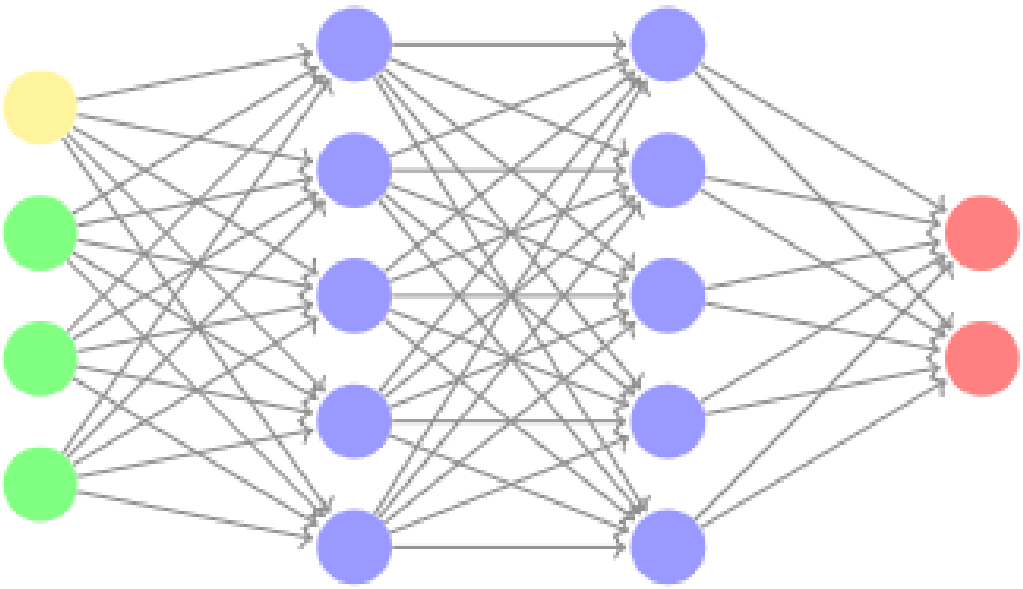
CHALMERS
UNIVERSITY OF TECHNOLOGY



Energy consumption prediction for electric buses using machine learning

Master's thesis in Infrastructure and Environmental Engineering

ANTONIA WISE

DEPARTMENT OF ARCHITECTURE AND CIVIL ENGINEERING

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2024
www.chalmers.se

Master's thesis 2024

Energy consumption prediction for electric buses using machine learning

ANTONIA WISE

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Architecture and Civil Engineering
Division of Geology and Geotechnics
Chalmers University of Technology
Gothenburg, Sweden 2024

Energy consumption prediction for electric buses using machine learning
ANTONIA WISE

Supervisor: Arsalan Najafi, Department of Architecture and Civil Engineering
Examiner: Kun Gao, Department of Architecture and Civil Engineering

Master's Thesis 2024
Department of Architecture and Civil Engineering
Division of Geology and Geotechnics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Visualisation of a basic Neural Network model.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2024

Abstract

With the increased adoption of electric buses, understanding their energy consumption (EC) has become crucial. For stakeholders such as city planners and bus company owners, having accurate predictions of energy consumption is essential for effective planning and resource allocation. Thus, identifying the relevant data to be collected for accurate predictions is of high importance. Machine learning models have emerged as the most promising tools for predicting energy consumption, offering the precision and reliability needed by stakeholders. Hence, this report aims to forecast energy consumption in electric buses by finding the important features in energy consumption and then exploring a suitable machine learning technique for the given data. Additionally, the report compares the selected model of Multilayered Perceptron Nueral Network (MLPNN) with two other models and assesses the impact of temporal factors on energy consumption predictions. To achieve this purpose, first, feature selection is conducted using correlation analysis and multicollinearity checks via the Variance Inflation Factor (VIF). The base MLPNN model is constructed using the Keras library in Python, with hyperparameter optimisation performed using GridSearch from the sklearn library. Afterward, the performance of the MLPNN model is compared to that of two other models: Random Forest (RF) and Extreme Gradient Boosting (XGB), using standard metrics such as Mean Square Error (MSE) and Mean Absolute Error (MAE). Feature importance is evaluated for each model, with the MLPNN model assessed using SHapley Additive exPlanations (SHAP). Temporal effects on features are also analysed. The features deployed in the model are: 'total mileage', 'speed', 'AC switch', 'outside temperature', 'inside temperature', 'run mileage', 'run duration', 'bus ID' and 'time category'. The optimal hyperparameters for the MLPNN model are: batch size of 20, 100 epochs, Stochastic Gradient Descent (SGD) optimizer, Rectified Linear Unit (ReLU) activation function, learning rate of 0.01, 2 hidden layers, 32 neurons per layer, and no regularisation. The evaluation shows that the MLPNN model, using the selected features and optimised hyperparameters, does not outperform the RF and XGB models in terms of MAE and MSE. Feature importance analysis reveals that while MLPNN provides stable importance measures, RF and XGB models are dominated by a single feature: a run mileage (the Euclidean distance between the origin and destination of trips) of over 50%. And secondly, run duration with 20%. SHAP analysis suggests that Run duration and run mileage are most significant for MLPNN as well. When examining the temporal impact on features, no features are impacted by time, contrary to initial expectations that speed would show a substantial temporal effect. The study concludes that the MLPNN model, as constructed, is not significantly better than simpler models in predicting the energy consumption of electric buses for the given dataset. However, there is potential for improvement with additional features or more training data. Future research should explore the inclusion of other relevant features and larger datasets to enhance model performance.

Keywords: Energy Consumption; Electric Buses; Machine Learning; Multi Layered Perceptron Neural Network; Random Forest; eXtreme Gradient Boosting; Correlation; Validation Inflation Factor; SHapley Additive exPlanations.

Acknowledgements

I would like to thank my supervisor Arsalan Najafi and also the extra help from Omkar Parishwad for having the patience with me while I learnt programming and machine learning from scratch to complete this thesis. It has been an opportunity to learn lots of new skills that I hope will be helpful in the future.

Antonia Wise, Gothenburg, June 2024

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

Acronym	**Definition**
Adam	Adaptive Moment Estimation
AdaBoost	Adaptive Boosting
BES	Battery Energy Storage
BEB	Battery Electric Bus
CNN	Convolutional Neural Network
Deep MLP	Deep Multilayer Perceptron
DL	Deep Learning
DNN	Deep Neural Network
EC	Energy Consumption
EV	Electric Vehicle
GBDT	Gradient-Boosting decision tree
GHG	Greenhouse Gas Emissions
HVAC	Heating, Ventilation, and Air Conditioning
ICE	Internal Combustion Engine Bus
LightGBM	Light gradient boosting
LSTM	Long-Short Term Memory
MAE	Mean Absolute Error
ML	Machine Learning
MLP	Multilayer Perceptron
MLP-NN	Multilayer Perceptron Neural Network
MLR	Multiple Linear Regressor
MSE	Mean Square Error
RBF	Radial Basis Function
RB-NN	Radial-basis neural network
RF	Random Forest
ReLU	Rectified Linear Unit
RMSPRop	Root Mean Square Propogation
SGD	Stochastic Gradient Descent
SHAP	SHapley Additive exPlanations
SOC	State of Charge
SVR	Support Vector Regression

Acronym	**Definition**
VIF	Variance Inflation Factor
XGB	Extreme Gradient Boosting
MAE	Mean Absolute Error
LightGBM	Light Gradient Boosting
kernelSVR	Kernel Support Vector Regressor
GBDT	Gradient Boosting Decision Tree
RB-NN	Radial Basis Neural Network

Nomenclature

Below is the nomenclature of indices, parameters, and variables that have been used throughout this thesis.

Indices

l	layer number
j	neuron number start
k	neuron number end

Parameters

m	Vehicle mass
SoC	State of Charge
C_r	Rolling resistance
C_d	Drag coefficient
acc/dec	Acceleration/Deceleration
V_a	Average speed
SN	Number of Stops
L	Route Length
g	Road Gradient
T_a	Ambient Temperature
$HVAC$	Heating, Ventilation, and Air Conditioning
$Aux.$	Auxiliary Power
FI	Feature importance

Variables

a	Activation function
b	Bias
C	Cost function
f	predicted output, final or partly. prediction function
L	total number of layers
n	Number of iterations
N	Number of trees or features
N	Number of neurons in first layer
v	Weights and biases
w	Weight
x	Neuron, Input
y	Correct value
z	Output
δ	error
σ	Sigmoid function
η	Learning rate
ϕ	SHAP value

Contents

List of Figures

List of Tables

1
Introduction

The transition to sustainable transportation is critical in addressing global environmental challenges. Among the innovations leading this change, electric buses represent a promising solution for reducing urban emissions. This thesis explores the potential of machine learning techniques in predicting energy consumption for electric buses, aiming to enhance efficiency and operational planning.

1.1 Background

Vehicle emissions pose a substantial environmental impact and according to Ember vehicles cause 23% of CO2 emissions [1], prompting numerous countries to promote electrification. The International Energy Agency states that the Energy consumption is rising and that renewable energy and shifting to Electric vehicles are one of the many strategies for reaching the goal of zero emissions [2]. Congestion on roads is another huge challenge, leading many large cities to actively promote public transport as a solution. To address both challenges, the city of Guangzhou in China is actively promoting both electrification and public transport solutions. The challenge is that, despite recognising the necessity of electrification, accurately predicting the energy required for a trip remains a difficult task. Various stakeholders are reliant on this information for informed decision-making. We have received data from 5890 buses in Guangzhou that will be used to predict the energy consumption.

1.2 Aim

The aim of the project is to identify which machine learning model would be suitable to predicting energy consumption in electric buses. The prediction should be done trip wise and not continuously, so that the stakeholders can have the prediction before the trip is conducted. By researching what type of models exist and have been used for similar tasks it would be possible to identify what features will be needed. This could be vehicular and operational data such as State of Charge (SoC), acceleration and speed but also topological and external data such as temperature, direction and length of route. Then, develop a machine learning model incorporating the identified features and identify hyperparameters to enhance the accuracy and comprehensiveness of the model. Furthermore, evaluate the features and their impact on energy consumption for certain situations using the developed model. And finally interpret the model compared to other models, providing valu-

able information for stakeholders and contributing to the understanding of energy consumption patterns in electric buses.

1.3 Limitations

The vehicular and operational data, such as speed, acceleration, deceleration, state of charge (SoC), is limited to the 5890 buses in Guangzhou, China. The topological and external data, such as latitude, longitude and temperature is therefore limited to the same region. The report will only be evaluating one model, a Multi Layered Perceptron Neutral Network (MLPNN), which is decided in the literature study. The other models presented are only for comparison. The results are also to be predicted tripwise, and not continuously.

1.4 Research questions

The following questions will be answered to meet the aim of the project.

- Which machine learning model is suited to predict tripwise energy consumption in electric buses?
- What features are needed for predicting tripwise energy consumption in electric buses?
- What is the contribution and importance of each feature in the prediction of energy consumption in electric buses?
- How does the model compare to other models using standard metrics MAE and MSE?
- Does the time of day impact the model and if it does which features are impacted when?

1.5 Literature study

Compared to an Internal Combustion Engine Bus (ICE) the Battery Electric Bus (BEB) has to consider other features in planning for a small and large scale. From the decisions the driver or automatic system has to make when driving to the planning of the city and where to put chargers for the buses. According to Abdelaty & Mohamed there are three types of features to consider in planning; economic, operational and environmental [3]. The first includes the costs of the bus, battery, chargers, etc. The costs for chargers and batteries are the highest and are for investors of infrastructure important to know the cost of [3]. Being able to choose the right charger and battery for the area requires knowledge of the energy consumption. The second feature consists of all the parameters that are connected to the operation of the bus, such as charger availability and charging time, speed, and energy consumption of the bus. All are needed for planning by the city, driver, investor, or other stakeholders. Also, here the energy consumption is a crucial factor for the rest of the parameters. The third feature is the environmental impact of the BEB. It consists of measures of air quality, greenhouse gas emissions, etc. They are essential

not only for the city but also the municipality, country and world as whole and the people living on it. It often comes with legal requirements and taxes that add to the economic feature. Also, it is a health factor that, in the long run, can also be an economic factor. Most importantly is the leading environmental impact that should be prevented.
In summary the stakeholders, from driver, owner, investor, planner, municipality, country to everyone affected, require that the energy consumption is accurately predicted so they can plan according to the laws and economy.
The work started with a literature study to gain knowledge on the topic and decide on a model to use. The literature was selected by the examiner and supervisor, and further reading was done on the literature stated in the given literature. There were two main goals in the study: to find out which parameters are important to consider when predicting energy consumption, and which models are of high accuracy when predicting the energy consumption in electric buses.
In the report *Machine learning prediction models for battery-electric bus energy consumption in transit* [4] by Abdelaty et al. they reviewed 33 reports and discussed which parameters are necessary when predicting energy consumption using 7 different models. The parameters were as follows:

- Vehicular
  - Vehicle mass [m]
  - State of Charge [SoC]
  - Rolling resistance [C_r]
  - Drag coefficient [C_d]
- Operational
  - Acceleration/Deceleration [acc/dec]
  - Average speed [V_a]
  - Number of Stops [S_N]
- Topological
  - Route Length [L]
  - Road Gradient [g]
- External
  - Ambient Temperature [T_a]
  - Heating, Ventilation, and Air Conditioning, [HVAC]
  - Auxiliary Power [Aux.]

Not all of these parameters were of equal necessity for all types of models that they studied and there might be other parameters that have an impact on energy consumption that were not taken into consideration.

In the report by Roy et al.[5], the energy consumption of electric vehicles was explored. The authors divided their parameters into 4 categories depending on whether they were given before or after the trip (pre-trip and on-trip features) and if they were logged directly or if they had to be calculated (engineered). The engineered parameters are mostly time-dependent, such as traffic conditions that change during peak hours. But also spatially dependent, such as altitude and temperature. They also evaluated the driving behavior that was calculated from the braking and

acceleration parameters. These types of parameters were the ones that made the model more reliable with fewer errors in prediction compared to when they were not accounted for, with 3-4% less errors.
The prediction model gets more complicated when some parameters are needed to be calculated. Being able to predict the energy consumption pre-trip with fewer parameters being given is to be preferred for planners, however, only if the accuracy is good enough.

After calculating the importance of the features, the Model was then to be decided. Six reports, including Abdelaty et al.[4] that were already studied, were reviewed. In total, there were 24 types of models, as follows:

- Robust Regression
- 1stOpt
- Linear Regression
- Neural Network Regression
- Gaussian Process Regression
- Ensamble Regression
- Support Vector Machine Regression
- Decision Tree Regression
- Extreme gradient boosting (XGB)
- Random forest (RF)
- Multilayer perceptron (MLP)
- Support vector regression (SVR)
- Multilayer Perceptron Neural Network (MLP-NN)
- Multivariate Nonlinear Regression Model
- Long-short Term Memory (LSTM) Time Series Prediction Model
- Convolutional Neural Network (CNN) Time Series Prediction Model
- Multiple Linear Regressor (MLR)
- Kernel support vector regressor (kernelSVR)
- Adaptive boosting (AdaBoost)
- Light gradient boosting (LightGBM)
- Deep Multilayer perceptron(Deep MLP)
- Radial basis function interpolation model (RBF)
- Gradient-Boosting decision tree (GBDT)
- Radial-basis neural network (RB-NN)

When evaluating their accuracy, the metrics given by the authors were compared. All reports, except [6], used the R^2 metric to validate their methods. The reports usually used more than one metric, but having a common one made comparison easier. The table of the accuracy of the models that used R^2 metric for accuracy is given in Table 1.1.Were green highlights show the models with R^2 value above 97%. The Models of highest accuracy, over 97% were linear Regression [7], Neural Network Regression [7], Gaussian Process Regression [7], Multilayer Perceptron Neural Network (MLP-NN) [3], Adaptive Boosting [5], Light gradient boosting [5] and Multilayer perceptron neural network model (MLP-NN) [4].

Table 1.1: Categorised List of Models with References.

Model Type	Par.	R^2	Ref.
Non-linear Regression Models			
Robust regression	4	max 0.9624	[8]
1stOpt	4	max 0.9401	[8]
Gaussian Process Regression	8	0.98	[7]
Linear Regression Models			
Linear Regression	8	0.99	[7]
Multiple Linear Regression Model (MLR)	12	0.943	[4]
Multiple Linear Regressor (MLR)	7	0.952	[5]
Neural Network Models			
Neural Network Regression	8	0.98	[7]
Radial-Basis Neural Network (RB-NN)	12	0.5458	[4]
Decision Tree Models			
Decision Tree (DT)	7	0.956	[5]
Decision Tree Regression	8	0.92	[7]
Decision Tree model (DT)	12	0.946	[4]
Random forest (RF) perceptron	7	0.957	[5]
Boosting Models			
Extreme gradient boosting (XGB)	7	max 0.9136	[9]
Gradient-Boosting decision tree (GBDT)	12	0.942	[4]
Light Gradient Boosting (LightGBM)	7	0.99	[5]
Adaptive boosting (AdaBoost)	7	0.993	[5]
Ensemble Regression	8	0.95	[7]
Support Vector Machines (SVM) Models			
Support Vector Machine Regression	8	0.93	[7]
Support Vector Machine Learning	12	0.946	[4]
Kernel Support Vector Regressor (kernelSVR)	7	0.952	[5]
Support Vector Regression (SVR)	7	max 0.6994	[9]
Multilayer Perceptron Models			
Multilayer Perceptron Neural Network (MLP-NN)	11	0.993	[3]
Multilayer Perceptron Neural Network (MLP-NN)	12	0.971	[4]
Deep Multilayer Perceptron(Deep MLP)	7	0.955	[5]
Multilayer Perceptron (MLP)	7	max 0.9221	[9]
Interpolation Models			
Radial-Basis Function interpolation model (RBF)	12	0.952	[4]

As seen in Table 1.1, having more parameters does not necessarily result in higher accuracy, which [4] also states. During the development of the model in this report, many of these parameters will be accounted for at the start and, by correlation, be removed throughout the process. This goes for both directly logged and calculated parameters.
Looking at the accuracy, two reports state that a Multilayer Perceptron Neural Network model (MLP-NN) had accuracy above 97% [3][4]. One reported that a Neural Network regression had accuracy above 97%. A Multilayer Perceptron Neural Network (MLP-NN) model will therefore be developed in this report. It can be noted that the less complex models Gaussian process regression, Linear regression, and ”normal” Neural Network Regression also have high accuracy [7]. These three models are, however, from the same report and have no comparisons to compare to except similar methods that do not have as high accuracy. These methods will there for not be developed further. It is also noted that boosting can increase the accuracy [5]; therefore, the boosting aspect is also taken into account in this report. The two models decided for comparison are Random forest and Extreme Gradient Boosting. One is a basic model, and the other uses boosting.

2

Theory

This chapter contains the theory required to understand the following chapters. It describes the basics of machine learning and the structure of the models used later in this report.

2.1 Choosing features

Before starting a model, one needs to know which features could be used for the model. Literature study gives insight on what other researches have found to be useful features, but ones own data needs to be evaluated as well.
Correlation analysis serves as a foundational step in clarifying the relationships between various features. By getting insight into the correlations, features demonstrating significant associations with other variables can be identified and given priority for further inspection if they can be of use as a feature in the final model. Following the identification of prospective features through correlation analysis, attention must be directed towards addressing multicollinearity. A phenomenon arising from high intercorrelations among independent variables. Not only the correlation between two variables but also the impact of more of them on each other. Multicollinearity poses a substantial challenge, as it promotes instability in the estimation of model coefficients, thereby compromising the reliability of results. To mitigate multicollinearity, the Validation Inflation Factor (VIF) serves as a practical tool. VIF quantifies the degree to which the variance of an estimated regression coefficient is amplified owing to multicollinearity. By computing VIF for each variable, it becomes feasible to discern and improve multicollinearity, thereby eliminating variables that contribute disproportionately to the inflation of variance in regression coefficients.
In crafting a robust model, it is essential to navigate the delicate balance between correlation and multicollinearity. While correlation analysis provides valuable insights into feature relevance, the possibility of multicollinearity poses a significant concern, potentially destabilising the model's predictive power. By integrating techniques like VIF, we not only refine feature selection but also fortify the model against multicollinearity's negative effects. This systematic approach ensures that selected features are not only relevant but also resilient against redundancy, thereby enhancing the model's predictive effectiveness without compromising its integrity.

2.2 Building the main model

After selecting the features, the next step is to employ a model for predicting the desired outcome. The primary focus of this report lies on the Multi-Layered Perceptron Neural Network (MLPNN). It will be contrasted against two fundamental models, namely Random Forest and Extreme Gradient Boosting. This comparative analysis ensures consistency by utilising the same dataset, rather than relying on models from disparate literature sources with varying datasets. This chapter aims to provide a comprehensive description of the MLPNN. Beginning with an explanation of its structure, followed by insights into its training process, and concluding with a commonly used calibration method.

2.2.1 Neural Network

A Neural network consists of three main parts: Neurons x_j, weights w_j, and layers. Layers always have an input and output layer, being the input data and the result respectively. The hidden layers are between the input and output. There can be many of these layers in a neural network. A neuron is each single parameter, either the pure input data, the resulted calculated output data, but also all the calculated parameters developed in the hidden layers. In figure 2.1 all the circles represents neurons. The weights are partially the lines connecting the neurons. High weights are given to inputs of higher importance. When the machine calculates in the hidden layer it takes the sum of each input neuron times its weight.

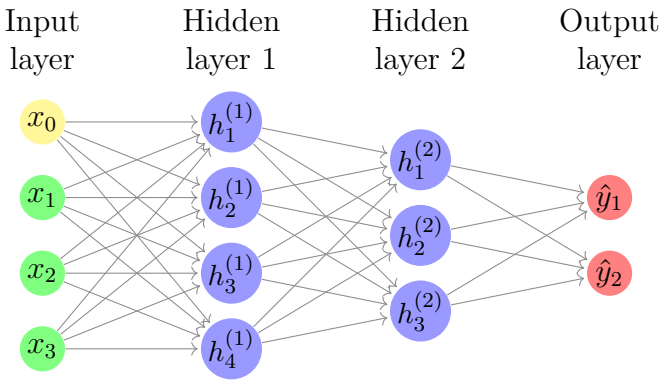


Figure 2.1: Basic Neural Network model

2.2.1.1 Perceptrons

Perceptrons are a type of artificial neuron [10] of the most basic kind. It takes several binary inputs from the input layer and produces a single binary output in the output layer as in figure 2.2. It uses the simple equation 2.1, where b is the bias, not to be confused with the weight w_j that affects a single neuron x_j. The bias affects the overall decision of the perceptron, all the inputs and their weights combined.

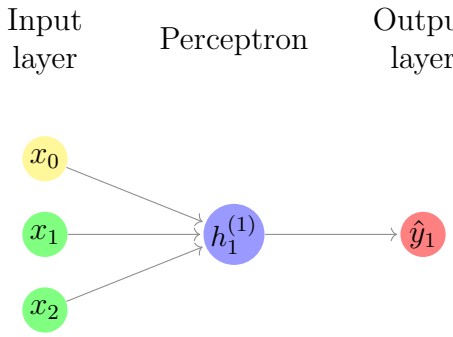


Figure 2.2: Basic Perceptron

$$output = \begin{cases} 1, & \text{if } \sum_j w_j x_j + b \leq 0 \\ 0, & \text{if } \sum_j w_j x_j + b > 0. \end{cases} \tag{2.1}$$

2.2.1.2 Sigmoid function and neurons

Sigmoid neurons are also artifical neurons [10]. They have the benefit over perceptrons that they are not affected as much of small changes in weight and bias that the result can completely change. Instead a small change in weight or bias only results in a small change in output. This is due to it being able to handle more than just binary numbers, instead it handles the range in between as well. It can do this due to the sigmoid function. Just like the bias handled the output for both the weight w_j and input x_j, the sigmoid function handles the variation for all of them as in 2.2 where $z = \sum_j w_j x_j + b$ the original perceptron equation 2.1. The sigmoid function can be seen as a smooth version of the perceptron as in figure 2.3.

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}} \tag{2.2}$$

There are several other artficial neuron functions that just like the sigmoid function changes how and when the neuron will be activated and therefore the result of the output. Some of the most common are as follows in eq 2.3 - 2.5 and also shown in figure 2.3.

$$\tanh(z) \equiv \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{2.3}$$

$$ReLU(z) \equiv max(0, x) \tag{2.4}$$

$$LeakyReLU(z) \equiv max(\alpha x, x) \tag{2.5}$$

2.2.2 Validation

Data is divided into three parts when developing a model. One used for training the model, one for testing and the rest is the real world data when using the model.

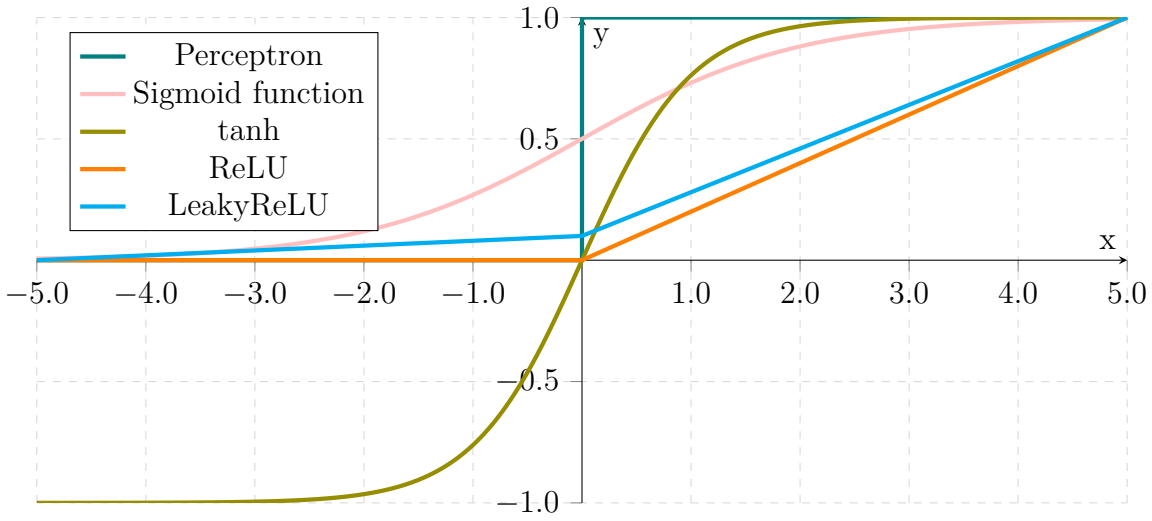


Figure 2.3: Plots of activation functions

It is important that the training and testing data is not the same, since there has to be a way to check that the model can handle data that it has never seen before.

2.2.2.1 Training

To train a model you need to go through five steps: input, feedforward, compute error, backpropagation and output. Different functions were discussed earlier, they will be denoted as the activation function a_j^l where l is the layer and j is the neuron. The activation function updates equation 2.1 with the neuron functions 2.2 - 2.5 to equation 2.6. The components are vectors and are therefore denoted without the j indication for the neuron. It is also to be noted as before that $z^l = \sum w^l x^l + b^l$.

$$a^l(z^l) = \sigma(w^l x^{l-1} + b^l) \tag{2.6}$$

If the model gives the right output during training, a cost function (sometimes called loss or objective function) is used for validation[10]. There are several different kinds, but the most basic one is the Mean Square Error (MSE) as in equation 2.7, where n indicates the number of training examples. It is simply the difference between the predicted value, a, given as the output from the model, and the actual correct value, $y(x)$. This correct value has to be known during training to validate that the model is correct. If the result is close to 0 then the result is sure to be correct.

$$C(w,b) = \frac{1}{2n} \sum_x ||y(x) - a||^2 \tag{2.7}$$

When using a cost function that relies on the result getting as small as possible the training will require to get weights and biases that make the cost function small, a minimisation problem. There are several optimisation algorithms such as Root Mean Square Propogation (RMSPRop), Adaptive Gradient Algorithm (AdaGrad) and Adaptive Moment Estimation (Adam) to name some. The most common way to do this is by Stochastic Gradient Descent (SGD), as equation 2.8. ∇C is the

gradient of the cost function, simply put it is the multidimensional derivative in calculus. This we want to become close to 0. Δv are the changes in the weights and biases, there could technically be more variables as well. The learning rate, η, is added to compute it in several small steps. There are several other methods as well, but this is the most basic one.

$$\Delta v = -\eta \nabla C \tag{2.8}$$

Calculating the gradient however requires one to calculate the partial derivatives of the cost function. This is done through backpropagation and requires four equations 2.9 - 2.12. k is just like j a neuron, if j is the starting neuron when counting in a layer then k is the final one so from j to k. if l is the layer then L is the total number of layers. $\odot$ is the Hadamard product, an elementwise product of vectors. The first equation 2.9 is the equation for the error in the output layer superscripted by L. The $\frac{\partial C}{\partial a_j^L}$ part is the rate of change of the cost function with respect to the activation function. It can also be read as $\nabla_a C$. $\sigma'(z^L)$ is how fast the activation function σ changes at each layer and neuron.

$$\delta^L = \frac{\partial C}{\partial a_j^L} \odot \sigma'(z^L) \tag{2.9}$$

The second equation 2.10 calculates the error in terms of the error in the next layer. This is the step that gives the name to backpropagation. You move backwards through the network and getting the error δ^l for each $l+1$ layer along the way. First 2.9 is used to calculate the error for the final output layer then 2.10 to calculate the rest, from the back (the output layer) all the way to the front (input layer).

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{2.10}$$

The third equation 2.11 calculates the rate of change of the cost with respect to the biases. The two earlier equations 2.9 and 2.10 are used to calculate this, since it gave δ_j^l.

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{2.11}$$

The final equation is 2.12, which calculates the rate of change of the cost with respect to the weights. Compared to 2.11, it also requires the activation function for each neuron a_k^{l-a}, not only the error.

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \tag{2.12}$$

To summarise, there are five steps to train a model input, feedforward, compute error, backpropagation and output [10].

Input: First the activation, a^1, is set for the input layer.

Feedforward: Secondly $z^l = w^l a^{l-a} + b^l$and $a^l = \sigma(z^l)$ are calculated using equation 2.6.

Compute error: Thirdly the output error is calculated using equation 2.9

Backpropagation: Forthly backprogation through all the layers using equation 2.10.

Output: Finally the output is given by 2.11 and 2.12 which gives the the gradient of the cost function.

How often the model is updated using this method is defined by number of epochs (iterations) but also batch size, which is the number of samples used for each iteration of the model.

2.2.2.2 Testing

The testing does not change the model structure like training does, it simply tests the model for accuracy compared to the actual correct answer using metrics. There are several different metrics to evaluate a model. We use some common ones in this report, including Mean Absolute Error (MAE) , Mean Squared Error (MSE) , Root Mean Squared Error (RMSE) and R-squared (R^2) as given in equations 2.13 - 2.16.

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{2.13}$$

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.14}$$

$$\text{RMSE} = \sqrt{\text{MSE}} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{2.15}$$

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}i)^2}{\sum i = 1^n(y_i - \bar{y})^2} \tag{2.16}$$

2.2.3 Calibration

The calibration is an iterative process that is done until the model predicts the way that is required for the task. Defining how good of a model is needed and for how long this process continues depends on the model. The features decided before constructing the model are usually optimised to fit the model better, finding the parameters that works best for the model can be done in several ways. Since the model does not think in the traditional human way, some of the features decided before the model was constructed simply do not help the model or makes it worse. Another common problem is overfitting. It occurs when a model learns to capture noise or random fluctuations in the training data, leading to poor generalisation performance on unseen data. This phenomenon can arise when the model becomes too complex relative to the size and complexity of the training dataset. Adjusting hyperparemters can both make overfitting better and worse, so this has to be taken into consideration when adjusting them. One study suggest using the default parameters since the adjustment sometimes does not improve the model to a large enough extent and is non-inferior [11]. However, it is still commonly done and is therefore discussed here.

In a MLPNN model there are several hyperparameters that can be adjusted to improve the model. The most basic ones include the number of hidden layers, the number of neurons in each layer, the activation functions used in each layer, the learning rate, the batch size, the number of epochs and the optimisation algorithm. Manual tuning is often used in student research, but there are more defined methods that are less prone to error, such as the one suggested by Yang and Shami[12]. This is often called Hyperparameter Optimisation instead of tuning. It reduces human error, makes the model more accurate, and makes the research more easily reproducible. Yang and Shami state in their report the six steps of hyperparameter optimisation to be;

1. Select the objective function and the performance metrics;
2. Select the hyper-parameters that require tuning, summarise their types, and determine the appropriate optimisation technique;
3. Train the ML model using the default hyper-parameter configuration or common values as the baseline model;
4. Start the optimisation process with a large search space as the hyper-parameter feasible domain determined by manual testing and/or domain knowledge;
5. Narrow the search space based on the regions of currently tested well-performing hyper-parameter values, or explore new search spaces if necessary.
6. Return the best-performing hyper-parameter configuration as the final solution.

Some of the most common methods used are; Grid search, which is a brute force method that tests all the different combinations of hyperparameter. A less time consuming version is Random search which does not test all types of combinations but only some. A more complex method is Bayesian optimisation that requires two models that iteratively updates each other. And many more that will not be discussed in this report.

2.3 Comparison to other models and Evaluation

This report introduces three prominent models utilised for analysis. There are two Ensemble models among them, including the Random Forest and Extreme Gradient Boosting models. Complementing these is a more intricate model known as the Multi-Layer Perceptron Neural Network (MLPNN).
The Random Forest and Extreme Gradient Boosting models offer robust yet accessible frameworks for analysis. Their suitability lies in their ability to handle a wide range of data types and complexities while providing reliable predictions. These models are capable of capturing nonlinear relationships within the data and are often chosen for their versatility and ease of implementation. In contrast, the MLPNN represents a deeper exploration into the complexities of neural network architectures. Its design incorporates multiple layers of interconnected nodes, enabling it to learn intricate patterns and relationships within the data. While more computationally intensive, the MLPNN offers unparalleled flexibility and predictive power, making it the focal point of evaluation in this report.
Various approaches can be employed when comparing and evaluating different mod-

els. A fundamental method involves utilising the metrics provided by the models during prediction. However, when assessing the models individually, additional methodologies come into play.
In this section, two methods will be further explained: Feature Importance and SHapley Additive exPlanations (SHAP). The former offers a broad and user-friendly approach, while the latter is tailored towards neural networks with hidden layers. Through the examination of these methods, a comprehensive evaluation of the models can be achieved, allowing for insights into their performance and underlying mechanisms.

2.3.1 Random Forest

Random forest models are an ensemble learning method used for classification and regression tasks in machine learning. They are one of the most basic models used. Random Forest models offer advantages such as robustness to overfitting, scalability to large datasets, and high predictive performance across various domains. RF consist of multiple decision trees, each trained on a different subset of the training data and making independent predictions. The final prediction is typically determined by averaging the predictions of all individual trees (for regression) or using a majority voting scheme (for classification). Mathematically, a Random Forest model can be represented as follows in Equation 2.17:

$$\hat{f}(x) = \frac{1}{N} \sum_{i=1}^{N} f_i(x) \tag{2.17}$$

where $\hat{f}(x)$ is the predicted output, N is the number of trees in the forest, and $f_i(x)$ is the prediction of the i-th tree.
Each decision tree in the Random Forest is built using only a portion of the available data and considering only a few random features at a time. The goal of splitting is to create groups of data points that are as similar as possible. This similarity is measured by criteria like how often a particular feature appears in a group or how much uncertainty exists within a group.
Figure 2.4 illustrates a basic decision tree model, which serves as one of the constituent trees in a Random Forest. The tree starts at the root node (x) and recursively splits the feature space based on feature values (d_1, d_2, etc.). Each internal node represents a decision based on a specific feature, and each leaf node represents a final decision or prediction.

2.3.2 Extreme Gradient Boosting (XGBoost)

Extreme Gradient Boosting is a more extreme version of gradient boosting. Gradient boosting is similar to RF in many ways since they both use decision trees as figure 2.4 to reach a result. The key difference is that RF builds several independent decision trees and takes the average. However, Gradient boosting starts with one tree, updates it by correcting errors to make a new one and goes on until the model is no longer improving and then has the final tree as a result. The Extreme Gradient

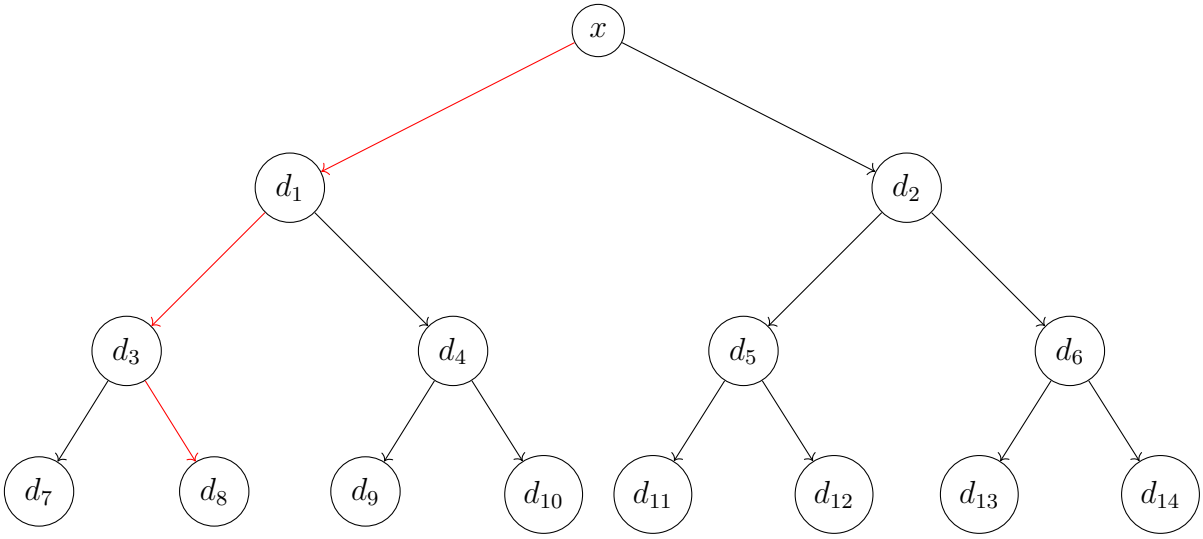


Figure 2.4: Basic Decision Tree model

Boosting model adds optimisation, regularisation and other valuable techniques that improves the many flaws of the standard gradient boosting model. For comparison between models, standardised metrics such as Mean Square Error (MSE) and Mean Average Error (MAE) are often used.
For evaluating the significance of features in a model, feature importance is often used. However, for more complex models with hidden layers such as MLPNN , there are other methods for evaluation. One being SHapley Additive exPlanations (SHAP).

2.3.3 Feature Importance

Feature importance is a technique used to assess the significance of each feature in a machine learning model's predictions. It helps identify which features have the most influence on the model's output.
One way to calculate feature importance is to examine the weights assigned to features in the model's parameters. In the case of neural networks, the weights of the first layer are often used for this purpose. Feature importance is typically calculated as the proportion of the total sum of weights attributed to each feature. Mathematically, the feature importance (FI_i) of feature i can be computed as equation 2.18:

$$FI_i = \frac{\sum_{j=1}^{M} |w_{ij}|}{\sum_{i=1}^{N} \sum_{j=1}^{M} |w_{ij}|} \times 100 \tag{2.18}$$

where:

- w_{ij} represents the weight of feature i in neuron j of the first layer.
- N is the total number of features.
- M is the number of neurons in the first layer.

This formula calculates the importance of each feature as a percentage of the total absolute sum of weights in the first layer of the neural network.

2.3.4 SHapley Additive exPlanations (SHAP)

SHapley Additive exPlanations (SHAP) is a method used in machine learning to explain the output of black-box models, such as the MLPNN model. It provides a way to understand the contribution of each feature to the model's prediction. To use the SHAP method, a trained machine learning model and the corresponding dataset are needed.
The first step in using SHAP is to create an explainer object. This explainer object acts as a bridge between the model and the SHAP library. It encapsulates the model's functionality and allows SHAP to generate explanations for its predictions. Once the explainer is created, it is used to calculate SHAP values for the dataset. SHAP values quantify the impact of each feature on the model's predictions. They provide insights into how the model arrives at its decisions and which features are most influential in driving those decisions. To calculate SHAP values, one starts with a dataset X consisting of N samples, where each sample x_i is a vector of features. This dataset is passed to the explainer's 'shap_values' method, which computes the SHAP values for each sample in the dataset.
For a given sample x_i, the SHAP values represent the contribution of each feature to the difference between the model's output for x_i and the expected output. Mathematically, the SHAP value ϕ_{ij} for feature j and sample i is defined as equation 2.19:

$$\phi_{ij} = \frac{1}{N} \sum_{k=1}^{N} \left[f(x_i^{(k)}) - f(x_i^{(\neg j,k)}) \right] \tag{2.19}$$

where:

- f is the model's prediction function.
- $x_i^{(k)}$ is the k-th permutation of sample x_i, where the feature j is masked (i.e., set to a reference value).
- $x_i^{(\neg j,k)}$ is the k-th permutation of sample x_i, where all features except j are masked.

In simpler terms, the SHAP value for a feature and sample represents the average difference between the model's predictions when the feature is included and when it's excluded, over all possible combinations of features.
By computing SHAP values for each sample in the dataset, a set of explanations that reveal how each feature contributes to the model's predictions across different samples is generated. These explanations can then be used to interpret the model's behavior, understand feature importance, and identify patterns in the data.
After calculating SHAP values, one can visualise the results using various plots and visualisations provided by the SHAP library. These visualisations help you understand the relationships between features and predictions, identify important features, and detect patterns in the data.

3
Methodology

This chapter outlines the process of data preparation, model construction, optimisation, and evaluation, ending it with a comparative analysis against alternative models. To begin with, the focus is on comprehending the available dataset and ensuring its alignment with the desired outcomes. Following that are feature selection techniques, including correlation analysis and VIF. They are used to discern relevant variables. The subsequent stages involve the construction of the model, followed by optimisation to enhance performance. Lastly, the MLPNN model undergoes evaluation and comparison with the Random Forest and XGB models. These processes were executed using Python in the VS Code environment, ensuring methodological precision and reproducibility. The code is provided in its complete format in the appendices.

3.1 Data

The forthcoming section will provide a detailed overview of the data collected. It will comprise the steps involved in the data handling process, as well as the methodology applied for feature selection utilising correlation and VIF analysis.

3.1.1 Data description

The data was gathered in Guangzhou, China, on May 1, 2021, from 5890 electric buses operating along designated routes, depicted as black lines in the study area map in Figure 3.1. The dataset comprises three distinct files: 'CAN data' (which contains the main operational data. We respect the original name of the file and call it *CAN data* for the rest of the report), 'GPS data', and 'Timetable data'. The CAN data encompasses essential information and charging details for electric bus equipment, including vehicle location, collected at 1-second intervals. For May 1, 2021, a total of 18,202,522 data entries were recorded. The GPS data tracks location specifics during bus operation, collected concurrently with CAN data on May 1, 2021. The data intervals range from 15 seconds to 15 minutes, yielding a total of 2,488,593 entries. The Departure Data includes departure schedules, encompassing departure and arrival times, duration, mileage, and station names for each route. For May 1, 2021, there were 5890 entries, with intervals ranging from 0 to 15 minutes. Tables 3.1, 3.2 and 3.3 provides a breakdown of these values.
To aid in understanding of the data, it was important to grasp the data types and the range of values they covered. Following Abdelaty et al.'s approach for consistency

Figure 3.1: Map of the study area with routes shown in black lines.

Table 3.1: Vehicle Information

Category	**Value**
Total Number of Involved Vehicles	5890 vehicles
Total Number of Routes	470 routes
Number of Non-duplicate Routes	141 routes
Total Service Coverage	Refer to the map

Table 3.2: Data entries information

Data Type	**Number of Entries**
Departure Information	5890 entries
GPS Information	2,488,593 entries
Electric Bus Mobile Equipment Data	18,202,522 entries

Table 3.3: Interval information

Data Type	Transmission Interval
Departure Information	0 to 15 minutes
GPS Information	15 seconds to 15 minutes
CAN Data	1-second intervals for the same vehicle

with previous research [4], the data was classified into four categories: Vehicular, Operational, Topological, and External. These categories are clarified in Tables 3.4, 3.5, 3.6, and 3.7. These tables describe the data types and the corresponding ranges of values for each category, offering comprehensive insights into the dataset. Additionally, Tables 3.8 and 3.9 provide the same kind of details regarding GPS data and Departure data, respectively.
It's worth noting that not all features fall under numerical data types such as int64 and float64; some are categorised as objects. These include ID names for routes and stations, often represented in Chinese characters. Additionally, features such as times and RPMs of motors are classified as object datatypes, despite appearing as numbers, owing to the formatting of the values in the data. Hence, an example of object data is provided in this column.

3.1.2 Handling the data

Prior to selecting the data to be incorporated into the model using correlation and VIF, the datasets were merged into a single file for more efficient management. At this stage, it was determined that the GPS data would be omitted entirely, as it did not provide any unique information not present in the other two files. The files were merged by matching the time in the CAN data to the time in departure data. Since they are not is the same intervals the CAN data which is collected each second ('local_time') finds a match in the ranges in the begin and end time ('triplog_begin_time' and 'triplog_end_time') of the departure data. The bus ids ('busid') are also matched to make sure that the data is from the same bus.For more details, refer to the code in A.1. The time matching procedure accommodated the interval difference between the two data files, with CAN data collected every second and departure data at intervals of up to 15 minutes. Since the objective is to predict energy consumption, the feature of the state of charge (SOC) was essential. Energy consumption (EC) was derived by computing the difference in SOC for each time step, thus introducing it as a new feature.
The desired outcome necessitates the data to be aggregated within the interval of a trip, spanning from the beginning to the end of a trip between two stops within the route. To achieve this, the data was aggregated using predefined functions for each feature, as outlined in Tables 3.10, 3.11, 3.12 and 3.13. For further details, refer to the code provided in A.2.
Another feature added was a time category called 'time_cat', which gave the trip

Table 3.4: Table of CAN data of category 'External Features' and their types with descriptions.

Feature	**Description**	**Data Type**	**Range or example**
ac_set_temperature	AC Set Temperature	float64	30
ac_switch	AC Switch	float64	0 - 1
ac_workmode	AC Work Mode	float64	0 - 41
inside_car_temperature	Inside Car Temperature	float64	-273 - 52
leftctrl_radiatortemp	Left Control Radiator Temperature	float64	0 - 58
leftctrl_temp	Left Control Temperature	float64	-256 - 55
local_time	Local Time	object	e.g 20210501 181250
outside_car_temperature	Outside Car Temperature	float64	-273 - 97
redis_time	Redis Time	object	e.g 20210501 181248
rightctrl_radiatortemp	Right Control Radiator Temperature	float64	0 - 58
rightctrl_temp	Right Control Temperature	float64	-256 - 75
rightmotor_windingtemp	Right Engine Winding Temperature	float64	0 - 211
leftmotor_windingtemp	Left Engine Winding Temperature	float64	0 - 255

Table 3.5: Table of CAN data of category 'Operational Features' and their types with descriptions.

Feature	Description	Data Type	Range or example
acceleration_pedal	Acceleration Pedal	float64	0 - 102
brakepedal_status	Brake Pedal Status	float64	0 - 1
handbrake_status	Handbrake Status	float64	0 - 3
leftmotor_rpm	Left Engine RPM	object	e.g 40
rightmotor_rpm	Right Engine RPM	object	e.g 23.2244444
speed_can	Speed	float64	0 - 67.635
speed_onwheel	Wheel Speed	float64	0 - 88
total_voltage	Total Voltage	float64	-1000 - 652

Table 3.6: Table of CAN data of category 'Topologiacl Features' and their types with descriptions.

Feature	Description	Data Type	Range or example
direction_x	Direction	float64	0 - 359
ew_direction_x	East-West Direction	object	e.g E
latitude_x	Latitude	float64	0 - 23.879
longitude_x	Longitude	float64	0 - 114.017
ns_direction_x	North-South Direction	object	e.g N
total_mileage	Total Mileage	float64	0 - 231242

a category depending on the time of day. The categories were as table 3.14, which are not of equal time intervals since some times in the day are more interesting than others due to the higher traffic flow.

When handling the outliers, where the data ranges were not realistic, the row of data was simply removed if there existed a value out of reasonable range. The duration with outliers was removed by sorting out rows where values below 0 existed. Also, values above 90 were removed since they were few and most probably incorrect. The inside and outside temperatures, as seen in the ranges of the data there, seemed to be values of -273 degrees, which is not feasible in reality, so all temperatures below 10 degrees were removed for both inside and outside temperatures. The temperature should never be below 10 degrees in the month of May. Therefore, these were deemed as unrealistic and faulty data. It is also unrealistic to have values above 40 degrees this month, so they were also removed. There were also singular outliers in run mileage. Hence, values below 40 were removed. Nevertheless, the data still consist of many outliers of the energy consumption. Therefore, some of the specific ones connected to run duration and run mileage were removed. Such as the ones were run duration was above 60 and EC below 7. Also, when run mileage was below 10 and EC above 7, as well as when it was above 20 and EC below 5. The outliers could partially be due to the restarting of the buses, which adds a row of data where the bus goes from fully charged to fully depleted battery without moving. Even though the exact reason for some of the outliers is unknown. Therefore extra caution was taken, and the upper and lower 1 percentiles of EC were removed, leaving

Table 3.7: Table of CAN data of category 'Vehicular Features' and their types with descriptions.

Feature	Description	Data Type	Range or example
busid	Vehicle ID	int64	1 - 500
charge_status	Charging Status	float64	0 - 5
dipped_headlight	Dipped Headlight	float64	0 - 21
door1_status	First Door Status	float64	0 - 6
door2_status	Second Door Status	float64	0 - 7
front_foglamp	Front Fog Lamp	float64	0 - 9
gears	Gear Position	object	e.g 01
headlight	Headlight	float64	0 - 11
left_trunlight	Left Turn Signal	float64	0 - 8
leftmotor_workingmode	Left Engine Working Mode	float64	0 - 10
reversing_light	Reverse Light	float64	0 - 3
right_trunlight	Right Turn Signal	float64	0 - 12
rightmotor_workingmode	Right Engine Working Mode	float64	0 - 1
soc	State of Charge	float64	0 - 100
speaker_status	Speaker Status	float64	0 - 13
stoplight	Tail Light	float64	0 - 4
total_current	Total Current	float64	-637 - 636

Table 3.8: Table of GPS data and their types with descriptions

Feature	**Description**	**Data Type**	**Range or example**
body_valid	Validity	object	e.g true
busid	Vehicle ID	int64	1 - 500
data_serial	Serial Number	float64	0 - 37950
direction	Direction	float64	0 - 360
ew_direction	East-West Direction	object	e.g E
gps_key	Hashkey	object	e.g ws0vckm26
gps_mileage	Mileage	float64	0
gps_time	GPS Time	object	e.g 20210501 031219
gps_type	GPS Type	float64	0 - 1
gps_unit	GPS Unit	object	e.g DM
gps_valid	GPS Valid	float64	0
id	ID	float64	2.105010×10^{15} - 2.105012×10^{15}
latitude	Latitude	float64	0 - 23.90778
longitude	Longitude	float64	0 - 114.0411
ns_direction	North-South Direction	object	e.g \ N
redis_time	Redis Time	object	e.g 20210501 031216
reply_flag	Reply Flag	object	e.g true
speed	Current Speed	float64	0 - 275.81
version	Version	float64	0 - 59

Table 3.9: Table of Departure data and their types with descriptions

Feature	Description	Data Type	Range or example
busid	Vehicle ID	int64	1 - 500
change_info	Change Info	object	e.g 自动结束
depart_interval	Departure Interval	object	e.g 20
direction	Route Direction	float64	0 - 2
from_station_id	Departure Station ID	float64	628 - 10156465
from_station_name	Departure Station Name	object	e.g 科学城（天泰二路）总站
route_id_run	Route ID	float64	8 - 84051
route_name_run	Route Name	object	e.g 496 路
route_sub_id	Sub-Route ID	object	e.g -845943
route_sub_name	Sub-Route Name	object	e.g 科学城（天泰二路）总站 科韵路总站
run_date	Departure Date	object	e.g 2021-05-01
run_duration	Duration of Operation	object	e.g 54
run_mileage	Mileage	float64	0 - 63
running_no	Trip Number	object	e.g 6
service_name	Service Name	object	e.g 全程
service_type	Service Type ID	float64	-121 - 11
to_station_id	Arrival Station ID	float64	628 - 10156465
to_station_name	Arrival Station Name	object	e.g 科韵路总站
triplog_begin_time	Departure Time	object	e.g 2021-05-01 12:14:00
triplog_end_time	Arrival Time	object	e.g 2021-05-01 13:08:42
triplog_id	Departure ID	float64	914740258 - 915214387

Table 3.10: Features and Aggregation Functions for Category External Features

Feature	Aggregation Function
ac_set_temperature	mean
ac_switch	max
ac_workmode	mode
inside_car_temperature	mean
leftctrl_radiatortemp	mean
leftctrl_temp	mean
local_time	last
outside_car_temperature	max
redis_time	last
rightctrl_radiatortemp	mean
rightctrl_temp	mean
rightmotor_windingtemp	median
run_date	last
triplog_begin_time	first
triplog_end_time	last

Table 3.11: Features and Aggregation Functions for Category Operational Features

Feature	Aggregation Function
acceleration_pedal	count non-zero
brakepedal_status	count non-zero
depart_interval	median
handbrake_status	count non-zero
leftmotor_rpm	median
rightmotor_rpm	median
run_duration	sum
speed_can	mean
speed_onwheel	median
total_voltage	mean

Table 3.12: Features and Aggregation Functions for Category Topological Features

Feature	Aggregation Function
direction.1	mean
direction_x	mean
ew_direction_x	mode
latitude_x	last
longitude_x	last
ns_direction_x	mode
run_mileage	sum
total_mileage	max

Table 3.13: Features and Aggregation Functions for Category Vehicular Features

Feature	**Aggregation Function**
busid	mode
busid.1	mode
change_info	last
charge_status	max
dipped_headlight	count non-zero
door1_status	count non-zero
door2_status	count non-zero
ec	sum
front_foglamp	count non-zero
from_station_id	first
from_station_name	first
gears	mode
headlight	count non-zero
left_trunlight	count non-zero
leftmotor_workingmode	mode
reversing_light	count non-zero
right_trunlight	count non-zero
rightmotor_workingmode	mode
route_id_run	last
route_name_run	last
route_sub_id	last
route_sub_name	last
running_no	last
service_name	last
service_type	last
soc	min
speaker_status	count non-zero
stoplight	count non-zero
to_station_id	last
to_station_name	last
total_current	max
triplog_id	mode

Table 3.14: Time Categories

Time	Category
00:00 - 06:00	1
06:00 - 10:00	2
10:00 - 13:00	3
13:00 - 16:00	4
16:00 - 20:00	5
20:00 - 24:00	6

2387 rows of data out of the original 5890 rows.

3.1.3 Feature selection

Now that the data preprocessing is complete, the next step involves determining the features for the model. This selection process is carried out using two key techniques: correlation analysis and Validation Inflation Factor (VIF)

3.1.3.1 Correlation

Correlation analysis was conducted for all features, encompassing both original and newly added ones from previous steps. Only features with numerical values were considered for correlation calculations, while those with an object data type were excluded from this stage onwards. Pandas' correlation function, employing Pearson correlation as the default method, was utilised for these calculations. The threshold for feature inclusion was set at 0.6, meaning only features with correlations exceeding 0.6 or falling below -0.6 were retained. Additionally, features demonstrating a correlation of 1.0 were eliminated, as a perfect correlation is unrealistic. The resulting high correlation features are detailed in Table 3.15 and 3.16.

A heatmap of the correlation for these features are shown in figure 3.2. The heatmap shows the correlations for all the features that have high correlation with at least one other feature. There are therefore many of the correlation that are under the 0.6 threshold in the plot.

3.1.3.2 VIF

Following the correlation analysis, the Variance Inflation Factor (VIF) was computed using the 'statsmodels' library in Python, as detailed in the code provided in A.3. VIF serves to quantify multicollinearity, which extends beyond individual feature correlations to encompass correlations among multiple features. Detecting multicollinearity is crucial, as it ensures that independent features maintain their impact within the model. For instance, consider features like 'door1_status' and 'door2_status', both likely indicating door statuses when a bus stops. Including both in the model could lead to redundancy, where their similarities diminish the

Table 3.15: Feature Correlations of the Features with Correlation Above 0.6, part 1

Feature Correlations		
Feature 1	**Feature 2**	**Correlation**
brakepedal_status	stoplight	0.85287
triplog_begin_time	triplog_end_time	0.99762
leftctrl_radiatortemp	ac_switch	-0.65408
total_current	acceleration_pedal	0.61418
dipped_headlight	triplog_end_time	0.62578
local_time	triplog_begin_time	0.99788
rightctrl_radiatortemp	rightmotor_workingmode	0.63748
ec	duration	-0.76565
rightctrl_radiatortemp	total_voltage	-0.83685
rightmotor_windingtemp	rightctrl_radiatortemp	0.87808
speed	leftmotor_rpm	0.79463
dipped_headlight	redis_time	0.62659
triplog_id	triplog_end_time	0.97964
rightctrl_radiatortemp	leftctrl_temp	0.64690
rightmotor_windingtemp	leftmotor_windingtemp	0.62861
triplog_id	time_cat	0.96348
dipped_headlight	triplog_begin_time	0.62753
soc	local_time	-0.85067
leftmotor_rpm	speed_onwheel	0.81148
redis_time	triplog_id	0.98047
rightmotor_windingtemp	total_voltage	-0.75312
dipped_headlight	local_time	0.62659
redis_time	triplog_end_time	0.99888
direction	route_sub_id	0.69444

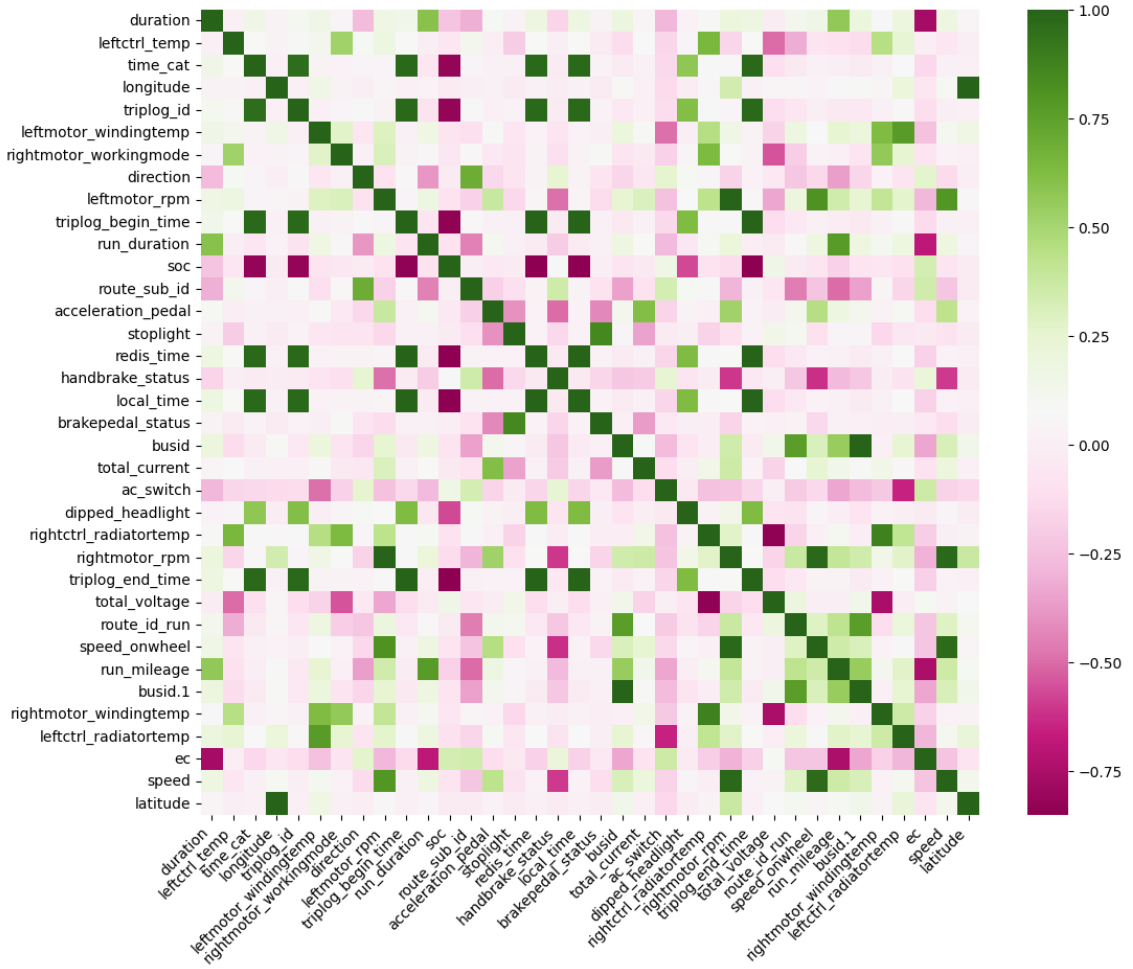


Figure 3.2: Figure of the correlation heatmap for high correlation features

impact of other independent features. A VIF value above 10 suggests high multicollinearity, while values below 1 indicate negligible correlation. Features surpassing this threshold were removed, and VIF was recalculated accordingly. The results of the VIF analysis are presented in tables 3.17, 3.18, 3.19, and 3.20.
Subsequently, features were iteratively eliminated until only those falling within the 1-10 range remained for input into the ML model, as detailed in table 3.21. Notably, certain features deemed essential for model performance were retained, despite their VIF values exceeding the recommended range. These include total voltage, which exhibits a high correlation with total current, inside car temperature (correlated with outside temperature), and SOC (correlated with total voltage). Retaining these features acknowledges their potential significance in contributing to the model's predictive capability despite their elevated VIF values. There were other features that also had a good correlation and multicollinearity in the first iteration but were removed due to the research questions. These were total current and total voltage, acceleration, and brake pedal status. These four features could possibly be important for predicting Energy Consumption. However, they are not data that could be collected before the trip commences and were therefore removed.
The busids are there as a place holder for the data. Total mileage, run duration and

Table 3.16: Feature Correlations of the Features with Correlation Above 0.6, part 2

Feature Correlations		
Feature 1	**Feature 2**	**Correlation**
busid.1	route_id_run	0.76282
leftmotor_rpm	rightmotor_rpm	0.99354
soc	redis_time	-0.85066
soc	triplog_id	-0.82510
soc	triplog_begin_time	-0.84205
longitude	latitude	0.99718
rightmotor_rpm	speed_onwheel	0.97811
local_time	time_cat	0.97628
local_time	redis_time	1.0
triplog_begin_time	time_cat	0.97645
speed	speed_onwheel	0.97070
speed	rightmotor_rpm	0.97975
redis_time	triplog_begin_time	0.99788
redis_time	time_cat	0.97628
dipped_headlight	triplog_id	0.61251
run_duration	run_mileage	0.77318
handbrake_status	speed_onwheel	-0.62150
local_time	triplog_id	0.98047
triplog_id	triplog_begin_time	0.98279
busid	busid.1	1.0
local_time	triplog_end_time	0.99888
leftmotor_windingtemp	leftctrl_radiatortemp	0.77005
soc	triplog_end_time	-0.85113
triplog_end_time	time_cat	0.97535
soc	time_cat	-0.82495
busid	route_id_run	0.76282

Table 3.17: VIF for External Features before filtering.

Feature	**VIF**
ac_set_temperature	1.4969
ac_switch	2.7334
ac_workmode	1.9193
inside_car_temperature	1.7634
leftctrl_radiatortemp	9.0491
leftctrl_temp	3.1647
rightctrl_radiatortemp	23.6156
rightctrl_temp	2.5374
rightmotor_windingtemp	11.8324
outside_car_temperature	2.8208

Table 3.18: VIF for Operational Features before filtering.

Feature	VIF
acceleration_pedal	3.0500
brakepedal_status	4.6627
depart_interval	1.1195
handbrake_status	3.2734
leftmotor_rpm	47.8795
rightmotor_rpm	12.6960
speed_onwheel	34.0187
total_voltage	8.2019

Table 3.19: VIF for Topological Features before filtering.

Feature	VIF
direction.1	1.0542
direction_x	1.3144
run_mileage	6.5567
total_mileage	1.0090

run mileage can all be decided before the trip starts. Speed can be predicted using the speed limits of the roads; even if this is not the exact data used here, it should be for the model to be relatively similar unless the driver was breaking any speed limits. AC switch, Outside and inside temperatures can not be predicted exactly before a trip. However, they can be predicted using weather forecasts. Therefore, these values were kept. These final features used in the VIF are the features to be used in the model. That being; bus id, total mileage, speed, AC switch, outside car temperature, inside car temperature, run duration, run mileage and time categories.

3.2 Building model

The model construction begins with the initial setup using standard parameters and the selected features. Subsequently, parameter optimisation is undertaken to refine the model for improved accuracy. Following optimisation, the model undergoes comparison with two alternative models, namely Random Forest and XGBoost.
The initial model construction utilises the 'Keras' library in Python, as illustrated in the code provided in A.4. This initial framework serves as the foundation upon which subsequent optimisations and comparisons are built.

3.2.1 Model architecture

The model architecture consists of a sequential neural network built using the Keras framework in Python. It comprises several densely connected layers, each performing specific transformations on the input data, see code A.4. This model is then the base for the hyperparameter optimisation.

- **Input Layer:** The input layer is specified with an input dimension matching

Table 3.20: VIF for Vehicular Features before filtering.

Feature	VIF
busid	1801439850948198.5000
busid.1	3002399751580330.5000
charge_status	1.2795
dipped_headlight	3.0914
door1_status	2.0076
door2_status	1.5925
ec	5.5447
front_foglamp	1.6135
from_station_id	2.5159
headlight	1.1122
left_trunlight	1.1020
leftmotor_workingmode	3.1914
reversing_light	1.0942
right_trunlight	1.1419
rightmotor_workingmode	3.3703
route_id_run	5.3650
route_sub_id	3.5739
running_no	3.7725
service_type	1.5736
soc	4.6761
speaker_status	NaN
stoplight	4.6058
to_station_id	2.5563
total_current	1.9116
triplog_id	18.9142

Table 3.21: Features and VIF after removing high VIF features

Feature	VIF
busid	7.8945
total_mileage	1.0082
speed	2.3380
ac_switch	1.9873
outside_car_temperature	100.3056
inside_car_temperature	93.4107
run_duration	14.8886
run_mileage	15.2596
time_cat	9.3011

Table 3.22: Parameters for Grid Search

Parameter	Values
batch_size	10, 20, 50, 100
epochs	10, 50, 100, 125, 150
optimizer	adam, sgd
activation	relu, sigmoid
learning_rate	0.001, 0.01, 0.1
hidden_layers	1, 2, 3
neurons_per_layer	16, 32, 64, 128
regularisation	None, l1, l2

the number of features, which was decided in the previous chapter.

- **Hidden Layers:** The model includes one hidden layer to begin with but more added later, each containing 64 neurons. These layers are densely connected, meaning each neuron in a layer is connected to every neuron in the subsequent layer. Since the model was only decided to be of basic MLPNN structure was to be dense apart from other types. The activation function used in these layers is ReLU as a base to start with before trying other hyperparameters later.
- **Output Layer:** The output layer consists of a single neuron, which produces the model's predictions. This layer has no activation function, which means that it outputs continuous values directly.
- **Compilation:** The model is compiled using the Adam optimiser as a base before trying other hyperparameter later. The loss function used for training is mean squared error (MSE), and also mean absolute error (MAE).

3.2.2 Hyperparameter optimisation

Following the methodology outlined by Yang and Shami [12], hyperparameter optimisation is conducted. Initially, the hyperparameters and their corresponding values are determined, as shown in Table 3.22. The 'GridSearchCV' function from the sklearn library in Python is utilised to determine the optimal hyperparameters from the specified combinations. This process involves training the model with all 8640 combinations and selecting the set of hyperparameters that yield the best results based on Mean Squared Error (MSE) and Mean Absolute Error (MAE). Subsequently, the final model to be employed is determined based on these optimised hyperparameters and the final model's code can be found in A.5.

3.3 Evaluation

For comprehensive model evaluation, a multi-faceted approach encompassing both traditional metrics and advanced techniques was adopted. Mean Absolute Error (MAE) and Mean Squared Error (MSE) were selected as primary metrics due to their widespread use in the literature, facilitating direct comparison with other mod-

els and studies. These metrics offer insights into the model's overall performance and its ability to accurately predict energy consumption.
In addition to MAE and MSE, feature importance analysis played a pivotal role in understanding the relative contributions of different input features in the two comparison model's predictions. Traditional Feature importance plots can not be used on Neural Networks due to the hidden layers.
Furthermore, SHAP (SHapley Additive exPlanations) was used for the MLPNN model instead of the traditional Feature importance used for the other two models. It was used to gain a deeper understanding of the model's behavior, particularly its complex interactions and non-linear relationships. Unlike traditional feature importance methods, SHAP offers a more nuanced perspective by considering the impact of each feature's individual values on the model's output. This systematic approach enabled a comprehensive assessment of feature importance, accounting for both independent features but also their interactions. To complement these analytical approaches, scatter plots were generated to visually explore the relationship between each feature value and the predicted energy consumption. These plots provided intuitive insights into how changes in individual feature values influenced the model's predictions across different scenarios, enhancing interpretability and facilitating actionable insights.
Collectively, these evaluation methods offer a robust framework for assessing the MLPNN model's performance, elucidating its underlying mechanisms, and identifying areas for refinement and improvement.

4
Results and Discussion

Due to the high reliability when assessing the R^2 metric of several models and the possibility of managing the kind of data given, a Multilayered Perceptron Neural Network model is chosen. To evaluate this model, it was to be checked using standard metrics, and these were also compared to two other models, Random forest and Extreme Gradient Boosting.
These two models had Feature Importance implemented to evaluate the different features, they could then be compared to MLPNN by implementing SHAP instead. The time categories and the individual features impact on the energy consumption were also evaluated using comparison plots.

4.1 Final models

The metrics used for a general comparison of the MLPNN model compared to Random Forest and XGB using the same dataset were MSE and MAE. They all used the same features decided by correlation and multicolinearity. The chosen features are as follows:

- Total Mileage
- Speed
- AC Switch
- Outside Car Temperature
- Inside Car Temperature
- Run Mileage
- Run duration
- Time Category
- Bus ID

Details are given below to describe how data is originated from and what impact they could possibly have on the energy consumption.

External features

- **Temperature** (‘outside_car_temperature‘, ’inside_car_temperature’ and ’ac_switch’ from CAN data): Significantly affects HVAC energy consumption and battery efficiency.

- **Time of day**('time_cat' that is calculated based on 'local_time' data): Time of day can correlate with traffic patterns and energy consumption.

Operational features

- **Speed** ('speed' from CAN data): Average and variance in vehicle speed indicate efficiency and energy consumption rates.

Topological features

- **Mileage** ('total_mileage' from CAN data and 'run_mileage from departure data): The total distance the bus has driven since it was constructed can give information on battery depletion due to age. The run mileage can give information on how much the distance for a trip matters for energy consumption.

Vehicular features

- **Bus ID** ('busid' from CAN data): The Bus ID is there as a placeholder to see if a certain bus behaves differently.

4.2 Hyperparameters of MLPNN

The hyperparameters used for the MLPNN model were after the gridsearch as shown in table 4.1 in bold text

Table 4.1: Parameters for Grid Search.

Parameter	Values
batch_size	10, **20**, 50, 100
epochs	10, 50, **100** , 125, 150
optimiser	Adam, **SGD**
activation	**ReLu**, sigmoid
learning_rate	0.001, **0.01**, 0.1
hidden_layers	1, **2**, 3
neurons_per_layer	16 **32**, 64, 128
regularisation	**None**, l1, l2

Batch Size (20): A batch size of 20 is chosen and is the second lowest value of the four chosen for the gridsearch. Smaller batch sizes lead to more frequent updates but may result in noisy gradients, while larger batch sizes provide more stable gradient estimates but require more memory and computational resources. Since the middle value is chosen, one can assume the model has stabilised.

Epochs (100): The model is trained for 100 epochs, allowing it to see the entire training dataset multiple times. This increases the model's ability to learn complex

patterns in the data. However, training for too many epochs may lead to overfitting, where the model memorises the training data instead of learning generalisable patterns. Since this was not the largest or the smallest of the alternatives given to the gridsearch, one can assume that it has not overfitted since it would have then chosen the lesser epoch alternatives.

Optimiser (SGD): Adam optimiser is often chosen for its adaptive learning rate capabilities and efficient convergence. Adam is an extension of SGD. It is reasonable to assume that Adam should yield better results compared to SGD, given its ability to adjust learning rates based on past gradients adaptively. However, since the gridsearch found that SGD was the better fit, one can assume that it is good enough to not need Adams complexity.

Activation Function (ReLU): ReLU is selected as the activation function. It is common due to its simplicity and effectiveness in capturing nonlinear relationships within the data, which works with the type of data in the given dataset. It introduces sparsity in the network, aiding in preventing overfitting by limiting the number of active neurons. Unlike the sigmoid function that was not chosen in the gridsearch the ReLu function only activates a neuron if it has a positive output. Sigmoid never deactivates a neuron it works on a smooth scale. See figure 2.3 from the theory chapter for a more detailed visualisation of the comparison of the two.

Learning Rate (0.01): The learning rate of 0.01, chosen through grid search, is the middle among the alternatives provided. A higher learning rate facilitates faster convergence but may lead to unstable training, while a lower rate ensures more stable updates while requiring longer training times. In this context, the selection of the middle value assures that no further alternatives were needed and the value is stable for the model.

Hidden Layers (2): The gridsearch selected 2 hidden layers for the model. Having more hidden layers enables the model to learn complex hierarchical features, potentially enhancing its predictive capabilities. However, an excessive number of layers may lead to overfitting, where the model performs well on the training data but poorly on unseen data. The selection of two hidden layers suggests a balanced approach, avoiding overfitting.

Neurons per Layer (32): Each hidden layer consists of 32 neurons, determining the representational capacity of the network. Since this is the lowest value and it did not require more, one can assume that it has stabilised.

Regularisation (None): No regularisation technique is applied, implying that the model's objective function does not include additional penalty terms. While regularisation can improve generalisation performance, it also introduces additional hyperparameters that need to be tuned.
It's worth noting that the gridsearch could have included additional values for further optimisation. Since there were no parameters which were at the edge of their

respective ranges it is assumed that it is optimised. However, while the selected values might have been optimal within the provided options, the absence of further choices precludes certainty. Further exploration of a broader range of values could potentially yield even better results.

4.3 Metrics comparison

The metric shows that with the MLPNN model having the chosen features and hyperparameters it is still showing almost the same results as RF and XGB for both metrics MAE and MSE. Where Random Forest is the worst, MLPNN second, and XGB the best. Nevertheless, the results shown in table 4.2 are very similar for all three.

Table 4.2: Comparison of Model Performance

Model	Mean Squared Error	Mean Absolute Error
Random Forest	3.5353	1.3646
XGBoost	3.3483	1.3405
MLPNN	3.4699	1.3528

The variation in model performance between MLPNN (Multilayer Perceptron Neural Network) and ensemble methods like Random Forest and XGBoost can be attributed to several factors, including the sensitivity of MLPNN to outliers. It was expected that the MLPNN model would perform best, but it did not have a margin. One significant factor contributing to the performance gap is the inherent sensitivity of MLPNN models to outliers. MLPNNs are more susceptible to overfitting and tend to capture complex patterns in the data, including outliers, which can adversely affect their generalisation ability. Even though preprocessing steps were undertaken to handle outliers, MLPNNs may still struggle with residual outliers that remain in the data. In contrast, ensemble methods like Random Forest and XGBoost are more robust to outliers due to their inherent averaging or boosting mechanisms. These methods can effectively mitigate the impact of outliers by aggregating predictions from multiple weak learners, thereby reducing the influence of individual outliers on the overall model performance. Additionally, differences in hyperparameters and model architecture between MLPNN and ensemble methods could also contribute to variations in performance. While hyperparameter tuning was performed to optimise model performance, MLPNNs may still be more sensitive to suboptimal hyperparameter settings compared to ensemble methods.
Overall, while the exact reasons for the performance difference cannot be conclusively determined, the sensitivity of MLPNN to outliers is a plausible explanation. Addressing outlier sensitivity and fine-tuning hyperparameters further may help narrow the performance gap between MLPNN and ensemble methods in future iterations.

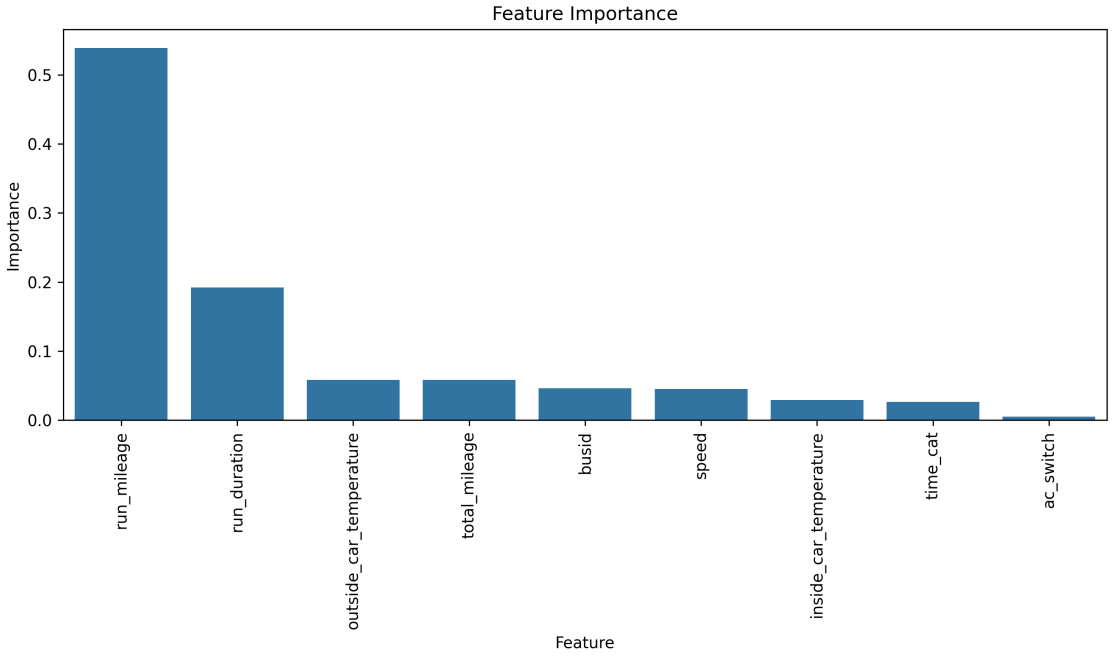


Figure 4.1: Feature importance of the RF model

4.4 Sensitivity analysis

The sensitivity analysis was done using Feature importance for Random Forest and Extreme Gradient Boosting models, and SHapley Additive exPlanations (SHAP) for MLPNN. For all three of the models, the 'Run mileage' has the most impact, with the 'run duration' being the second most important. These two features impact the model to a much further extent than the other features. It is noted again that the 'run mileage' denotes the Euclidean distance between the origin and destination of trips.

4.4.1 Feature Importance

The plots of feature importance for RF and XGB models are shown in figures 4.1 and 4.2. The two most important features of these two models are 'run mileage' and 'run duration'. For Random Forest, the third most important is 'outside temperature,' and for XGB, it is the 'time category'. They do not hold as much significance as the first two features, where 'run mileage' accounts for over 50% of the predicted EC and 'run duration' around 20%.

4.4.2 SHAP

SHAP bee-swarm plot showing how different features correlate to EC as the values of that feature change is shown in figure 4.3. It is to be noted that the thickness of each feature only represents more data of similar values, giving similar results. The range of the plots is to be evaluated and not the thickness.

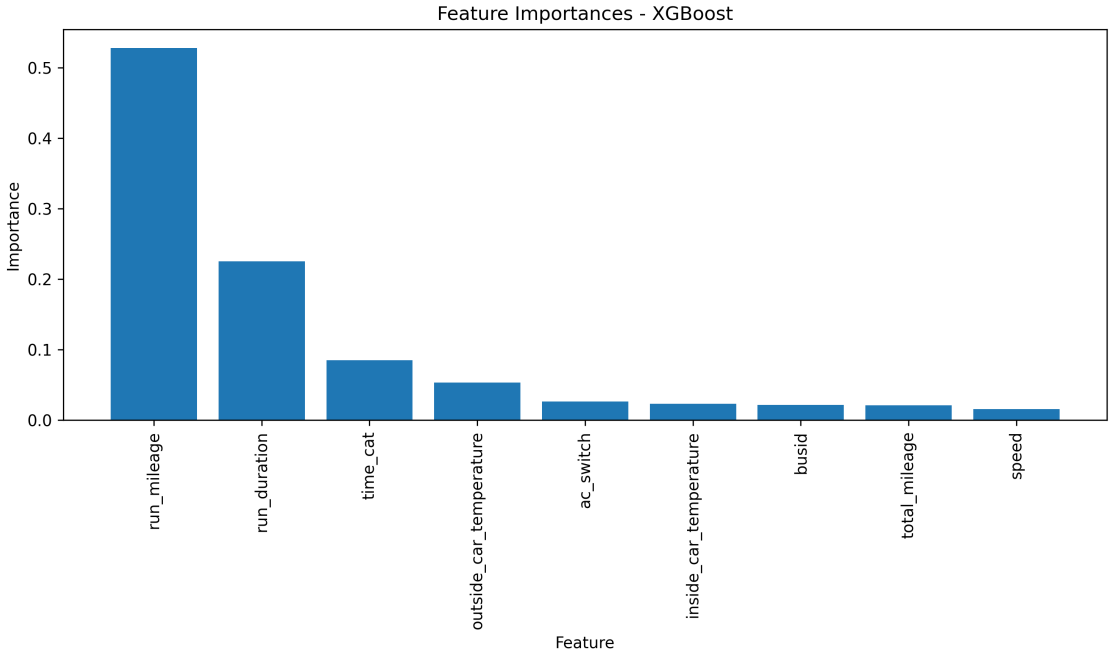


Figure 4.2: Feature importance of the XGBoost model

Run mileage exhibits the most impact on energy consumption. The longer mileage where the bus travels further has higher SHAP values, which indicates a high energy consumption. It also shows that short distances result in negative SHAP, which could be when the bus is charging. This observation is consistent with practical scenarios where a high state of charge typically occurs during bus charging, while consumption is observed when energy is being utilised. This observation aligns well with expectations of reality as well as the results of feature importance for RF and XGB, where run mileage was also the most important feature.
Run duration is the second most significant impact, consistent with the feature importance analysis of RF and XGB models. The majority of values appear to be well-balanced, with longer durations corresponding to higher SHAP values and higher energy consumption. The values are skewed to the right, indicating that high durations have a higher impact than lower. Although the range is not as wide as for run mileage it is larger than what is expected. In feature importance for RF and XGBoost models, the run duration was only about 20%, but here it seems it is closer to 'run mileage' in significance.
The **outside and inside temperatures** seem to complement each other as expected. High outside temperatures have higher SHAP, indicating more energy is being used. Similarly, for 'inside temperature', low inside temperatures used more energy, indicating that the bus is being cooled. Since the data is only collected during one day, one can not conclude how other temperatures would affect the energy consumption. But one would expect there to be a turning point where heating also uses more energy. However, this is not reflected in this plot. Compared to the Feature importance, the results are similar but not exactly the same. Both RF and MLPNN show that outside temperature is the third most important but still not as

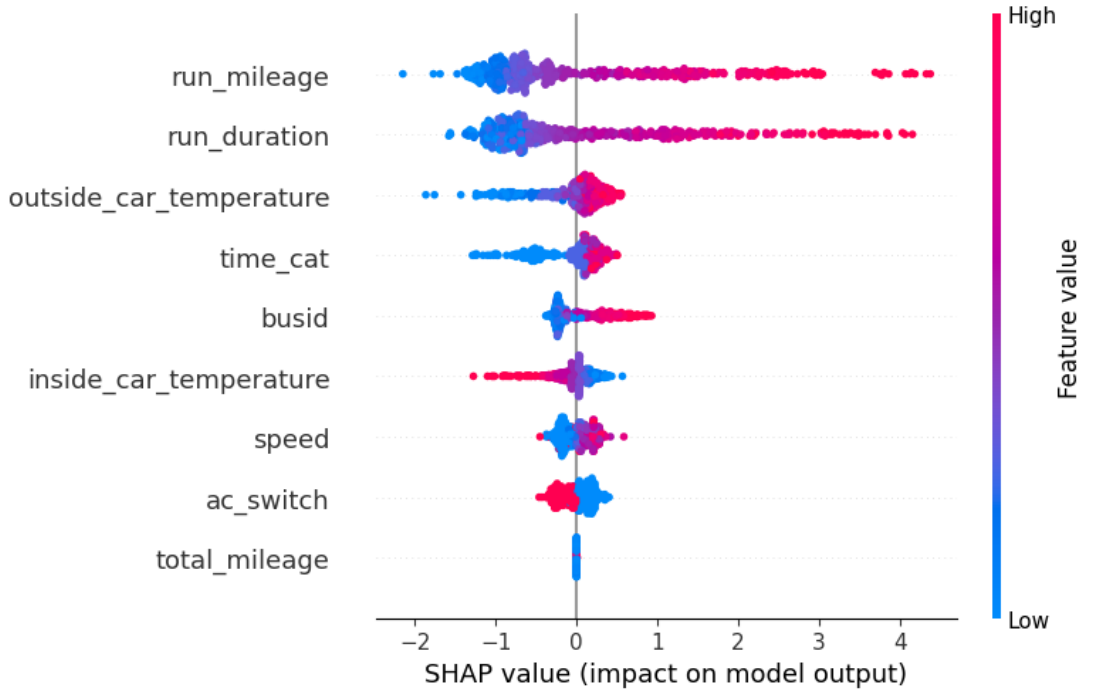


Figure 4.3: SHAP of MLPNN model

important as 'run duration'. Only XGB shows that it is not, but almost the same significance as the 'time category'. 'Inside temperature' is less important for all three models. As stated before, this could be due to the limited data on temperatures.

The **time categories** will undergo further evaluation in subsequent sections of the report. As a reminder, the categories range from the early hours of the day, starting from midnight to later hours until midnight. Upon brief examination of the SHAP values, it appears that the lowest SHAP values correspond to the morning hours, while the highest values are observed in the evening. This suggests that buses are likely charging during the night and operating during the morning and daytime hours. However, it's important to note that this observation requires further investigation and cannot be conclusively determined based solely on this plot. However, it is the fourth most important feature of the MLPNN model. The other two models show varying dependencies on time categories, and SHAP shows a third. It is expected to have an impact, but the unclear significance might suggest that it does not.
Bus ID is a parameter that is only there as a placeholder.
The influence of **speed** on energy consumption is less pronounced than anticipated, looking at the width of the plot. However, there is a discernible correlation between higher speeds and elevated energy usage. It does not show a clear relation between low and high values as the other features; some red values are on both negative and positive SHAP values. This could be due to the speeds only being up to 60 km/h, although additional data from higher speeds would enhance the understanding of their impact.

AC switch only has two input values, on or off, as reflected with only two colours in the plot. However, the output is the opposite of what was expected. When the AC is 1, the SHAP is negative, which would indicate that having the AC uses less energy than when it is off. There could be a simple answer that the value is 1 when the AC is off, which is not the standard procedure. However, since the data was collected on the first of May, the AC might not have been used significantly. Despite this, it is not what is expected and might reflect negatively on the results.
Total mileage also has minimum impact on energy consumption. The plot indicates that within the range provided in the data, total mileage does not significantly affect energy consumption. It could be due to the buses not being of significantly different total mileages to see an impact. But making this conclusion is impossible without more data.

4.5 Temporal comparison of time categories

For each feature, an analysis of the scatter plots will be conducted to observe how its value impacts predicted energy consumption both with and without considering the time categories. Generally the time does not have the impact on any of the features to a greater extent. This could be due to effective route management or specific lanes or priorities given to buses.
Beginning with the **time categories** compared to the predicted energy consumption in figure 4.4. Observing the data, it's notable that there are approximately equal numbers of values across the six categories, with the exception of the first category, which corresponds to around midnight. Interestingly, energy consumption does not show a specific trend of increase or decrease over time, which is favourable. This allows for the evaluation of other features without interference. However, there may be limited information available for the early morning values. There is a slight increase in highest and lowest energy consumption during the middle of the day compared to morning hours. This could suggest that there is no morning rush hour that affects the buses.
First up is the the **run duration** of a trip compared to the energy consumption seen in figure 4.5. Here longer durations required more energy, as expected. One can also see some probable outliers that are not following the rest perfectly. The ones with long durations but still low EC might be that low if the bus was standing still for a longer period during that trip. Looking at the time categories there are some but not completely clear correlations. Most of the trips from 20:00-24:00 are shorter trips, however it can be safe to assume that this is due to planning and not a natural phenomenon. Also between 00:00 and 10:00 the trips appear to be shorter compared to the rest of the day.
Run mileage compared to EC is shown in figure 4.6. A notable correlation emerges between energy consumption (EC) and 'run mileage', with longer distances corresponding to increased energy usage, as anticipated. However, just as with run duration the time category spanning from 6 to 10 o'clock, a considerable number of instances exhibit either minimal energy consumption or deviate from the norm by displaying a skewed distribution towards higher mileages. This suggests that buses

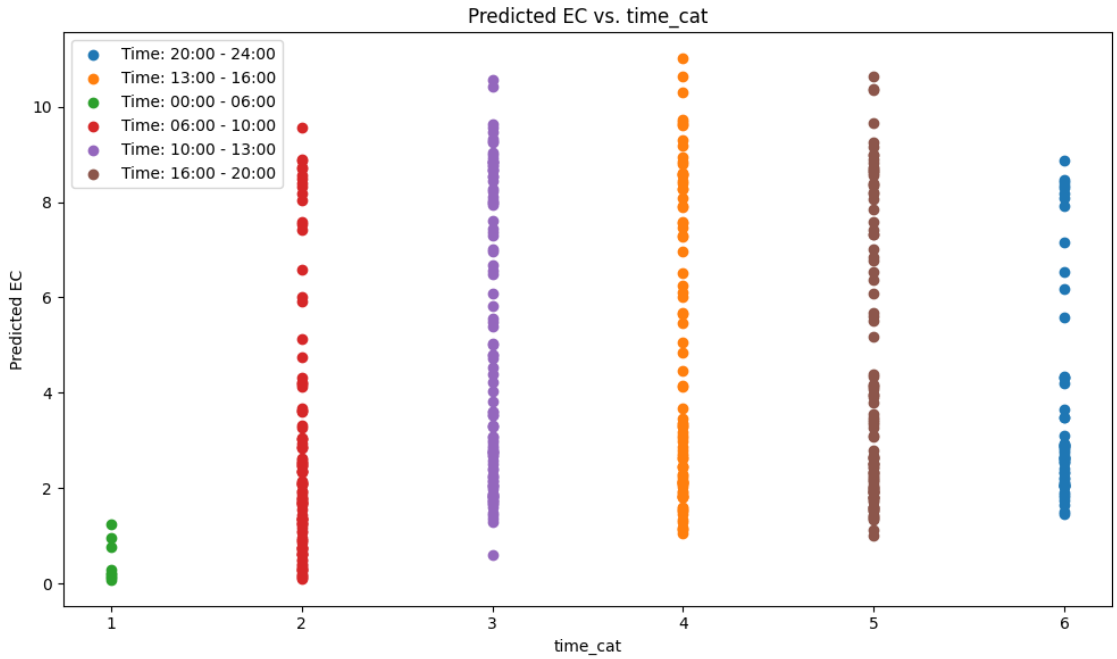


Figure 4.4: comparison between time categories and EC for the different time categories of MLPNN model

cover greater distances during these hours without a proportional increase in energy consumption. Why the energy consumption is so low at these times can not be evaluated but looking at this plot.

The comparison between **inside car temperature** and predicted EC is shown figure 4.7. There doesn't seem to be any prevalent correlation between the two. Additionally, time does not appear to exert any visible impact on this relationship. However there seems to be as suggested in SHAP as well that higher temperatures requires less Energy. as discussed earlier one would hope that there is a switch in the plot were the data shows that cooling also require more energy. From the range of data given this can not be concluded.

Continuing with the analysis of **outside car temperature** and its correlation with EC, illustrated in figure 4.8. Similar to inside car temperature, no significant correlation between these variables is evident. However, time does appear to influence outside temperature with highest consumption being during the middle of the day, which is expected.

In Figure 4.9, the relationship between **speed** and EC is depicted. It indicates no significant correlation between the two. One would expect that higher speed use more energy but according to this data it does not. However, as stated when discussing SHAP plot, a wider range of speed would be needed to discern this. The large amount of values at speed 0 is most likely due to the compressing of data from 1 second interval to tripwise, resulting in the average speed being zero if the bus stops many times during that trip. Furthermore, speed does not demonstrate any dependency on time, which is somewhat surprising, although not as unexpected as the lack of correlation with EC. One would have hoped to see a pattern showing the

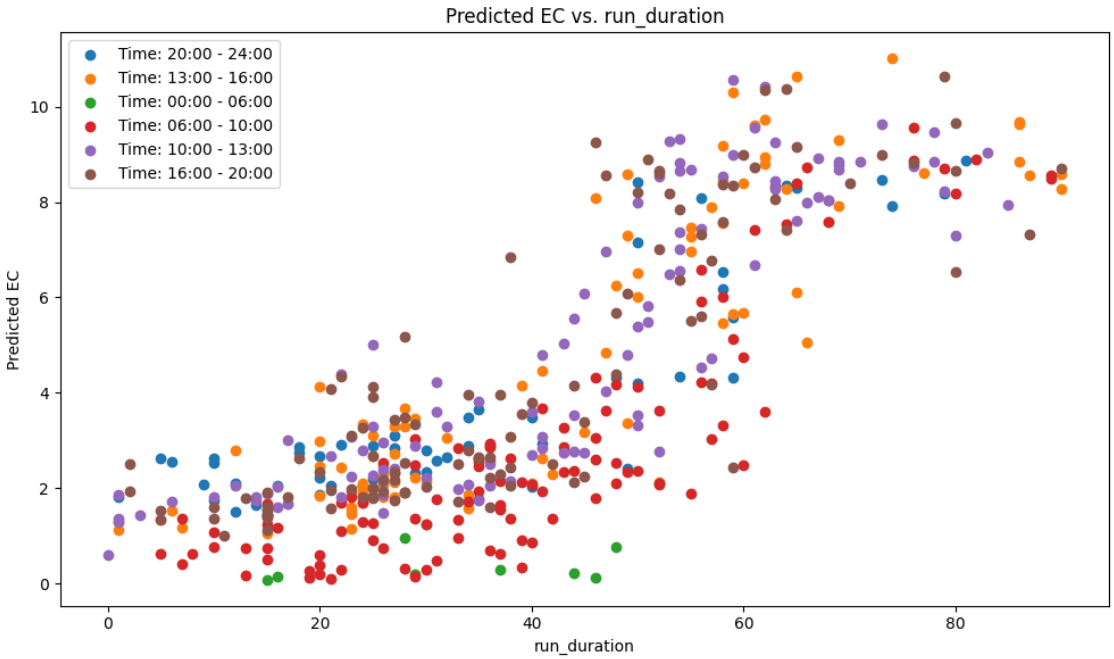


Figure 4.5: comparison between duration and EC for the different time categories of MLPNN model

rush hours before and after work, but it does not. If anything it shows that there is less consumption in the morning, which as discussed earlier is not what is expected. The comparison between **total mileage** and EC is shown in figure 4.10. It does not seem to be any connection between the two features. Not either with the time, which is expected. Despite the few instances, there are more instances of higher energy consumption from buses with greater total mileage. This suggests that buses with more mileage tend to consume more energy over time.
Comparison between **bus ID AC switch** and the predicted energy consumption are not plotted. Since there are only two values for AC switch, either on or off, not much can be evaluated regarding the time. Same goes for bus ID, since it is just a placeholder.

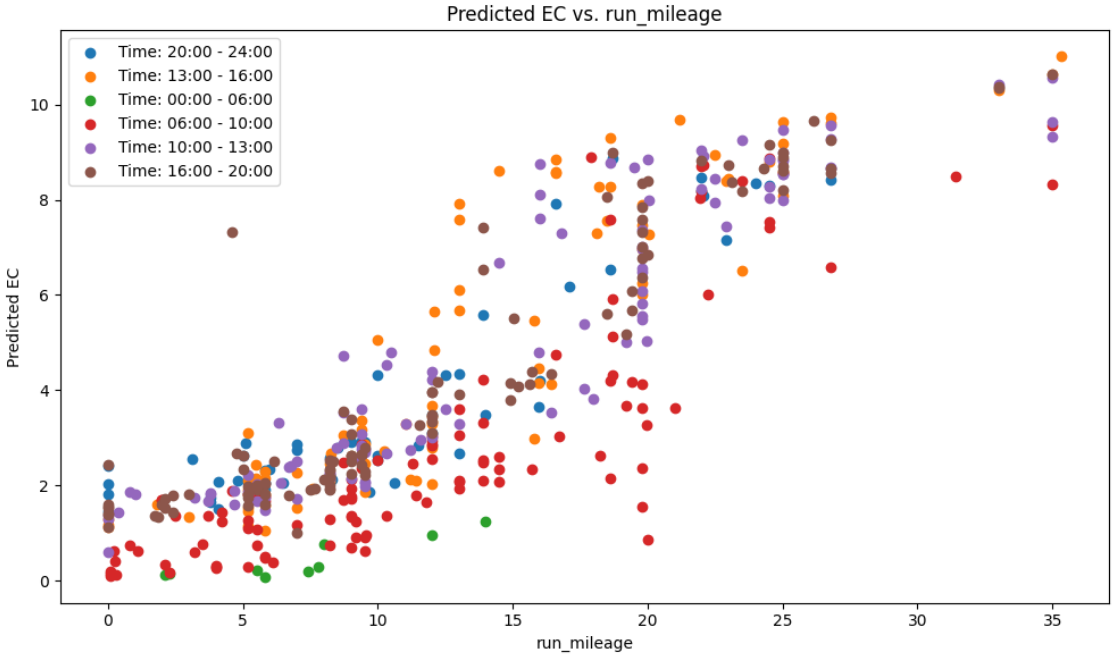


Figure 4.6: comparison between run mileage and EC for the different time categories of MLPNN model

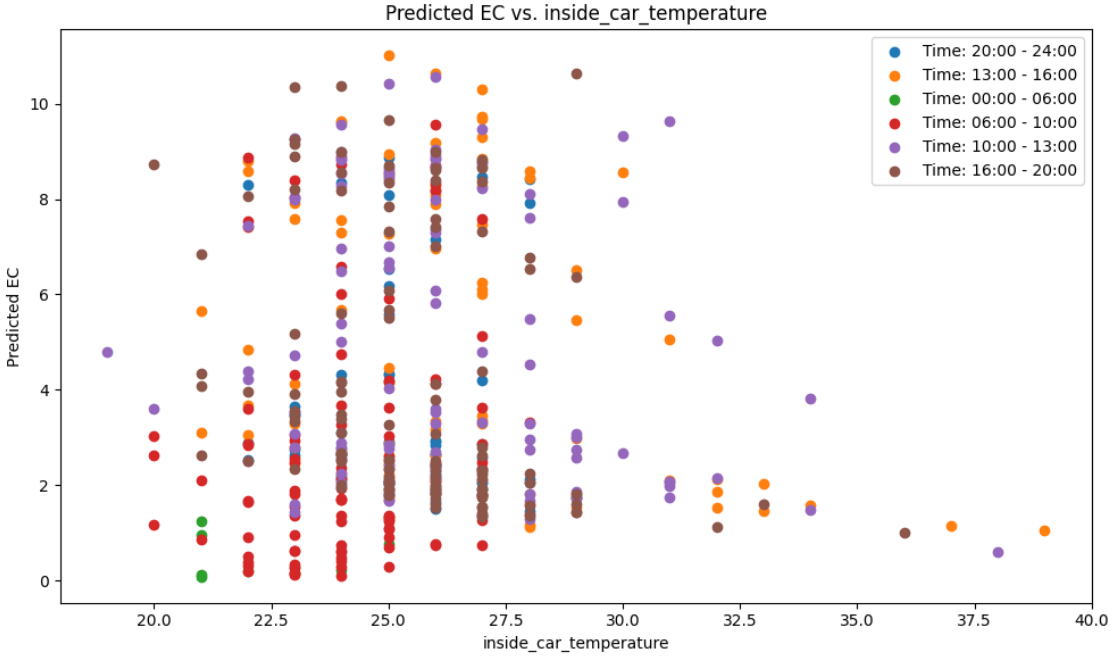


Figure 4.7: comparison between inside car temperature and EC for the different time categories of MLPNN model

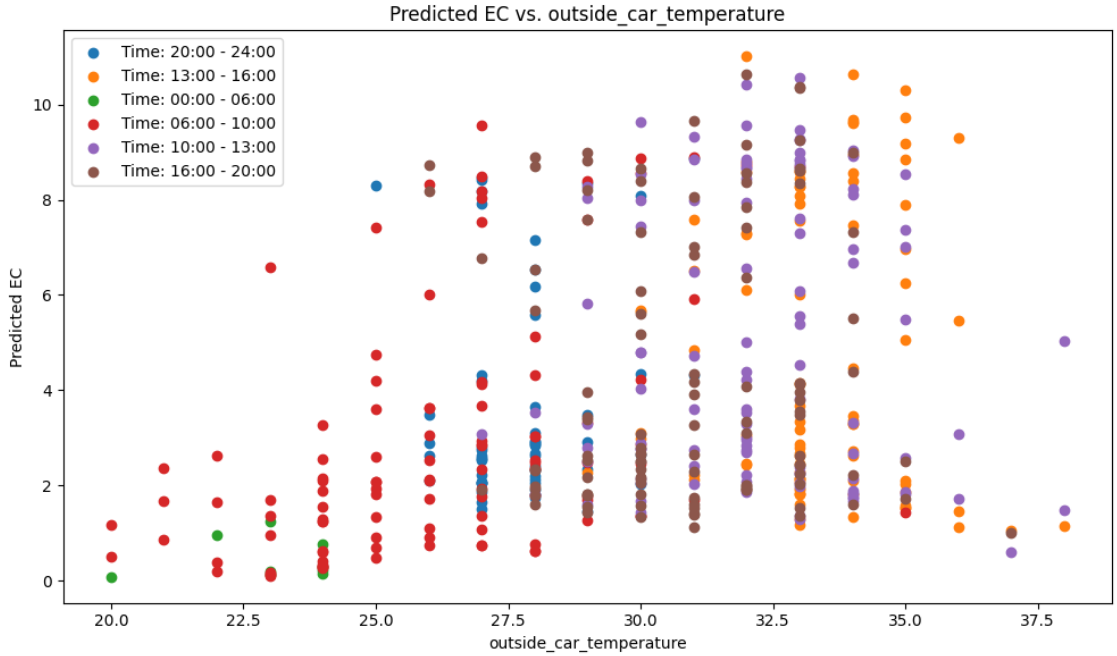


Figure 4.8: comparison between outside car temperature and EC for the different time categories of MLPNN model

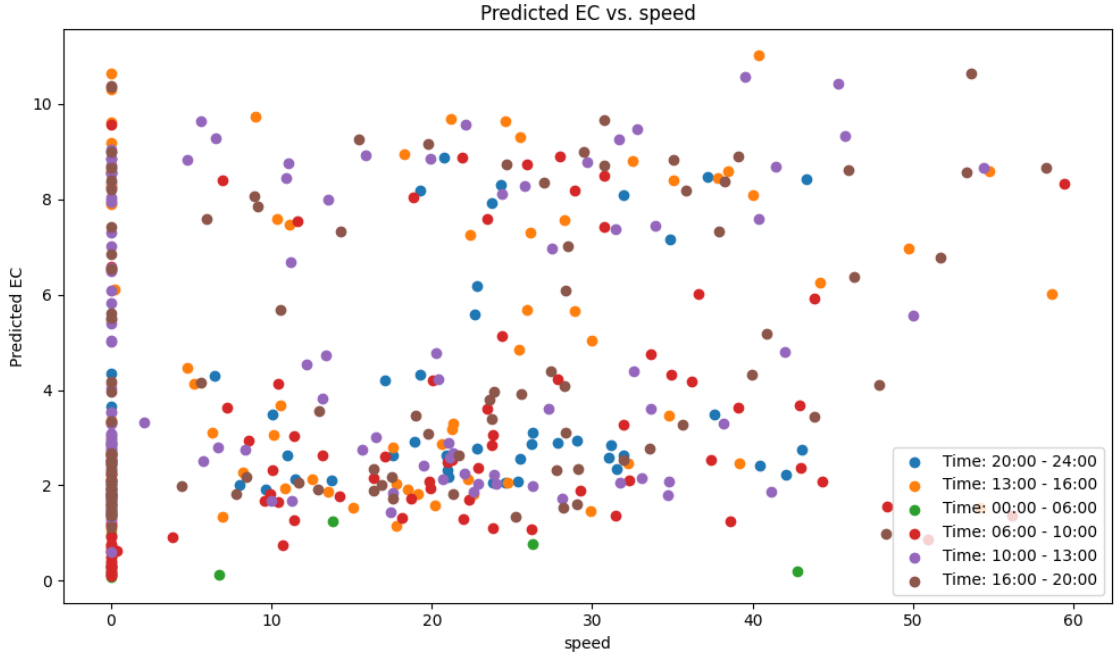


Figure 4.9: comparison between speed and EC for the different time categories of MLPNN model

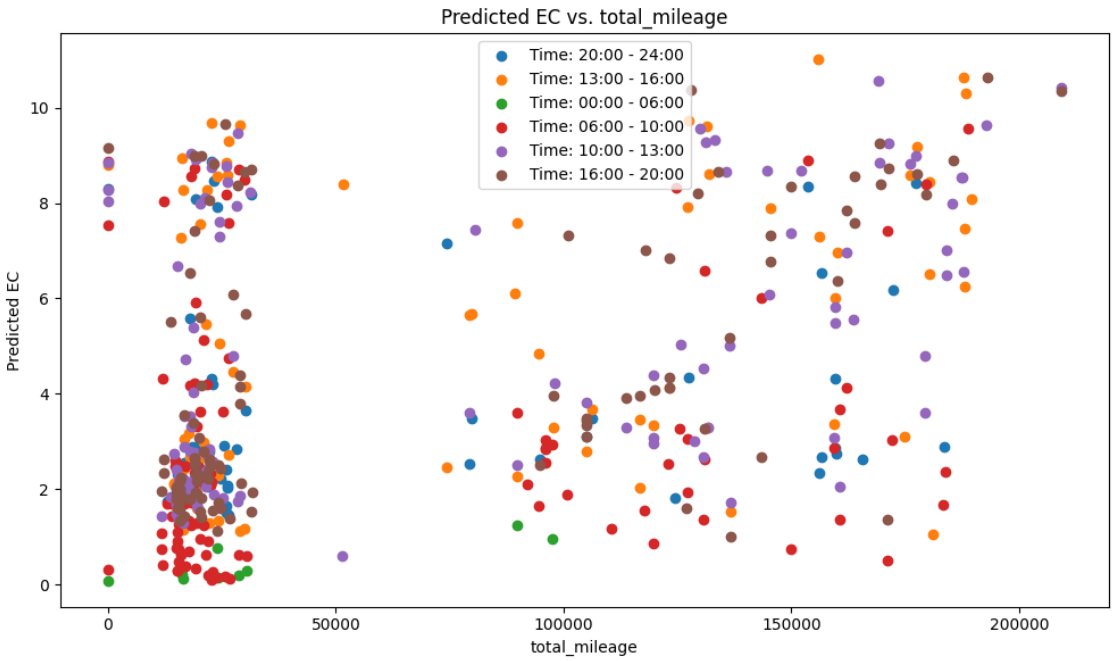


Figure 4.10: comparison between total mileage and EC for the different time categories of MLPNN model

5
Conclusion

From the literature study it was concluded that one of the best models for predicting energy consumption in car was a Multi Layered Perceptron Neural Network. It was decided that this model would work as a base for the study to see if it also would be suitable for electric buses using data from Guangzhou, China. According to the study, the deployed Multi-Layered Perceptron Neural Network (MLPNN) work efficiently if the features are adequate and the size of the data is big enough. Since the data was merged and compressed into tripwise datasets the accuracy of the model might be less. Electric buses energy consumption needs to be predicted before the trips commences. Therefore, charging can be done at the right time and preferably not during the route. For the city planners and other stakeholders this information is also needed to be known to be able to plan where the charging stations should be placed.
The required features for the model can still be debated but the final features are selected using correlation and multi colinearity (VIF). Specificaly, the features in this study include 'run duration', 'run mileage', 'AC switch', 'time category', 'outside temperature', 'inside temperature', 'total mileage', 'speed' and 'bus ID'. All of them should be available to predict or collect data on before a buss starts its route. These features were then added to the model and two other comparison models. Using feature importance and SHAP. The features importance showed that Run mileage had the most impact of around 50% and secondly it was run duration of around 20% for Random Forest (RF) and Extreme Gradient Boosting (XGB) both. The SHAP plot which is more detailed and often used for Neural networks shows similar results and gave some further insights on the different features impact.
Comparing the performance of the MLPNN to RF and XGB models using standard metrics such as Mean Squared Error (MSE) and Mean Absolute Error (MAE), the MLPNN did not perform as well as one expected. Being the second best of the three models. Specifically, the MSE for RF and XGB were 3.5353 and 3.3483, respectively, while the MLPNN had an MSE of 3.4690. Ending up in between the two other models. Although this is a higher value than XGB, it is not much higher. The MAE values for RF and XGB were 1.3646 and 1.3405, respectively, compared to 1.3528 for the MLPNN. Again, while the MLPNN performed worse than XGB, the results are not severe. One would have hoped that the MLPNN model had better results than the other two but that is not the case here. If it can become better than them can also not be concluded. All three models show similar outputs regarding the metrics when using these features and this data.
Time's impact on the model was verified by both feature importance analysis and SHAP (SHapley Additive exPlanations). None of the features showed to be im-

pacted greatly by the time. Although between 00:00 and 10:00 the energy consumption was generally lower, indicating the importance of time-related factors in predicting energy consumption. It is normally expected to see a higher correlation between 'speed' and EC, but with this range of speeds it does not occur. Having time to be less impact full on the features could indicate that the time planing is done efficiently before hand or that there are separate lanes or priorities given to buses.
Accurate predictions of energy consumption are crucial for planners to avoid mid-route charging, which has been successfully managed in Guangzhou, as shown by the time category comparisons. Mid-route charging is not only time-consuming and potentially inconvenient for travelers and commuters but also incurs additional monetary costs by not charging efficiently at planned times. Accurate predictions are also essential for planning the placement of charging stations within the city. Excess charging stations can be unnecessarily costly.
Although the model reveal proper results, the grid search results indicated potential for further hyperparameter tuning. Additionally, more data for some of the features would be preferred, such as speed and temperatures. They can also be directly collected by getting speed limits from the roads or weather forecasts instead of relying on data collected by the buses. Collecting additional data, such as road gradients or comprehensive weather conditions beyond just temperature, can also possibly enhance the model. These improvements can be pursued in future research to develop a more robust and accurate prediction model.

Bibliography

[1] M. Wiatros-Motyka. Global Electricity Review 2023. Retrieved from https://ember-climate.org/insights/research/global-electricity-review-2023/. 2023.

[2] M. Huismans. Tracking Clean Energy Progress 2023. Retrieved from https://www.iea.org/energy-system/electricity/electrification. 2023.

[3] H. Abdelaty and M. Mohamed. "Energy consumption uncertainty model for battery-electric buses in transit". In: 2021 IEEE Transportation Electrification Conference & Expo (ITEC). 2021, pp. 1–5. DOI: 10.1109/ITEC51675.2021.9490103.

[4] H. Abdelaty et al. "Machine learning prediction models for battery-electric bus energy consumption in transit". In: Transportation Research Part D: Transport and Environment 96 (2021), p. 102868. DOI: 10.1016/j.trd.2021.102868.

[5] M. Roy et al. "Reliable energy consumption modeling for an electric vehicle fleet". In: COMPASS '22: Proceedings of the 2022 ACM SIGCAS Conference. 2022, pp. 29–44. DOI: 10.1145/3530190.3534803.

[6] Y. Xing et al. "Operation energy consumption estimation method of electric bus based on CNN time series prediction". In: Mathematical Problems in Engineering 2022 (2022), Article 6904387. DOI: 10.1155/2022/6904387.

[7] Y. E. Ekici et al. "A novel energy consumption prediction model of electric buses using real-time big data from route, environment, and vehicle parameters". In: IEEE Access 11 (2023), pp. 104305–104322. DOI: 10.1109/ACCESS.2023.3316362.

[8] J. Bi et al. "Estimating remaining driving range of battery electric vehicles based on real-world data: A case study of Beijing, China". In: Energy 169 (2019), pp. 833–843. DOI: 10.1016/j.energy.2018.12.061.

[9] W. Achariyaviriya et al. "Estimating energy consumption of battery electric vehicles using vehicle sensor data and machine learning approaches". In: Energies 16.17 (2023), p. 6351. DOI: 10.3390/en16176351.

[10] M. Nielsen. Neural Networks and Deep Learning. Retrieved from http://neuralnetworksanddeeplearning.com/index.html. Self-published, 2019.

[11] H. J. P. Weerts, A. C. Müller, and J. Vanschoren. Importance of tuning hyperparameters of machine learning algorithms. Retrieved from https://arxiv.org/pdf/2007.07588.pdf. 2020.

[12] L. Yang and A. Shami. "On hyperparameter optimization of machine learning algorithms: Theory and practice". In: Neurocomputing 415 (2020), pp. 295–316. DOI: 10.1016/j.neucom.2020.07.061.

A

Appendix

A.1 Merge Files

```
import pandas as pd

# Load the DataFrame from the file
file_path_timetable = xxxtimetable
file_path_can = xxxcan

# Read the data without parsing dates initially
df_timetable = pd.read_csv(file_path_timetable, sep='\t',
    ↪ na_values=[r'\N'])
df_can = pd.read_csv(file_path_can, sep='\t')

# Convert date columns to datetime format with error handling
df_timetable['triplog_begin_time'] =
    ↪ pd.to_datetime(df_timetable['triplog_begin_time'],
    ↪ format='%Y-%m-%d␣%H:%M:%S', errors='coerce')
df_timetable['triplog_end_time'] =
    ↪ pd.to_datetime(df_timetable['triplog_end_time'], format='%Y-%m-%d␣
    ↪ %H:%M:%S', errors='coerce')

df_can['local_time'] = pd.to_datetime(df_can['local_time'],
    ↪ format='%Y%m%d␣%H%M%S', errors='coerce')

# Group rows by 'busid' and 'local_time'
grouped_timetable = df_timetable.groupby(['busid', 'triplog_begin_time',
    ↪ 'triplog_end_time'])

# Create an empty list to store merged data
handled_and_merged_data = []


# Iterate over each group
for group_name, group_data in grouped_timetable:
    # Find matching rows in the timetable DataFrame based on 'busid' and
        ↪ 'local_time'
    matching_rows = df_can[(df_can['busid'] == group_name[0]) &
        ↪ (df_can['local_time'].between(group_name[1], group_name[2]))]
```

```
    # Merge the data if a match is found
    if not matching_rows.empty:
        # Concatenate the columns of both rows horizontally
        merged_row = pd.concat([group_data.iloc[0],
            ↪ matching_rows.iloc[0]], axis=0)

        # Append the merged row to the list
        handled_and_merged_data.append(merged_row)


# Convert the list of merged data to a DataFrame
handled_and_merged_df = pd.concat(handled_and_merged_data, axis=1).T #
    ↪ Concatenate the rows vertically and transpose the DataFrame

# Save the DataFrame to a CSV file
handled_and_merged_df.to_csv('merged_data.csv', index=False)

print("done")
```

A.2 Handle Data

```
import pandas as pd
import matplotlib.pyplot as plt

# Load the merged DataFrame from the file
merged_data_file = 'merged_data.csv'
merged_df = pd.read_csv(merged_data_file, sep=',')

# Calculate the difference in 'soc' column
merged_df['ec'] = merged_df.groupby('busid')['soc'].diff()*(-1)

merged_df['leftmotor_rpm'] = pd.to_numeric(merged_df['leftmotor_rpm'],
    ↪ errors='coerce')
merged_df['rightmotor_rpm'] = pd.to_numeric(merged_df['rightmotor_rpm'],
    ↪ errors='coerce')

# Convert datetime to numerical format (e.g., timestamp)
merged_df['local_time'] =
    ↪ pd.to_datetime(merged_df['local_time']).astype('int64') // 10**9
merged_df['redis_time'] =
    ↪ pd.to_datetime(merged_df['redis_time']).astype('int64') // 10**9
merged_df['triplog_begin_time'] =
    ↪ pd.to_datetime(merged_df['triplog_begin_time']).astype('int64') //
    ↪ 10**9
merged_df['triplog_end_time'] =
    ↪ pd.to_datetime(merged_df['triplog_end_time']).astype('int64') //
    ↪ 10**9
```

```
# Calculate the difference in 'local_time' column
merged_df['duration'] = merged_df.groupby('busid')['local_time'].diff()/60


def mode_or_single_value(x):
    if len(x.unique()) == 1:
        return x.iloc[0]
    else:
        mode_result = x.mode()
        if not mode_result.empty:
            return mode_result.iloc[0]
        else:
            return None # Handle empty result

# Group the DataFrame by 'busid' from routewise(and 'local_time' for
    ↪ tripwise)
grouped = merged_df.groupby(['busid', 'local_time'])

# Define aggregation functions for different columns
aggregation_functions = {
    'busid' : mode_or_single_value, #busid only has one number, can not
        ↪ use mode
    'rightctrl_temp': 'median', # do not want the outliers such as
        ↪ negatives to impact as much
    'rightmotor_windingtemp' : 'median',# do not want the outliers such
        ↪ as negatives to impact as much
    'leftmotor_windingtemp' : 'median',# do not want the outliers such as
        ↪ negatives to impact as much
    'total_current' : 'max',# want the outliers such as negatives to
        ↪ impact as much
    'gears': mode_or_single_value, # Mode function to find the most
        ↪ common value
    'total_mileage' : 'max', #highest value is chosen
    'brakepedal_status' : lambda x: (x != 0).sum(), # Count non-zero
        ↪ values
    'handbrake_status' : lambda x: (x != 0).sum(), # Count non-zero values
    'soc' : 'min', #lowest value after the interval
    'direction' : 'mean', # average
    'leftmotor_workingmode' : mode_or_single_value, # Mode function to
        ↪ find the most common value
    'reversing_light': lambda x: (x != 0).sum(), # Count non-zero values
    'stoplight': lambda x: (x != 0).sum(), # Count non-zero values
    'speed' : 'mean',# want the outliers to impact
    'longitude': 'last', # Retain the last value within the interval
    'charge_status': 'max', # highest value, 0 not charging, 1 charging,
        ↪ 2 fully charged, 3 interuppted charging
    'door1_status': lambda x: (x != 0).sum(), # Count non-zero values
    'dipped_headlight': lambda x: (x != 0).sum(), # Count non-zero values
```

```
'ns_direction': mode_or_single_value, # Mode function to find the
    ↪ most common value
'rightctrl_radiatortemp': 'mean', # average
'total_voltage': 'mean', # average
'door2_status': lambda x: (x != 0).sum(), # Count non-zero values
'latitude': 'last', # Retain the last value within the interval
'leftmotor_rpm': 'median',
'leftctrl_temp': 'mean', # average
'leftctrl_radiatortemp': 'mean', # average
'rightmotor_rpm': 'median',
'left_trunlight': lambda x: (x != 0).sum(), # Count non-zero values
'front_foglamp': lambda x: (x != 0).sum(), # Count non-zero values
'speed_onwheel': 'median',# want the outliers to impact
'ew_direction': mode_or_single_value, # Mode function to find the
    ↪ most common value
'local_time': 'last', # Retain the last value within the interval
'rightmotor_workingmode': mode_or_single_value, # Mode function to
    ↪ find the most common value
'headlight': lambda x: (x != 0).sum(), # Count non-zero values
'right_trunlight': lambda x: (x != 0).sum(), # Count non-zero values
'redis_time': 'last', # Retain the last value within the interval
'ac_switch': lambda x: (x != 0).sum(), # Count non-zero valuess
'speaker_status': lambda x: (x != 0).sum(), # Count non-zero values
'outside_car_temperature': 'max', # max ot the time for trip
'inside_car_temperature': 'mean', # average
'ac_workmode': lambda x: (x != 0).sum(), # Count non-zero values
'ac_set_temperature': 'mean', # average
'acceleration_pedal': lambda x: (x != 0).sum(), # Count non-zero
    ↪ values
'busid.1':mode_or_single_value,
'triplog_id':mode_or_single_value,
'change_info':'last',
'depart_interval':'median',
'direction.1':'mean',
'from_station_id':'first',
'from_station_name':'first',
'to_station_id':'last',
'to_station_name':'last',
'route_id_run':'last',
'route_name_run':'last',
'route_sub_id':'last',
'route_sub_name':'last',
'run_date':'last',
'run_duration':'sum',
'run_mileage':'sum',
'running_no':'last',
'service_type':'last',
'triplog_begin_time':'first',
'triplog_end_time':'last',
```

```
    'service_name':'last',
    'ec':'sum',
    'duration':'sum'

}

# Resample each group to full trip intervals and apply aggregation
    ↪ functions
resampled_data = {}
for group_name, group_data in grouped:
    # Resample and aggregate data for each group ('busid' for routewise
        ↪ and 'triplog_id' for tripwise)
    resampled_group =
        ↪ group_data.groupby('triplog_id').agg(aggregation_functions)
    resampled_data[group_name] = resampled_group


# Concatenate resampled data for all groups
resampled_df = pd.concat(resampled_data.values(),
    ↪ keys=resampled_data.keys())

# add the time category column
# Define the time intervals and corresponding categories
time_intervals = [(0, 6), (6, 10), (10, 13), (13, 16), (16, 20), (20, 24)]
category_labels = [1, 2, 3, 4, 5, 6]

# Convert 'triplog_begin_time' to datetime format
resampled_df['triplog_begin_time'] =
    ↪ pd.to_datetime(resampled_df['triplog_begin_time'], unit='s')

# Extract hour component from 'local_time' and categorize into time
    ↪ intervals
resampled_df['hour'] = resampled_df['triplog_begin_time'].dt.hour
resampled_df['time_cat'] = pd.cut(resampled_df['hour'], bins=[interval[0]
    ↪ for interval in time_intervals] + [24], labels=category_labels,
    ↪ right=False)

# Convert 'time_cat' to integer type
resampled_df['time_cat'] = resampled_df['time_cat'].astype('int64')

# Convert 'triplog_begin_time' back to datetime format
resampled_df['triplog_begin_time'] =
    ↪ pd.to_datetime(resampled_df['triplog_begin_time']).astype('int64')
    ↪ // 10**9

# Drop the temporary 'hour' column, no longer needed
resampled_df.drop('hour', axis=1, inplace=True)
```

```
# Calculate the 1st and 99th percentiles for the 'ec' column
lower_percentile = resampled_df['ec'].quantile(0.01)
upper_percentile = resampled_df['ec'].quantile(0.99)

# Filter out rows with 'ec' values outside the 1st and 99th percentiles
resampled_df_filtered = resampled_df[(resampled_df['ec'] >=
    ↪ lower_percentile) & (resampled_df['ec'] <= upper_percentile)]

# Filter out rows with 'soc' values under 10%
resampled_df_filtered_soc =
    ↪ resampled_df_filtered[resampled_df_filtered['soc'] >= 10]

resampled_df_filtered_temp = resampled_df_filtered_soc[
    (resampled_df_filtered_soc['inside_car_temperature'] >= 10) &
    (resampled_df_filtered_soc['inside_car_temperature'] <= 40) &
    (resampled_df_filtered_soc['outside_car_temperature'] >= 10) &
    (resampled_df_filtered_soc['outside_car_temperature'] <= 40) &
    (resampled_df_filtered_soc['run_duration'] <= 90) &
    (resampled_df_filtered_soc['run_duration'] >= 0) &
    ~((resampled_df_filtered_soc['run_duration'] > 60) &
        ↪ (resampled_df_filtered_soc['ec'] < 7)) &
    ~((resampled_df_filtered_soc['run_mileage'] < 10) &
        ↪ (resampled_df_filtered_soc['ec'] > 7)) &
    ~((resampled_df_filtered_soc['run_mileage'] > 20) &
        ↪ (resampled_df_filtered_soc['ec'] < 5))
]



# Save
resampled_df_filtered_temp.to_csv('handled_data_tripwise.csv',
    ↪ index=False)
```

A.3 Feature Selection

```
#Main
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from statsmodels.stats.outliers_influence import variance_inflation_factor

def create_correlation_heatmap(dataframe,
    ↪ save_path='correlation_heatmap.png'):
    # Convert non-numeric columns to numeric
    for column in dataframe.select_dtypes(include=['object']).columns:
        dataframe[column] = pd.to_numeric(dataframe[column],
            ↪ errors='coerce')
```

```
    # Calculate the correlation matrix
    corr_matrix = dataframe.corr()

    # Set the size of the heatmap
    plt.figure(figsize=(12, 10))

    # Plot the heatmap
    heatmap = sns.heatmap(corr_matrix, annot=False, cmap='PiYG',
        ↪ fmt=".2f")

    # Rotate the x-axis labels for better readability
    heatmap.set_xticklabels(heatmap.get_xticklabels(), rotation=45,
        ↪ ha='right')

    # Save the heatmap as an image
    plt.savefig(save_path, bbox_inches='tight')
    plt.close()

def calculate_high_corr_features_values(df_corr):
    # Select only numeric columns
    numeric_df = df_corr.select_dtypes(include=['number'])

    # Calculate the correlation matrix
    corr_matrix = numeric_df.corr()

    # Set a threshold for correlation (adjust as needed)
    high_corr_features_values = set()

    # Find features with high correlation
    for i in range(len(corr_matrix.columns)):
        for j in range(i+1, len(corr_matrix.columns)):
            colname1 = corr_matrix.columns[i]
            colname2 = corr_matrix.columns[j]
            correlation_value = corr_matrix.iloc[i, j]
            # Adjusted condition to include only correlations within the
                ↪ range of 0.6 to 1
            if (correlation_value >= 0.6 and correlation_value <= 1) or
                ↪ (correlation_value <= -0.6 and correlation_value >= -1)
                ↪ and colname1 != colname2:
                high_corr_features_values.add((colname1, colname2,
                    ↪ correlation_value))

    return corr_matrix, high_corr_features_values



def calculate_vif(df):
    # Fill NaN values with the mean of each column
    df_filled = df.fillna(df.mean())
```

```
    # Initialize a list to store VIF values
    vif_values = []

    # Calculate VIF for each feature
    for i in range(df_filled.shape[1]):
        vif = variance_inflation_factor(df_filled.values, i)
        vif_values.append(vif)

    # Create a DataFrame to store VIF values along with feature names
    vif_df = pd.DataFrame({'Feature': df.columns, 'VIF': vif_values})

    return vif_df

def feature_selection():

    # Load data
    df = pd.read_csv('handled_data_tripwise.csv')

    # Calculate high correlation features
    corr_matrix, high_corr_features_values=
        ↪ calculate_high_corr_features_values(df)
    # Display all correlations
    #print("Correlations:", corr_matrix)

    # Display the features with high correlation
    print("High correlations:", high_corr_features_values)

    # Extract unique individual features with high correlation
    high_corr_features = set()
    for feature_pair in high_corr_features_values:
        high_corr_features.update(feature_pair[:2]) # Extracting only
            ↪ column names from the tuple

    # Filter out the numerical correlation values
    feature_names = [feature for feature in high_corr_features if
        ↪ isinstance(feature, str)]

    # Display the features with high correlation
    print("High correlations:")
    for feature in feature_names:
        print(feature)

    create_correlation_heatmap(df[feature_names])

    # Set display format for VIF DataFrame
    pd.set_option('display.float_format', lambda x: '{:.4f}'.format(x))

    # Calculate VIF
```

```
    df_numeric = df.select_dtypes(include=['float64', 'int64']) # Select
        ↪ only numeric columns
    vif_df = calculate_vif(df_numeric)

    # Print filtered features
    print("Features and VIF before removing high VIF features:", vif_df)

    df_remove = df_numeric.drop(columns=['busid', 'busid.1',
        ↪ 'direction.1', 'ec', 'local_time', 'redis_time',
    'ac_set_temperature', 'triplog_end_time', 'latitude',
        ↪ 'leftctrl_radiatortemp', 'leftmotor_rpm',
        ↪ 'leftmotor_windingtemp', 'rightctrl_temp',
    'handbrake_status', 'stoplight', 'route_sub_id',
        ↪ 'rightmotor_windingtemp', 'leftmotor_workingmode',
        ↪ 'reversing_light', 'charge_status',
    'door1_status', 'dipped_headlight', 'rightctrl_radiatortemp',
        ↪ 'door2_status', 'leftctrl_temp', 'rightmotor_rpm',
        ↪ 'left_trunlight', 'speed_onwheel',
    'rightmotor_workingmode', 'headlight', 'right_trunlight',
        ↪ 'triplog_id', 'from_station_id', 'to_station_id',
        ↪ 'route_id_run', 'running_no',
    'service_type', 'triplog_begin_time', 'speaker_status',
        ↪ 'front_foglamp', 'ac_workmode', 'longitude', 'depart_interval',
        ↪ 'duration', 'direction', 'soc',
      'total_voltage', 'total_current', 'acceleration_pedal',
          ↪ 'brakepedal_status'])

    # Recalculate VIF for the updated DataFrame
    vif_df_filtered = calculate_vif(df_remove)

    # Print filtered features after removal
    print("\nFeatures and VIF after removing high VIF features:",
        ↪ vif_df_filtered)

    # Collect feature names in a list and add single quotes around each
        ↪ feature name
    feature_names_list = ["'" + feature + "'" for feature in
        ↪ vif_df_filtered['Feature'].tolist()]

    # Join feature names into a single string with commas
    features_to_keep_str = ', '.join(feature_names_list)

    # Print the features in the desired format
    print("features_to_keep = [", features_to_keep_str, "]")

# Check if the script is being run directly
if __name__ == "__main__":
    # Call the main function
    feature_selection()
```

A.4 MLPNN model

```
import tensorflow as tf
import pandas as pd
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dense
import keras.backend as K
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from keras.models import load_model
import numpy as np
import seaborn as sns
from sklearn.model_selection import GridSearchCV
from keras.wrappers.scikit_learn import KerasRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error
from keras.regularizers import l1, l2
import pickle
import joblib

def r2_score(y_true, y_pred):
    SS_res = K.sum(K.square(y_true - y_pred))
    SS_tot = K.sum(K.square(y_true - K.mean(y_true)))
    return (1 - SS_res/(SS_tot + K.epsilon()))

def create_model(hidden_layers=1, neurons_per_layer=64,
    ↪ activation='relu', optimizer='adam', learning_rate=0.01,
    ↪ regularization=None):
    model = Sequential()
    model.add(Dense(neurons_per_layer, activation=activation,
        ↪ input_dim=input_dim, kernel_regularizer=regularization))
    for _ in range(hidden_layers - 1):
        model.add(Dense(neurons_per_layer, activation=activation,
            ↪ kernel_regularizer=regularization))
    model.add(Dense(output_dim, activation=None))
    model.compile(optimizer=optimizer,
                loss='mean_squared_error',
                metrics=['mae', 'mse'])
    return model

def MLPNN():
    # Load data
    df = pd.read_csv('handled_data_tripwise.csv')

    features_to_keep = ['busid', 'total_current', 'total_mileage',
        ↪ 'brakepedal_status', 'busid', 'speed', 'total_voltage',
        ↪ 'ac_switch', 'outside_car_temperature',
        ↪ 'inside_car_temperature', 'acceleration_pedal', 'run_mileage',
```

```
    ↪ 'duration', 'time_cat' ]

# Filter DataFrame to keep only desired features
X = df[features_to_keep]
y = df['ec']
# Replace NaN values with zero
X = X.fillna(0)
y = y.fillna(0)

global input_dim, output_dim
input_dim = X.shape[1]
output_dim = 1

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
    ↪ test_size=0.2, random_state=42)

# Instantiate the MinMaxScaler
scaler = MinMaxScaler()

# Fit the scaler to training data
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Create KerasRegressor object
model = KerasRegressor(build_fn=create_model, verbose=0)

# Define the parameters for grid search
param_grid = {
    'batch_size': [10, 20, 50, 100],
    'epochs': [10, 50, 100, 125, 150],
    'optimizer': ['adam', 'sgd'],
    'activation': ['relu', 'sigmoid'],
    'learning_rate': [0.001, 0.01, 0.1],
    'hidden_layers': [1, 2, 3],
    'neurons_per_layer': [16, 32, 64, 128],
    'regularization': [None, 'l1', 'l2'],
}

# Instantiate GridSearchCV
grid_search = GridSearchCV(estimator=model, param_grid=param_grid,
    ↪ cv=3, verbose=2)

# Fit GridSearchCV to training data
grid_search.fit(X_train_scaled, y_train)

# Retrieve best model and parameters
best_model = grid_search.best_estimator_
best_params = grid_search.best_params_
```

```
    print("Best Parameters:", best_params)

    # Predict values using the best model
    y_pred = best_model.predict(X_test_scaled)

    #Evaluate the Model
    mse = mean_squared_error(y_test, y_pred)
    print(f'Mean Squared Error: {mse}')
    rmse = np.sqrt(mse) # Calculate RMSE
    print(f'Root Mean Squared Error: {rmse}')
    mae = mean_absolute_error(y_test, y_pred) # Calculate RMSE
    print(f'Mean Absolute Error: {mae}')

    # Save the best model and relevant data
    save_model_and_data(best_model, X_train_scaled, X_test_scaled,
        ↪ y_train, y_test, best_params)

    return best_model, features_to_keep

def save_model_and_data(model, X_train_scaled, X_test_scaled, y_train,
    ↪ y_test, best_params=None, features_to_keep=None):
    # Save the trained model
    with open('mlpnn_model.pkl', 'wb') as f:
        pickle.dump(model, f)

    # Save input features and target variables
    joblib.dump(X_train_scaled, 'X_train_scaled.pkl')
    joblib.dump(X_test_scaled, 'X_test_scaled.pkl')
    joblib.dump(y_train, 'y_train.pkl')
    joblib.dump(y_test, 'y_test.pkl')

    # Optionally, save best parameters
    if best_params:
        with open('best_params.pkl', 'wb') as f:
            pickle.dump(best_params, f)

    # Optionally, save features_to_keep
    if features_to_keep:
        with open('features_to_keep.pkl', 'wb') as f:
            pickle.dump(features_to_keep, f)

# Call the MLPNN function
best_model = MLPNN()
```

A.5 MLPNN best model

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import Dense
from sklearn.metrics import mean_squared_error, mean_absolute_error
import matplotlib.pyplot as plt
import seaborn as sns
import shap
from pdpbox import pdp
from keras.optimizers import Adam
from keras.optimizers import SGD

def plot_feature_importance(model, features, save_path=None):
    # Retrieve the weights of the first layer
    first_layer_weights = model.layers[0].get_weights()[0]

    # Calculate total sum of weights
    total_sum = np.abs(first_layer_weights).sum()

    # Calculate feature importance as a percentage of the total sum of
        ↪ weights
    feature_importance = (np.abs(first_layer_weights).sum(axis=1) /
        ↪ total_sum) * 100

    # Create a DataFrame to store feature names and their importance
    importance_df = pd.DataFrame({'Feature': features, 'Importance':
        ↪ feature_importance})

    # Sort the DataFrame by importance
    importance_df = importance_df.sort_values(by='Importance',
        ↪ ascending=False)

    # Plot the feature importance
    plt.figure(figsize=(10, 6))
    sns.barplot(x='Feature', y='Importance', data=importance_df)
    plt.title('Feature Importance')
    plt.xlabel('Feature')
    plt.ylabel('Importance (%)')
    plt.xticks(rotation=90) # Rotate x-axis labels by 45 degrees
    plt.tight_layout() # Adjust layout to prevent overlapping
    # Save the plot if save_path is provided
    if save_path:
        plt.savefig(save_path)

    plt.show()
```

```
def plot_losses(history, title, save_path=None):
    plt.plot(history.history['loss'], label='Training Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title(title)
    plt.legend()
    # Save the plot if save_path is provided
    if save_path:
        plt.savefig(save_path)

    plt.show()

def plot_metrics(history, metric, title, save_path=None):
    plt.plot(history.history[metric], label='Training ' + metric)
    plt.plot(history.history['val_' + metric], label='Validation ' +
        ↪ metric)
    plt.xlabel('Epochs')
    plt.ylabel(metric)
    plt.title(title)
    plt.legend()
    # Save the plot if save_path is provided
    if save_path:
        plt.savefig(save_path)

    plt.show()

def MLPNN(df, learning_rate=0.01): # Added learning_rate as a parameter
    def create_model(input_dim, output_dim):
        model = Sequential()
        model.add(Dense(32, activation='relu', input_dim=input_dim))
        model.add(Dense(32, activation='relu'))
        model.add(Dense(32, activation='relu'))
        model.add(Dense(output_dim, activation=None))
        optimizer = SGD(learning_rate=learning_rate) # Define optimizer
            ↪ with learning rate
        model.compile(optimizer=optimizer, # Assign optimizer here
                      loss='mean_squared_error',
                      metrics=['mae', 'mse'])
        return model

    # Load data
    df = pd.read_csv('handled_data_tripwise.csv', sep=',')

    features_to_keep = ['total_mileage', 'speed', 'ac_switch',
        ↪ 'outside_car_temperature', 'inside_car_temperature',
        ↪ 'run_mileage', 'run_duration', 'time_cat' , 'busid']
```

```
    # Filter DataFrame to keep only desired features
    X = df[features_to_keep]
    y = df['ec']
    # Replace NaN values with zero
    X = X.fillna(0)
    y = y.fillna(0)

    input_dim = X.shape[1]
    output_dim = 1

    # Split the data into training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(X, y,
        ↪ test_size=0.2, random_state=42)

    # Instantiate the MinMaxScaler
    scaler = MinMaxScaler()

    # Fit the scaler to training data
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # Create KerasRegressor object
    model = create_model(input_dim, output_dim)

    # Train the model
    history = model.fit(X_train_scaled, y_train, epochs=100,
        ↪ batch_size=20, validation_data=(X_test_scaled, y_test),
        ↪ verbose=2)

    # Predict values using the model
    y_pred = model.predict(X_test_scaled)

    # Evaluate the Model
    mse = mean_squared_error(y_test, y_pred)
    print(f'Mean Squared Error: {mse}')
    rmse = np.sqrt(mse)
    print(f'Root Mean Squared Error: {rmse}')
    mae = mean_absolute_error(y_test, y_pred)
    print(f'Mean Absolute Error: {mae}')

    return model, features_to_keep, history, X_train_scaled,
        ↪ X_test_scaled, X_test, y_test, y_pred

# Load data
df = pd.read_csv('handled_data_tripwise.csv')

# Call the MLPNN function
best_MLPNN_model, features_to_keep, history, X_train_scaled,
    ↪ X_test_scaled, X_test, y_test, y_pred = MLPNN(df)
```

```
# Plot the learning curves
plot_losses(history, 'Training and Validation Loss',
    ↪ save_path='TrainingValidationLossMLPNN.png')
plot_metrics(history, 'mse', 'Mean Squared Error',
    ↪ save_path='MeanSquaredErrorMLPNN.png')
plot_metrics(history, 'mae', 'Mean Absolute Error',
    ↪ save_path='MeanAbsoluteErrorMLPNN.png')

# Plot feature importance
plot_feature_importance(best_MLPNN_model, features_to_keep,
    ↪ save_path='FeatureImportanceMLPNN.png')

# Create the SHAP explainer object
explainer = shap.Explainer(best_MLPNN_model, X_train_scaled)

# Calculate SHAP values
shap_values = explainer(X_test_scaled)

# Plot the SHAP summary plot
shap.summary_plot(shap_values, X_test_scaled,
    ↪ feature_names=features_to_keep, show=False)

# Save the plot as SHAPmlpnn.png
plt.savefig('SHAPmlpnn.png')

# Show the plot
plt.show()

# Define the mapping of time categories to actual times
time_category_mapping = {
    1: '00:00 - 06:00',
    2: '06:00 - 10:00',
    3: '10:00 - 13:00',
    4: '13:00 - 16:00',
    5: '16:00 - 20:00',
    6: '20:00 - 24:00'
}

# Get unique time categories from the test data
time_categories = X_test['time_cat'].unique()

# For each subset, calculate the predicted EC values using your MLPNN
    ↪ model
predicted_ec_values = []
for time_category in time_categories:
    # Filter data for the current time category
    indices = X_test['time_cat'] == time_category
    X_filtered = X_test_scaled[indices]
```

```
    # Predict EC values using the MLPNN model
    y_pred_filtered = best_MLPNN_model.predict(X_filtered)

    # Append predicted EC values to the list
    predicted_ec_values.append(y_pred_filtered)

# Plot the relationship between each feature and the predicted EC value
    ↪ for each time category
for feature in features_to_keep:
    plt.figure(figsize=(10, 6))
    for j, time_category in enumerate(time_categories):
        indices = X_test['time_cat'] == time_category
        plt.scatter(X_test[indices][feature], predicted_ec_values[j],
            ↪ label=f'Time: {time_category_mapping[time_category]}')
    plt.title(f'Predicted EC vs. {feature}')
    plt.xlabel(feature)
    plt.ylabel('Predicted EC')
    plt.legend()
    plt.tight_layout()
    plt.savefig(f'{feature}_vs_predicted_EC.png') # Save the plot as an
        ↪ image file
    plt.show()
```

A.6 RF model

```
import numpy as np
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, mean_absolute_error
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Load data
df = pd.read_csv('handled_data_tripwise.csv')
df = df.fillna(0)
df = df.reset_index(drop=True)

# Define features to keep
features_to_keep = [ 'busid', 'total_mileage', 'speed', 'ac_switch',
    ↪ 'outside_car_temperature', 'inside_car_temperature',
    ↪ 'run_mileage', 'run_duration', 'time_cat' ]

# Extract features (X) and target variable (y)
X = df[features_to_keep].values
y = df['ec'].values
```

```
# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    ↪ random_state=42)

# Train the Random Forest Model
n_estimators = 100
rf_regressor = RandomForestRegressor(n_estimators=n_estimators,
    ↪ random_state=42)
rf_regressor.fit(X_train, y_train)

# Make Predictions
y_pred = rf_regressor.predict(X_test)

# Evaluate the Model
mse = mean_squared_error(y_test, y_pred)
print(f'Mean Squared Error: {mse}')
mae = mean_absolute_error(y_test, y_pred)
print(f'Mean Absolute Error: {mae}')

# Plot feature importance
def plot_feature_importance(model, features, save_path=None):
    feature_importance = model.feature_importances_
    importance_df = pd.DataFrame({'Feature': features, 'Importance':
        ↪ feature_importance})
    importance_df = importance_df.sort_values(by='Importance',
        ↪ ascending=False)

    plt.figure(figsize=(10, 6))
    sns.barplot(x='Feature', y='Importance', data=importance_df)
    plt.title('Feature Importance')
    plt.xlabel('Feature')
    plt.ylabel('Importance')
    plt.xticks(rotation=90)
    plt.tight_layout()
    if save_path:
        plt.savefig(save_path, dpi=300)
    plt.show()

# Plot feature importance
plot_feature_importance(rf_regressor, features_to_keep,
    ↪ save_path='FeatureImportanceRF.png')

# Plot metrics
def plot_metrics(y_test, y_pred, save_path=None):
    plt.figure(figsize=(10, 6))
    xx = np.arange(0, len(y_test), 1)
    plt.plot(xx[1:100], y_test[1:100], 'r--', linewidth=1, label='Test')
    plt.plot(xx[1:100], y_pred[1:100], 'b', linewidth=1,
        ↪ label='Prediction')
```

```
    plt.legend(fontsize=8)
    plt.grid()
    plt.xlabel("Sample")
    plt.ylabel("Energy (kWh)")
    plt.title("Comparison of Prediction and Test")
    plt.tight_layout()
    if save_path:
        plt.savefig(save_path, dpi=300)
    plt.show()

# Plot metrics
plot_metrics(y_test, y_pred, save_path='MetricsRF.png')
```

A.7 XGBoost model

```
import numpy as np
import xgboost as xgb
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, mean_absolute_error
import matplotlib.pyplot as plt
import pandas as pd

# Load data
df = pd.read_csv('handled_data_tripwise.csv')
df = df.fillna(0)
df = df.reset_index(drop=True)

# Define features to keep
features_to_keep = [ 'busid', 'total_mileage', 'speed', 'ac_switch',
    ↪ 'outside_car_temperature', 'inside_car_temperature',
    ↪ 'run_mileage', 'run_duration', 'time_cat' ]

# Extract features (X) and target variable (y)
X = df[features_to_keep].values
y = df['ec'].values

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    ↪ random_state=42)

# Create and train the XGBoost Model
params = {
    'objective': 'reg:squarederror',
    'max_depth': 3,
    'learning_rate': 0.1,
    'eval_metric': 'rmse'
}
```

```
num_rounds = 100
xgb_regressor = xgb.train(params, xgb.DMatrix(X_train, label=y_train),
    ↪ num_rounds)

# Make Predictions
y_pred = xgb_regressor.predict(xgb.DMatrix(X_test))

# Evaluate the Model
mse = mean_squared_error(y_test, y_pred)
print(f'Mean Squared Error: {mse}')
mae = mean_absolute_error(y_test, y_pred)
print(f'Mean Absolute Error: {mae}')

# Get feature names
feature_names = df[features_to_keep].columns.tolist()

# Plot feature importances
def plot_feature_importance(model, features, save_path=None):
    feature_importance = model.get_score(importance_type='gain')
    total_importance = sum(feature_importance.values())
    coefficients = {feature: importance / total_importance for feature,
        ↪ importance in feature_importance.items()}

    sorted_importance = sorted(coefficients.items(), key=lambda x: x[1],
        ↪ reverse=True)

    plt.figure(figsize=(10, 6))
    plt.bar(range(len(sorted_importance)), [importance for feature,
        ↪ importance in sorted_importance], align="center")
    plt.xticks(range(len(sorted_importance)), [features[int(feature[1:])]
        ↪ for feature, importance in sorted_importance], rotation=90)
    plt.xlabel("Feature")
    plt.ylabel("Importance")
    plt.title("Feature Importances - XGBoost")
    plt.tight_layout()
    if save_path:
        plt.savefig(save_path, dpi=300)
    plt.show()

# Plot feature importance
plot_feature_importance(xgb_regressor, feature_names,
    ↪ save_path='FeatureImportanceXGB.png')

# Plot metrics
def plot_metrics(y_test, y_pred, save_path=None):
    plt.figure(figsize=(10, 6))
    xx = np.arange(0, len(y_test), 1)
    plt.plot(xx[1:100], y_test[1:100], 'r--', linewidth=1, label='Test')
```

```
    plt.plot(xx[1:100], y_pred[1:100], 'b', linewidth=1,
        ↪ label='Prediction')
    plt.legend(fontsize=8)
    plt.grid()
    plt.xlabel("Sample")
    plt.ylabel("Energy (kWh)")
    plt.title("Comparison of Prediction and Test")
    plt.tight_layout()
    if save_path:
        plt.savefig(save_path, dpi=300)
    plt.show()

# Plot metrics
plot_metrics(y_test, y_pred, save_path='MetricsXGB.png')
```

DEPARTMENT OF ARCHITECTURE AND CIVIL ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY