regularization strength.

$$R(\mathbf{w}) = \frac{\lambda}{2}\|\mathbf{w}\|_2^2 = \frac{\lambda}{2}\sum_{i=1}^{n} w_i^2 \tag{2.9}$$

Weight decay is a hyperparameter that has two main benefits. Firstly, it helps reduce overfitting by encouraging smaller weight vectors, which often leads to simpler and more robust models. Secondly, when properly tuned, it can reduce noise in the training targets and improve the model's ability to generalize to unseen data [21].

2.3.3 Early stopping

Early stopping is another regularization technique. It aims to train the model long enough to reach good generalization but stop before overfitting begins [22]. Early stopping is also useful for saving computational resources, especially during cross-validation, where the model is trained multiple times with different hyperparameter configurations. With early stopping suboptimal configurations are terminated quickly, reducing unnecessary computation and improving overall efficiency.

Stopping at a first indication of overfitting usually leads to missing out on later optima since metrics such as validation performance can vary due to randomness in optimization, batch sampling, or noise in the data. Therefore, various stopping criteria exist that aim to find a good trade-off between generalization and computational cost.

Patience-based early stopping is a simple stopping criterion based on the idea that if the validation error does not improve for a set number of epochs (called patience), the model is considered to have started overfitting. In this case, training is stopped, and the model weights corresponding to the lowest validation error are returned. This decision is made even if the increase in validation error is small. The steps are shown in Algorithm 1.

Algorithm 1 Early Stopping with Patience

Input: P (patience), E (number of training epochs)
Output: θ (best model weights)
best_error $\leftarrow \infty$
epochs_no_improve $\leftarrow 0$
$\theta \leftarrow$ None
for $e = 1$ to E **do**
    if validation_error(e) $<$ best_error **then**
        best_error $\leftarrow$ validation_error(e)
        epochs_no_improve $\leftarrow 0$
        $\theta \leftarrow$ current model weights
    else
        epochs_no_improve $\leftarrow$ epochs_no_improve $+ 1$
    end if
    if epochs_no_improve $\geq P$ **then**
        return θ
    end if
end for
return θ

2.3.4 Ensemble Learning

Ensemble learning is a technique in machine learning where multiple models called inducers or base learners are combined to improve prediction accuracy. Ensemble methods are considered the state-of-the-art solution for many machine learning tasks and can be used with different types of models such as decision trees, neural networks, or linear regression models [23]. By combining the outputs from several models, the chance of errors from any one model is reduced. There are various ways to combine the outputs of the models, but a simple approach for classification is majority voting. In majority voting each inducer has equal impact and the final prediction is the class that receives the most votes from the individual models.

Ensemble methods are widely used because they often perform better than individual models, especially when the models are accurate and make different types of mistakes. This diversity is important because similar models that make the same predictions in an ensemble will not improve results much. One way to create diversity is by splitting the training data into different parts and training each model on a separate subset. This idea can be combined with cross-validation, which already splits the dataset in this way. Instead of just using the folds for finding hyperparameters, the model from each fold can be saved and later used as part of an ensemble.

Despite the higher performance of ensemble models, there are a few downsides to consider. First, they can lead to longer prediction times, especially when the ensemble includes a large number of complex models. Second, ensemble models can reduce interpretability, since the final output is based on the combined decisions of several inducers.

2.4 Metrics for Model Evaluation

When evaluating classification models, several metrics are used to assess performance, all of which rely on four key values: True Positives, False Positives, True Negatives, and False Negatives, defined in Table 2.1. Two other important metrics are precision and recall. Precision measures how accurate the model is in its positive predictions, see Equation 2.10. Recall indicates the proportion of actual positive instances correctly identified by the model, see Equation 2.11.

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{2.10}$$

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{2.11}$$

	Predicted Class 1	Predicted Class 0
Actual Class 1	True Positives	False Negatives
Actual Class 0	False Positives	True Negatives

Table 2.1: Definition of True Positives, False Positives, True Negatives, and False Negatives.

2.4.1 Accuracy and Balanced Accuracy

Balanced accuracy is a metric used to evaluate the performance of classification models, and can be particularly useful in cases where class distributions are imbalanced. Unlike standard accuracy (see Equation 2.12), balanced accuracy ensures that each class contributes equally to the overall performance measurement. It is calculated as the average of the recall values for each of the N classes, see Equation 2.13. This provides a better assessment of model performance across imbalanced datasets.

$$\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{True Positives} + \text{False Positives} + \text{True Negatives} + \text{False Negatives}} \tag{2.12}$$

$$\text{Balanced Accuracy} = \frac{1}{N} \sum_{i=1}^{N} \frac{\text{True Positives}_i}{\text{True Positives}_i + \text{False Negatives}_i} \tag{2.13}$$

2.4.2 Weighted F1 Score

The weighted F1 score is another metric that accounts for class imbalance. It uses the harmonic mean of precision and recall for each class just like the normal F1 score, but then it also weights each class's score by w_i. Where w_i is the number of samples in class i divided by the total number of samples in the dataset, see Equation 2.14.

This metric is useful when dealing with imbalanced datasets, as it reflects overall performance while considering the distribution of classes.

$$\text{Weighted F1 Score} = \sum_{i=1}^{N} w_i \cdot \frac{2 \cdot \text{Precision}_i \cdot \text{Recall}_i}{\text{Precision}_i + \text{Recall}_i} \tag{2.14}$$

2.5 Multiple Imputation by Chained Equations

Multiple Imputation by Chained Equations (MICE) is a method for handling missing data in multivariate datasets [24]. Unlike simpler imputation methods, such as mean or median imputation, MICE accounts for the relationships between variables improving the quality of the imputations.

The MICE process begins by filling missing values with placeholder values, for example, using mean values. Next for each variable with missing values, the algorithm resets it to missing and uses a model to predict the missing values based on the other variables. This step is repeated for every variable with missing values. This is considered one cycle. This process is described in a step by step explanation in Algorithm 2.

Algorithm 2 Iterative Imputation Algorithm

Input: $\mathcal{D}$ (dataset with missing values), $\mathcal{K}$ (number of cycles)
Output: $\hat{\mathcal{D}}$ (imputed dataset)
Let $\hat{\mathcal{D}}$ be a copy of $\mathcal{D}$ with missing values replaced by temporary placeholders (e.g., mean/median)
for $k = 1$ to K cycles **do**
    for each variable v_i with missing values **do**
        Set values in v_i in that were placeholders back to missing
        Fit a model $\mathcal{M}_i$ on $\hat{\mathcal{D}}$ with v_i as the target and the rest as predictors
        Use $\mathcal{M}_i$ to predict and impute missing values in v_i
    end for
end for
return $\hat{\mathcal{D}}$

The algorithm operates iteratively and refines the imputation with each cycle. Usually between 5-50 cycles are run to ensure that the computation converges to stable and accurate values. The iterative process allows for the imputation to handle more complex dependencies between the variables.

3

Methods

This chapter outlines the tools and methodology used in this thesis. The approach includes data compilation, preprocessing, baseline model selection, hyperparameter search for the transformer encoder model, and model evaluation. In this chapter and the rest of the thesis, vehicles that have experienced an ECU failure are referred to as failed vehicles, while those that have not are referred to as healthy vehicles.

3.1 Tools

To view, find, and analyze the different features of the dataset, Azure Data Studio was utilized. The software built in this thesis was developed using Python version 3.13.1. Python is widely used for data science because of its many useful libraries. The libraries used and their use cases are listed in Table 3.1.

Library	**Version**	**Use Case**
Pyodbc [25]	4.0.30	Connect to the database and send SQL queries.
SciPy [26]	1.7.3	Data exploration.
NumPy [27]	1.21.0	Compiling, cleaning, and manipulating data.
Pandas [28]	1.3.0	Handling structured data.
Matplotlib [29]	3.4.2	Visualizing and exploring the data using plots.
Seaborn [30]	0.11.1	Visualizing and exploring the data using plots.
Scikit-learn [31]	0.24.2	Machine learning models and data preprocessing.
LightGBM [15]	4.6.2	Machine learning model.
PyTorch [32]	1.9.0	Artificial Neural Networks and Transformer model.
Lazy Predict [33]	0.2.1	Quickly testing multiple machine learning models.

Table 3.1: Python libraries used in the thesis with their version and use case.

3.2 Data Exploration and Compilation

Volvo provided a database containing data from all their trucks, which we explored using Azure Data Studio. We focused on two main types of data. The first type, inspired by the SCANIA Component X Dataset [1], was operational data collected from truck sensors. This data measures the vehicle's performance, for example, total engine running time or total distance driven with different loads. The features

in this dataset were either scalar or in histogram format, meaning some features had multiple columns representing different intervals. As an example, the feature distance driven with different load levels have intervals such as 0-2 tons and 2-4 tons. The original operational features are listed in Table 3.2.

Row	Operational feature	#Columns
1	Total distance driven	1
2	Total engine running time	1
3	Total time in power take-off mode	1
4	Total time running in top gear	1
5	Total engine fuel consumption	3
6	Number of gear shifts	6
7	Total time spent with different engine temperature	8
8	Brake and retarder data	9
9	Total time spent in different ambient pressure	12
10	Total distance driven with different loads	32
11	Total distance driven in incline/decline over time	32

Table 3.2: The eleven original operational features and the number of columns per feature.

To ensure that data was reported at least monthly, each operational feature was queried across several vehicles. One insight was the high number of zero values in features represented as histograms with multiple intervals, particularly in the outer intervals. To address this, intervals were merged (see Table 3.3) to reduce the number of zero values and improve computational efficiency. Additionally, three operational features listed in rows 6, 7, and 9 in Table 3.2 were removed. This was due to a high proportion of missing values, which is explained in Section 3.3.1.

Operational feature	#Columns
Total distance driven	1
Total engine running time	1
Total time in power take-off mode	1
Total time running in top gear	1
Total engine fuel consumption	1
Brake and retarder data	3
Total distance driven with different loads	7
Total distance driven in incline/decline over time	8

Table 3.3: The eight final operational features and the number of columns per feature after merging intervals.

The second source of data consists of fault codes, which are logged when errors occur in engine components. We selected fault codes that were related to the ECU and had occurred in vehicles with ECU failures. This resulted in 59 features and

59 columns. By combining the operational data and the fault code data, the final dataset contained 67 features across a total of 82 columns.

3.3 Data Preprocessing Steps

Data preprocessing is essential for turning raw data into high-quality input, meaning data that is clean, consistent, and relevant for machine learning models [34]. This follows the well-known concept in computer science: Garbage In, Garbage Out. In other words, if the input data is poor the model's results will also be poor, regardless of how advanced the models are. The following is the data preprocessing pipeline used for preparing the datasets.

3.3.1 Data Cleaning and Aggregation

The first step in preprocessing involved identifying and removing duplicate records to avoid data redundancy, which could skew model training. Additionally, entries unrelated to the task were removed to ensure that only relevant data remained.

Next, to minimize the number of missing values in the operational data, a trial-and-error aggregation approach was used to determine the optimal granularity of the time dimension. The goal was to reduce the percentage of missing values while still keeping enough points in the time dimension. Weekly, biweekly, and monthly intervals were explored. Monthly aggregation intervals gave the best results and were used. If there existed multiple readings within an interval, the mean value was used, see Figure 3.1. We considered the mean to be a reasonable choice, given that the data for each feature approximately increased linearly with time. After aggregation, most features had 1-13% missing values, while some had around 50% or more. MICE has been shown to perform optimally when the proportion of missing values is at or below approximately 50% [35]. Since one feature had 53% missing values, we decided to make this the threshold and features with more than 53% missing values were removed.

Date	Vehicle ID	Feature 1	Feature 2
2021-02-15	1	20	Nan
2021-02-17	1	40	10
2021-03-22	1	80	40
2021-03-25	1	120	60

Date	Vehicle ID	Feature 1	Feature 2
2021-02	1	30	10
2021-03	1	100	50

Figure 3.1: An example of the monthly aggregation method with mean replacement.

3.3.2 Overview of Data Classes

A total of 147 vehicles experienced an ECU failure within three years of the truck's assembly date, compared to hundreds of thousands of vehicles that did not experience an ECU failure. All trucks in the dataset have been in operation for at most five years. Based on this, the first approach was to divide the data into three classes of similar time frames. Since 99% of the failed vehicles had failed within three years this was decided as our upper class cutoff.

- **Class 0**: 3+ years until failure
- **Class 1**: 1-3 years until failure
- **Class 2**: 0-1 years until failure

Two types of datasets were created for the models: one tabular dataset for classical machine learning models and one time-sequential dataset for the time-sequential models. The time-sequential dataset consisted of one sequence of data for each vehicle and the label for each sequence was determined by the class of the last data point in the sequence. This is explained in Table 3.4, which presents a sequence for a vehicle. The sequence is labeled as 2 because the class for the final reading is 2. As a result of this class assignment method, all healthy vehicles were assigned the label 0 and all failed vehicles received the label 2. This created an imbalance in the dataset, as there were no instances of label 1, which was a significant issue.

Timesteps until failure	40	30	20	10	0
Class	0	1	1	2	2

Table 3.4: An example sequence for one vehicle which gets label 2 since the final reading has class 2.

To solve this issue, we tried to create new sequences with label 1 by cropping the sequences for the failed vehicles. However, this resulted in very short sequences which resulted in the models performing poorly. Instead, we decided to have two classes instead of three.

- **Class 0**: 3+ years until failure
- **Class 1**: 0-3 years until failure

When assigning healthy vehicles to Class 0 it was decided to exclude data points from the last three years. This was to ensure that the vehicle operates at least another three years after the last considered data point so it can safely be assigned Class 0. It was also required for the vehicles to have been in operation for at least five years to provide sufficient data.

3.3.3 Dataset overview

Two datasets were created from the same data. The first one was a tabular dataset consisting of 9683 rows used for the baseline models. The class distribution of the

tabular dataset between healthy vehicles and failed vehicles is shown in Figure 3.2. As seen in the figure, there is a four-to-one class imbalance.

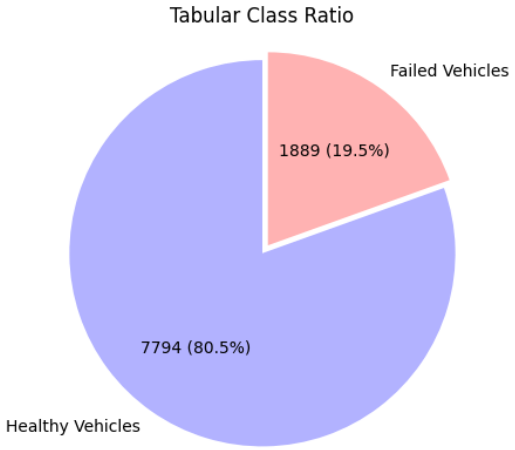

Figure 3.2: Class distribution of the tabular dataset between the classes: healthy and failed vehicles.

For the transformer encoder models a sequential dataset consisting of 432 sequences was used. The class distribution of the sequential dataset is shown in Figure 3.3. As seen in the figure, there is a two-to-one class imbalance. A key consideration with time series data is that the length of sequences can vary a lot, as shown in Figure 3.4. This variation exists not only between the two classes but also within each class. Failed vehicles have shorter sequences because they fail earlier, while the variation in healthy vehicles is due to differences in their production dates.

To make the sequences compatible with the transformer encoder model, each one was padded at the beginning with the value -1e6 to reach a uniform length of 36. This value was chosen instead of the more common zero because zeroes can appear in the rescaled data. Since the data is centered around zero, -1e6 provides a safe padding value that is clearly outside the normal data range. These padded values are used to match the required input shape. To ensure they do not affect the model, a padding mask is applied. This mask assigns large negative scores to the padded positions before applying the softmax function in Equation 2.5.

This padding length is slightly above the maximum sequence length of 35 months since it creates a three-year window. If the model is used on sequences longer than 36 months, the most recent 36 timesteps are kept. The dataset was split into two parts: 80% for training and 20% for testing while keeping the original class ratio in both sets.

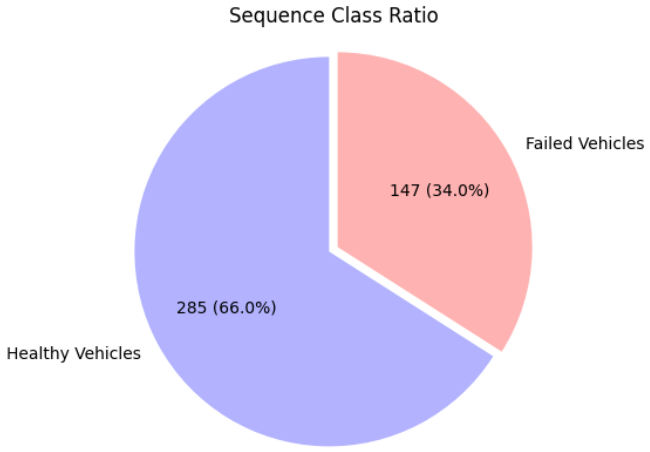

Figure 3.3: Sequence distribution between the two classes: healthy and failed vehicles.

3.3.4 Handling Missing Values with MICE

To fill in missing values MICE was used because it can handle complex dependencies between variables and it performs well even when there exist a high amount of missing values [24]. Being able to handle a higher amount of data was important since much of the available data contained a fair amount of missing values.

During cross-validation, an imputer was created for each fold by fitting it on the training data, and then used to impute missing values in both the training and validation sets. After cross-validation, a new imputer was fitted on the entire training set and used to impute missing values in both the full training set and the test set.

An important aspect of using a MICE imputer is selecting an appropriate model strategy for filling in missing values. Since the dataset consisted of continuous variables, a linear regression model was chosen, as it is the standard and often the most effective option for this type of data [35].

3.3.5 Rescaling Data with Robust Scaler

In the paper by Amorim et al. [36], multiple scalers were compared across several classification tasks. They found that the Quantile Transformer was best, but it could distort the linear correlations between variables, which would negatively affect the interpretability of the relationships between features. Furthermore, the inverse transformation of the Quantile Transformer does not guarantee recovery of the original values. Instead, we chose to rescale the data with the Robust Scaler, since it was the best performing scaler that kept the linear relations between features and provided a reliable inverse transformation. This property is valuable for interpreting the model's behavior in relation to the original features.

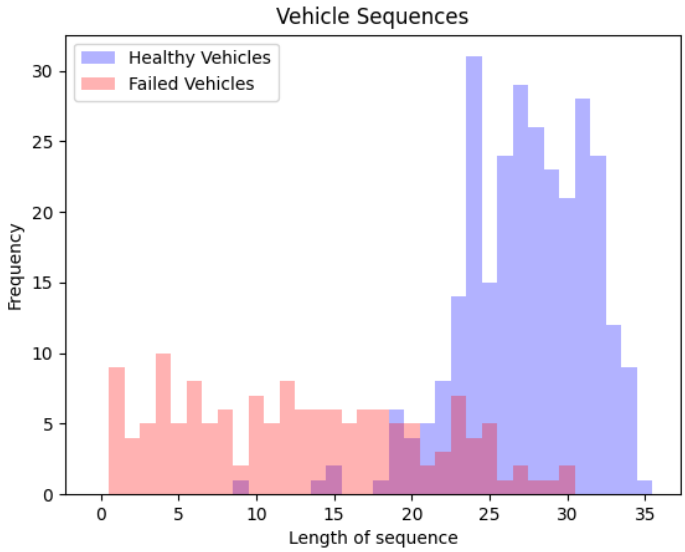

Figure 3.4: The length of the sequences and their frequency for the two classes: healthy and failed vehicles.

The Robust Scaler (see Equation 3.1) scales each feature instance x_i by first subtracting the median of the feature $\bar{x}$, and then dividing by the interquartile range (IQR). The IQR is the difference between the 75th percentile (x_{75}) and the 25th percentile (x_{25}) of the feature. Scaling the features with the IQR makes this method less sensitive to outliers compared to other methods, such as min-max normalization.

$$x_{scaled} = \frac{x_i - \bar{x}}{x_{75} - x_{25}} \tag{3.1}$$

3.4 Selecting Baseline Models Using Lazy Predict

To provide a comparison for the transformer encoder model, four baseline models were selected. Three of these were identified using the Lazy Predict library [33], which evaluated 26 traditional classification models on the dataset. Since the dataset was already imputed and scaled the corresponding preprocessing steps were removed from the built-in Lazy Predict pipeline. The top three performing models in terms of accuracy were the Extra Trees Classifier, LGBM Classifier, and Random Forest Classifier, as shown in Table 3.5. A fourth baseline model, Logistic Regression was included because it is commonly used as a reference and it differs from the ensemble-based methods. All baseline models from the scikit-learn and LightGBM libraries were trained using the default parameters without any hyperparameter tuning.

Model	Accuracy	Balanced Accuracy	F1 Score	Time (s)
Extra Trees Classifier	0.829	0.576	0.786	0.632
LGBM Classifier	0.828	0.574	0.268	0.784
Random Forest Classifier	0.823	0.547	0.767	1.581
Ada Boost Classifier	0.821	0.529	0.754	1.090
XGB Classifier	0.820	0.571	0.780	0.202
Linear Discriminant Analysis	0.815	0.538	0.759	0.079
Calibrated Classifier CV	0.813	0.500	0.730	0.146
Dummy Classifier	0.813	0.500	0.730	0.017
Support Vector Classifier	0.813	0.500	0.730	1.113
Ridge Classifier	0.813	0.514	0.742	0.027

Table 3.5: Performance of the top 10 traditional classifiers based on accuracy using Lazy Predict.

3.5 Architecture and Hyperparameter Search for Transformer Model

We decided to use fully learnable positional encodings, as they outperformed sinusoidal encodings on various classification tasks [11]. To find a transformer encoder architecture that works well with the dataset we performed a two-stage hyperparameter search, starting with a broad random search, followed by a local exhaustive grid search. We used 5-fold cross-validation in both phases to reduce variance and assess generalization. Each fold was trained for up to 200 epochs using early stopping with patience of 10 epochs. This setup aimed to allow models to converge while keeping computational efficiency. The training was conducted using the Adam optimizer with weight decay and a fixed learning rate. We empirically tested batch sizes of 16, 32, 64, and 128. This test concluded that a batch size of 64 offered the best trade-off between training speed and model stability.

We began the hyperparameter search with a random search over a parameter space inspired by previous transformer-based regression and classification models [5] [10] [11]. The search spanned 6912 total combinations across model and training parameters, summarized in Table 3.6. For the feedforward network within each encoder layer, we followed the approach in [10] by expanding its dimensionality to four times the hidden dimension.

Parameter Name	Parameter Type	Search Space
Number of heads	Model	1, 2, 4, 8
Hidden dimensions	Model	128, 256, 512
Number of layers	Model	1, 2, 3, 4, 5, 6
Activation function	Model	GELU, ReLU
Dropout	Model	0.2, 0.3, 0.4, 0.5
Learning rate	Training	1e-4, 1e-5, 1e-6
Weight decay	Training	0, 1e-4, 1e-5, 1e-6

Table 3.6: Hyperparameter search space for random search with a total of 6912 combinations.

After 1000 random samples, the best configuration found is shown in Table 3.7. This configuration served as the starting point for the local exhaustive grid search.

Parameter Name	Value
Number of heads	1
Hidden dimensions	128
Number of layers	5
Activation function	GELU
Dropout	0.4
Learning rate	1e-4
Weight decay	1e-6

Table 3.7: Best random search hyperparameter configuration after 1000 iterations.

To further improve the results we defined a local search space centered around the best random configuration. It was decided to keep the dropout fixed at 0.4 to reduce the number of combinations in the exhaustive search. The local exhaustive grid search covered 243 combinations shown in Table 3.8.

Parameter Name	Parameter Type	Search Space
Number of heads	Model	1, 2, 4
Hidden dimensions	Model	64, 128, 256
Number of layers	Model	4, 5, 6
Activation function	Model	GELU, ReLU
Dropout	Model	0.4
Learning rate	Training	3e-4, 1e-4, 3e-5
Weight decay	Training	1e-5, 1e-6, 1e-7

Table 3.8: Hyperparameter search space for local exhaustive search with a total of 243 combinations.

The best configuration from the local grid search is listed in Table 3.9. Compared to the random search configuration, it favors a slightly larger hidden dimension and replaces the GELU activation function with ReLU.

Parameter Name	Value
Number of heads	1
Hidden dimensions	256
Number of layers	4
Activation function	ReLU
Dropout	0.4
Learning rate	3e-4
Weight decay	1e-7

Table 3.9: Best local exhaustive search hyperparameter configuration.

These optimized hyperparameters were used to construct the final transformer encoder architecture visualized in Figure 3.5. The model uses a single attention head and four encoder layers, each with a hidden dimensionality of 256. This configuration leads to a model with approximately 2.4 million trainable parameters.

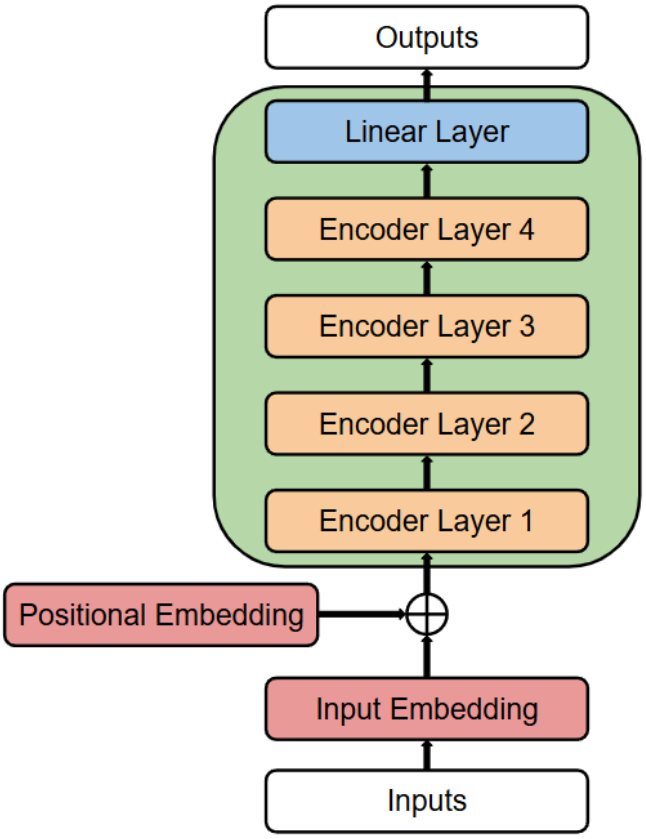

Figure 3.5: Final transformer encoder architecture based on parameters from the exhaustive search.

3.6 Model Evaluation

All models, including the transformer encoder and the selected baseline classifiers, are evaluated using accuracy, balanced accuracy, weighted F1 score, and a custom cost metric, see Equation 3.2. In addition, two naive classifiers that always predict the same class are included as benchmarks. The non-ensemble transformer encoder models were each trained and evaluated five times independently, and their metrics were averaged to obtain more stable performance estimates.

$$\text{Total Cost} = \frac{1}{n}(1 \times \text{FP} + 10 \times \text{FN}) \tag{3.2}$$

The custom cost function was developed, inspired by the approach in [1] to account for the different costs of incorrect predictions. In their case, there were five classes, with misclassification costs ranging from 7 to 500, depending on the severity of the error. In the context of the binary classification task addressed in this thesis, the model may either over-predict or under-predict the RUL of the ECU. An over-prediction which corresponds to a False Negative (FN) as shown in Table 3.10 occurs when the model estimates that the ECU will last longer than its actual lifespan. This can result in unexpected component failure that can lead to vehicle breakdowns, towing requirements, delivery delays, and possible spoilage of transported goods. An under-prediction is equivalent to a False Positive (FP) which occurs when the model predicts that the ECU will fail before its true lifespan. This may lead to unnecessary replacement of ECUs, resulting in extra costs and resource waste. Although over-predictions are more costly, defining the exact cost ratio between the two types of errors is difficult. A rough estimation used in this work, assumes that over-predictions are around ten times more costly than under-predictions, see Table 3.11.

	Predicted Class 1	Predicted Class 0
Actual Class 1	TP	**FN**
Actual Class 0	**FP**	TN

Table 3.10: Outline of confusion matrix for binary classification.

	Predicted Class 1	Predicted Class 0
Actual Class 1	0	**10**
Actual Class 0	**1**	0

Table 3.11: Estimated cost ratio between FP and FN for the RUL of ECUs. The cost for TP and TN are zero.

Based on the cost estimation for FP and FN provided in Table 3.11, a custom cost function is presented in Equation 3.2. To ensure a fair comparison, the cost function was normalized by dividing by the number of predictions, n. This normalization was necessary because the transformer model makes predictions based on sequential data,

while the baseline models make predictions on individual data points within each sequence.

4

Results

This section presents the results of the thesis, beginning with the performance of the baseline models, followed by the results of the transformer encoder models. The models are compared based on accuracy, balanced accuracy, weighted F1 score, and a custom cost metric. In addition, confusion matrices are included to provide more detailed insight into where the models make incorrect predictions. Finally, for the ensemble models, plots are presented to visualize the level of vote agreement among the models within each ensemble.

4.1 Results for Baseline Models

Table 4.1 summarizes the performance of the baseline models, including simple naive classifiers. The models were evaluated using accuracy, balanced accuracy, weighted F1 score, and a custom cost metric. Two naive classifiers, which always predict a single class, serve as reference points.

Model	**Accuracy**	**Balanced Accuracy**	**F1 Score**	**Custom Cost**
Logistic Regression	0.75	0.66	0.77	1.08
Random Forest	0.82	0.55	0.77	1.68
Extra Trees	0.83	0.57	0.78	1.58
LGBM	0.83	0.57	0.78	1.57
Naive positive	0.20	0.50	0.31	0.81
Naive negative	0.80	0.50	0	1.87

Table 4.1: Results of baseline models. This also includes naive classifiers which either predicts only healthy or only failed.

To further assess the classification performance of the baseline models, confusion matrices were computed for each. Tables 4.2 through 4.5 show the distribution of true and false predictions for each model, broken down by actual and predicted classes.

	Pred 1	**Pred 0**
Actual 1	179	176
Actual 0	292	1256

Table 4.2: Confusion matrix for Logistic Regression

	Pred 1	**Pred 0**
Actual 1	20	335
Actual 0	13	1535

Table 4.3: Confusion matrix for Model Random Forest

	Pred 1	**Pred 0**
Actual 1	56	299
Actual 0	25	1523

Table 4.4: Confusion matrix for Model Extra Trees

	Pred 1	**Pred 0**
Actual 1	60	295
Actual 0	33	1515

Table 4.5: Confusion matrix for Model LGBM

4.2 Results for Transformer Models

Table 4.6 shows the performance of the transformer encoder models, comparing different hyperparameter search methods and both ensemble and non-ensemble versions. The Parameter column indicates whether the model's parameters were selected using a random or exhaustive search. The same evaluation metrics as for the baseline models are used: accuracy, balanced accuracy, weighted F1 score, and a custom cost. Two naive classifiers applied to the time series data are included as baselines to highlight how simple strategies compare to the transformer models.

Model	**Ensemble**	**Parameter**	**Accuracy**	**Balanced Accuracy**	**F1 Score**	**Custom Cost**
Transformer Encoder	No	Random search	0.80	0.78	**0.80**	1.05
Transformer Encoder	Yes	Random search	**0.85**	**0.84**	0.79	0.77
Transformer Encoder	No	Exhaustive search	0.80	0.79	**0.80**	0.97
Transformer Encoder	Yes	Exhaustive search	0.83	0.81	0.75	0.90
Naive Positive	-	-	0.34	0.50	0.18	**0.66**
Naive Negative	-	-	0.66	0.50	0.52	3.44

Table 4.6: Results of transformer encoder models and naive classifiers on the test set.

Similar to the baseline models, confusion matrices were also computed for the transformer models. Table 4.7 presents the average confusion matrix for the non-ensemble model using random search parameters, averaged over five runs. The confusion matrix for the best-performing ensemble model with random search parameters is shown in Table 4.9. Similarly, Table 4.8 displays the average confusion matrix for the non-ensemble model with exhaustive search parameters. The corresponding confusion matrix for the ensemble model with exhaustive search parameters is presented in Table 4.10.

	Pred 1	**Pred 0**
Actual 1	21.8	8.2
Actual 0	9.4	47.6

Table 4.7: Average confusion matrix over 5 runs for non-ensemble Transformer model with random search parameters.

	Pred 1	**Pred 0**
Actual 1	22.6	7.4
Actual 0	10	47

Table 4.8: Average confusion matrix over 5 runs for non-ensemble Transformer model with exhaustive search parameters.

	Pred 1	**Pred 0**
Actual 1	24	6
Actual 0	7	50

Table 4.9: Confusion matrix for ensemble Transformer model with random search parameters.

	Pred 1	**Pred 0**
Actual 1	23	7
Actual 0	8	49

Table 4.10: Confusion matrix for ensemble Transformer model with exhaustive search parameters.

To further investigate the behavior of the ensemble models, Figures 4.1 and 4.2 show the number of correct and incorrect predictions based on how many individual models agreed on the outcome. Figure 4.1 illustrates this for the ensemble model with random search parameters, while Figure 4.2 presents the same analysis for the exhaustive search variant. It can be observed in both cases that prediction accuracy increases as the number of agreeing models increases.

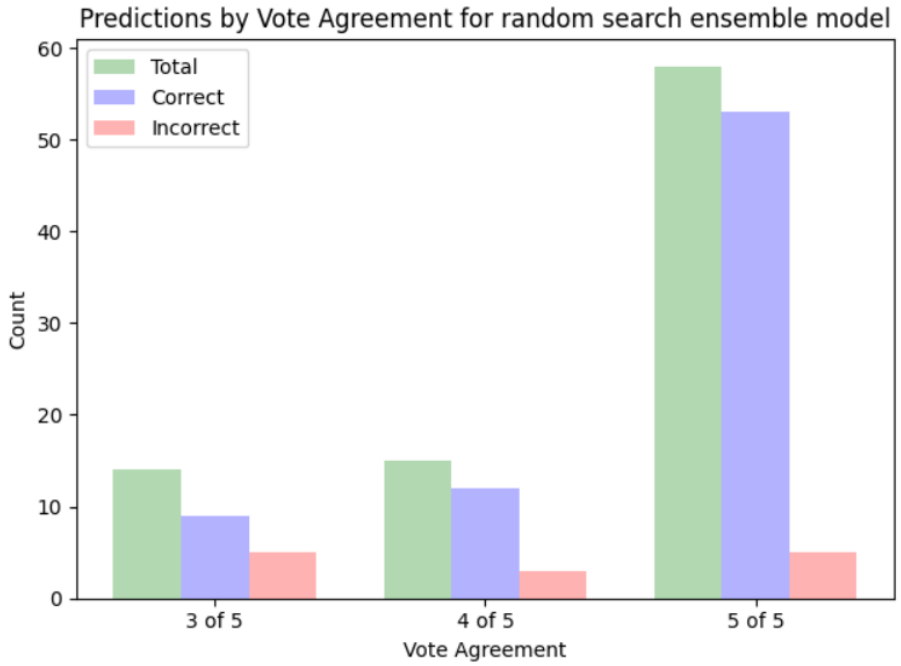

Figure 4.1: Correct and incorrect predictions made by the ensemble model from the random search, grouped by the number of agreeing models (three, four, or five).

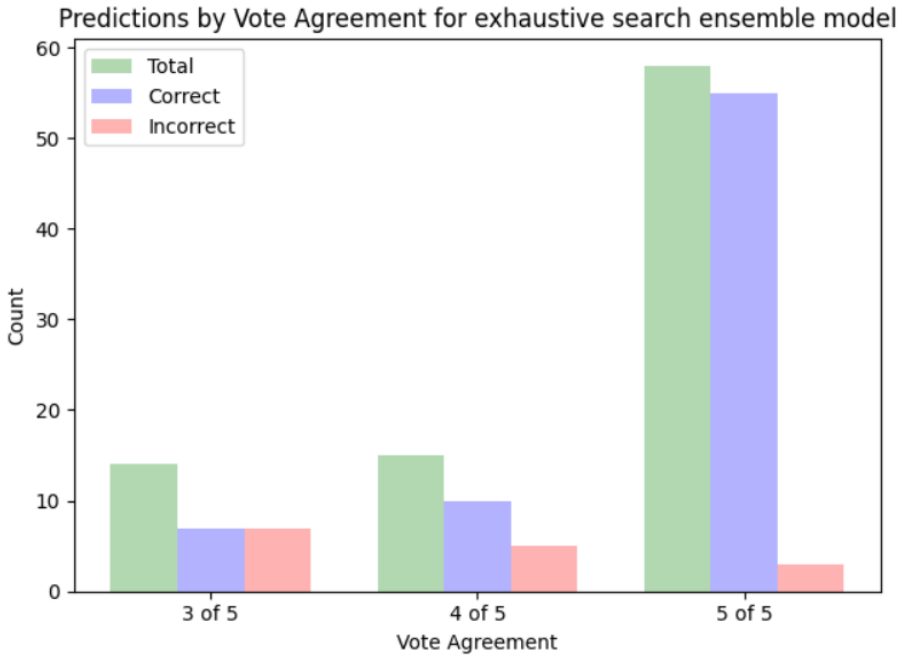

Figure 4.2: Correct and incorrect predictions made by the ensemble model from the exhaustive search, grouped by the number of agreeing models (three, four, or five).

5

Discussion

In this final chapter, we discuss the performance of the models based on the results presented in the previous chapter and address the three subgoals of the thesis. Finally, we present directions for future work and conclude the thesis.

5.1 Comparison of Model Performance

The first subgoal SG1 of the thesis was to evaluate and compare the transformer encoder model against traditional machine learning models. Before comparing the results, it is important to note that even though the data is the same for both the tabular (see Table 3.2) and sequential (See Table 3.3) datasets, the tabular dataset has a bigger class imbalance. This is because the sequences for the minority class are shorter, which makes the imbalance worse.

In Table 4.1, we observe that among the baseline models logistic regression had the lowest overall accuracy at 0.75, falling below the naive negative classifier. However, it achieved the highest balanced accuracy at 0.66 and only had 176 false negative predictions (see Table 4.2) which is much less than any other baseline model, leading to a custom cost of 1.08 which is the lowest among the baseline models.

In contrast, the Random Forest model achieved an accuracy of 0.82 which is higher than the logistic regression model. However, as seen in Table 4.3 it performed poorly on the positive cases leading to 335 false negatives compared to the 176 of the logistic regression. False negatives are the most costly incorrect prediction leading to the highest custom cost of 1.87 of any of the baseline models.

Extra Trees and LGBM produced very similar results across all metrics. Both achieved an accuracy of 0.83 which was the highest of the baseline models. They also had identical balanced accuracy of 0.57 and weighted F1 score of 0.78. The only difference was in the custom cost, where LGBM had a slightly lower value of 1.57 compared to Extra Trees' 1.58, due to a lower number of false negatives, as shown in Tables 4.4 and 4.5.

The transformer encoder model's results, shown in Table 4.6, were better than the baseline models across all standard metrics. However, on the custom cost metric the naive positive classifier obtained a cost lower than any of the transformer encoder

models. Based on the custom cost this would suggest that the best strategy for handling used ECUs is to always discard them instead of reinstalling them. For a model to be useful in the real world, it needs to perform better than the naive classifiers on this metric, which none of our models achieve. One reason the models do not reach a lower cost is that they are trained to maximize accuracy rather than minimize the custom cost. An important point is that the cost ratio between false positives and false negatives is only a rough estimate, and more work is needed to make it more accurate.

To address subgoal SG2, we aimed to optimize model performance by tuning the hyperparameters of a transformer encoder model. We used both a random grid search and a local exhaustive grid search. The random search helped us explore a wide range of parameter combinations with lower computational costs and was expected to provide a solid starting point. The exhaustive search was then applied with the idea to locally optimize this result and hopefully improve performance further.

However, as shown in Table 4.6, there was no improvement in either accuracy or weighted F1 score for the single transformer model from the exhaustive search compared to the random search model. Only small improvements were seen in balanced accuracy and the custom cost when using the parameters from the exhaustive search for the single model. The confusion matrices in Tables 4.7 and 4.8 show that the models make similar predictions. However, the random search model produces more over-predictions and fewer under-predictions compared to the exhaustive search model. Since over predictions are ten times more costly than under predictions this is what leads to the small difference in custom cost.

When comparing the two ensemble models, the one from the random search actually performed better than the one from the exhaustive search. From the confusion matrices in Tables 4.9 and 4.10 it is clear that the random search model outperforms both in terms of under and over-predictions. This was unexpected since the exhaustive search model had performed better during cross-validation. We believe this may be due to the small size of the dataset, which might have caused the exhaustive search to overfit the training data. We tried to prevent this by using techniques like ensemble learning and early stopping, but overfitting may still have occurred.

The third subgoal SG3, was to explore whether ensemble methods can improve the performance of the transformer encoder model. The results in Table 4.6 show that ensemble models performed better than the single models for both the random and exhaustive parameter search solutions. By looking at the ensemble model voting results in Figures 4.1 and 4.2, we can see that all five models in the ensemble agree on the final prediction in most cases. A large portion of these unanimous predictions are correct, which suggests that the models have a shared understanding of the data.

The improved performance of the ensemble compared to a single model is due to instances where only three or four models agree on the prediction. In these cases, due to the way the data was split during cross-validation, different models can correctly

identify parts of the more difficult examples. This contributes to the overall better performance of the ensemble model compared to any single model. This is true for both the transformer encoder trained with random search and the one trained with exhaustive search parameters.

5.2 Future Work

Continuing to collect more data where ECUs have failed and increasing the size of the dataset will likely improve the performance of the models. A larger dataset can help the models generalize better and make more accurate predictions. It would also be interesting to further explore additional operational features or truck-related data that could provide the models with more information.

Another potential improvement is to experiment with sinusoidal positional encodings in the transformer encoder model. Although this thesis used learnable positional encodings, which were recommended for classification tasks based on the study by Zerveas et al. [11], the original transformer model by Vaswani et al. [10] used sinusoidal encodings. The performance of different positional encoding techniques may also depend on the specific dataset, making this an area for further research.

Including alternative optimization techniques, such as the AdamW optimizer or Stochastic Gradient Descent in the hyperparameter search could potentially improve the performance. Additionally, combining these optimizers with learning rate schedulers may lead to faster convergence and more stable training performance.

Beyond exploring other hyperparameters, further improvements can come from exploring advanced hyperparameter search methods. Techniques like Bayesian Optimization that guide the search using a probabilistic model or Genetic Algorithms that evolve parameter settings over time could lead to stronger models.

Another direction for future research is to apply other time sequential models, such as Long Short-Term Memory, Gated Recurrent Unit, and Recurrent Neural Networks to this dataset. This would help assess the performance of the transformer encoder used in this work.

5.3 Conclusion

In this thesis, several transformer encoder models were developed and trained to classify the RUL of ECUs. These models were evaluated and compared against four traditional machine learning models and two naive classifiers. The transformer models outperformed the traditional machine learning models in terms of accuracy, balanced accuracy, weighted F1 score, and the custom metric. Additionally, ensemble transformer models showed better performance than single transformer models. However, the transformer models performed worse than a naive classifier when evaluated using the custom cost metric. Further research on this dataset is therefore

needed before the models can be applied in real-world operations. This work serves as a starting point for improving decision-making in ECU refurbishment, especially when assessing their reuse in Volvo trucks.

Bibliography

[1] T. Lindgren, O. Steinert, O. Andersson Reyna, Z. Kharazian, and S. Magnússon, *SCANIA Component X Dataset: A Real-World Multivariate Time Series Dataset for Predictive Maintenance (Version 2)*, Dataset, 2024. DOI: https://doi.org/10.5878/jvb5-d390.

[2] G. Sateesh Babu, P. Zhao, and X.-L. Li, "Deep convolutional neural network based regression approach for estimation of remaining useful life," in *Database Systems for Advanced Applications*, S. B. Navathe, W. Wu, S. Shekhar, X. Du, X. S. Wang, and H. Xiong, Eds., Cham: Springer International Publishing, 2016, pp. 214–228, ISBN: 978-3-319-32025-0.

[3] X. Li, Q. Ding, and J.-Q. Sun, "Remaining useful life estimation in prognostics using deep convolution neural networks," *Reliability Engineering & System Safety*, vol. 172, pp. 1–11, 2018, ISSN: 0951-8320. DOI: https://doi.org/10.1016/j.ress.2017.11.021.

[4] D. Chen, W. Hong, and X. Zhou, "Transformer network for remaining useful life prediction of lithium-ion batteries," *IEEE Access*, vol. 10, pp. 19 621–19 628, 2022. DOI: https://doi.org/10.1109/ACCESS.2022.3151975.

[5] O. Ogunfowora and H. Najjaran, *A transformer-based framework for multivariate time series: A remaining useful life prediction use case*, 2023. DOI: https://doi.org/10.48550/arXiv.2308.09884.

[6] C. Ferreira and G. Gonçalves, "Remaining useful life prediction and challenges: A literature review on the use of machine learning methods," *Journal of Manufacturing Systems*, vol. 63, pp. 550–562, 2022, ISSN: 0278-6125. DOI: https://doi.org/10.1016/j.jmsy.2022.05.010.

[7] F. Ahmadzadeh and J. Lundberg, "Remaining useful life estimation: Review," *International Journal of System Assurance Engineering and Management*, vol. 5, no. 4, pp. 461–474, 2014, ISSN: 0976-4348. DOI: https://doi.org/10.1007/s13198-013-0195-0.

[8] D. K. Frederick, J. A. DeCastro, and J. S. Litt, "User's guide for the commercial modular aero-propulsion system simulation (C-MAPSS)," Tech. Rep., 2007.

[9] Y. Cheng, J. Wu, H. Zhu, S. W. Or, and X. Shao, "Remaining useful life prognosis based on ensemble long short-term memory neural network," *IEEE Transactions on Instrumentation and Measurement*, vol. 70, pp. 1–12, 2021. DOI: https://doi.org/10.1109/TIM.2020.3031113.

[10] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017. [Online]. Avail-

able: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[11] G. Zerveas, S. Jayaraman, D. Patel, A. Bhamidipaty, and C. Eickhoff, *A transformer-based framework for multivariate time series representation learning*, 2020. DOI: https://doi.org/10.48550/arXiv.2010.02803.

[12] L. Breiman, "Random forests," *Machine learning*, vol. 45, pp. 5–32, 2001.

[13] L. Breiman, J. Friedman, R. A. Olshen, and C. J. Stone, *Classification and regression trees*. Routledge, 2017.

[14] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine learning*, vol. 63, pp. 3–42, 2006.

[15] G. Ke, Q. Meng, T. Finley, *et al.*, "LightGBM: A Highly Efficient Gradient Boosting Decision Tree," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, *et al.*, Eds., vol. 30, Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf.

[16] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778. DOI: https://doi.org/10.1109/CVPR.2016.90.

[17] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, Y. W. Teh and M. Titterington, Eds., ser. Proceedings of Machine Learning Research, vol. 9, Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256. [Online]. Available: https://proceedings.mlr.press/v9/glorot10a.html.

[18] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, *Improving neural networks by preventing co-adaptation of feature detectors*, 2012. [Online]. Available: https://arxiv.org/abs/1207.0580.

[19] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, Jan. 2014, ISSN: 1532-4435.

[20] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[21] A. Krogh and J. Hertz, "A simple weight decay can improve generalization," in *Advances in Neural Information Processing Systems*, J. Moody, S. Hanson, and R. Lippmann, Eds., vol. 4, Morgan-Kaufmann, 1991. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/1991/file/8eefcfdf5990e441f0fb6f3fad709e21-Paper.pdf.

[22] L. Prechelt, "Early stopping - but when?" In *Neural Networks: Tricks of the Trade*, G. B. Orr and K.-R. Müller, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 55–69, ISBN: 978-3-540-49430-0. DOI: https://doi.org/10.1007/3-540-49430-8_3.

[23] O. Sagi and L. Rokach, "Ensemble learning: A survey," English, *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 8, no. 4, Jul. 2018, Publisher Copyright: © 2018 Wiley Periodicals, Inc., ISSN: 1942-4787. DOI: https://doi.org/10.1002/widm.1249.

[24] M. J. Azur, E. A. Stuart, C. Frangakis, and P. J. Leaf, "Multiple imputation by chained equations: What is it and how does it work?" *International journal of methods in psychiatric research*, vol. 20, no. 1, pp. 40–49, 2011.

[25] *Pyodbc*, 2008. [Online]. Available: https://pypi.org/project/pyodbc/.

[26] E. Jones, T. Oliphant, P. Peterson, *et al.*, *SciPy: Open source scientific tools for Python*, 2001–. [Online]. Available: http://www.scipy.org/.

[27] C. R. Harris, K. J. Millman, S. J. van der Walt, *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. DOI: https://doi.org/10.1038/s41586-020-2649-2.

[28] W. McKinney *et al.*, "Data structures for statistical computing in python," in *Proceedings of the 9th Python in Science Conference*, Austin, TX, vol. 445, 2010, pp. 51–56.

[29] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. DOI: https://doi.org/10.1109/MCSE.2007.55.

[30] M. L. Waskom, "Seaborn: Statistical data visualization," *Journal of Open Source Software*, vol. 6, no. 60, p. 3021, 2021. DOI: https://doi.org/10.21105/joss.03021.

[31] F. Pedregosa, G. Varoquaux, A. Gramfort, *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[32] A. Paszke, S. Gross, F. Massa, *et al.*, *PyTorch: An imperative style, high-performance deep learning library*, 2019. arXiv: 1912.01703 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1912.01703.

[33] S. Pandala, *Lazy Predict*, Accessed: 2025-04-17, 2021. [Online]. Available: https://github.com/shankarpandala/lazypredict.

[34] A. Tawakuli, B. Havers, V. Gulisano, D. Kaiser, and T. Engel, "Survey:time-series data preprocessing: A survey and an empirical analysis," *Journal of Engineering Research*, 2024, ISSN: 2307-1877. DOI: https://doi.org/10.1016/j.jer.2024.02.018.

[35] J. N. Wulff and L. Ejlskov, "Multiple imputation by chained equations in praxis: Guidelines and review," *Electronic Journal of Business Research Methods*, vol. 15, no. 1, pp41–56, 2017.

[36] L. B. de Amorim, G. D. Cavalcanti, and R. M. Cruz, "The choice of scaling technique matters for classification performance," *Applied Soft Computing*, vol. 133, p. 109 924, 2023, ISSN: 1568-4946. DOI: https://doi.org/10.1016/j.asoc.2022.109924.