Embedded Machine Learning Application to EV Charging Demand Prediction and Control

End-semester project report submitted in partial fulfilment of the

requirements for the award of the degree

*Of*

# BACHELOR OF TECHNOLOGY (HONS.)

*In*

# ELECTRICAL ENGINEERING

*By*

***ADITYA NARAYAN SINGH (2021GEE001)***

***NAVIN KUMAR (2021UGEE014)***

***ABHISHEK RANJAN (2021UGEE017)***

***PRATIM CHATTERJEE (2021GEE018)***

***GAURAV BARA (2021UGEE020)***

***HARSH SAGAR (2021UGEE044)***

Under the esteemed guidance and supervision of

***Dr. KETHAVATH RAGHAVENDRA NAIK***

Assistant Professor

(dept. of Electrical Engineering)



**DEPARTMENT OF ELECTRICAL ENGINEERING**

**NATIONAL INSTITUTE OF TECHNOLOGY, JAMSHEDPUR**

**APRIL, 2025**

**NATIONAL INSTITUTE OF TECHNOLOGY, JAMSHEDPUR**

**CANDIDATE DECLARATION**

We hereby declare that the work submitted in the project entitled “***Embedded Machine Learning Application to EV Charging Demand Prediction and Control***” in partial fulfilment of the requirement for the award of degree of Bachelor of Technology (Hons.) in the Department of Electrical Engineering, National Institute of Technology, Jamshedpur has been carried out under the supervision and guidance of Dr. Kethavath Raghavendra Naik (Assistant Professor), Department of Electrical Engineering,N.I.T. Jamshedpur.

The matter presented in this work has not been submitted by us for the award of any other degree to this or any other University.

**ADITYA NARAYAN SINGH (2021UGEE001)**

**NAVIN KUMAR (2021UGEE014)**

**ABHISHEK RANJAN (2021UGEE017)**

**PRATIM CHATTERJEE (2021UGEE018)**

**GAURAV BARA (2021UGEE020)**

**HARSH SAGAR (2021UGEE044)**

This is to certify that the above statement made by the candidate is true and correct to the best of my knowledge and belief.

**Dr. KETHAVATH RAGHAVENDRA NAIK**

**Supervisor**

Date: 20/04/2025

**NATIONAL INSTITUTE OF TECHNOLOGY, JAMSHEDPUR - 831014**

**CERTIFICATE**

This is to be certified that the major project work entitled “Embedded Machine Learning Application to EV Charging Demand Prediction and Control” submitted by

**Aditya Narayan Singh (2021UGEE001)**

**Navin Kumar (2021UGEE014)**

**Abhishek Ranjan (2021UGEE017)**

**Pratim Chatterjee (2021UGEE018)**

**Gaurav Bara (2021UGEE020)**

**Harsh Sagar (2021UGEE044)**

has been carried out under my supervision, as fulfilment for the award of the degree of Bachelor of Technology (Hons.) in Electrical Engineering at the National Institute of Technology, Jamshedpur, Jharkhand.

**Dr. Kethavath Raghavendra Naik**

Assistant professor

Department of Electrical Eng.

N.I.T. Jamshedpur

**COUNTERSIGNED BY**

**Dr. Madhu Singh**

Head of Department, Electrical Engineering

N.I.T. Jamshedpur

Date: 28/04/2025

# ACKNOWLEDGEMENTS

We consider it as our pleasure to acknowledge our gratitude to our faculty guide, **Dr. Kethavath Raghavendra Naik** under whose guidance we have carried out our project work. He has inspired us with his suggestions and helped us with technical guidance in completing the project work. We shall remain grateful to **Dr. MADHU SINGH** (H.O.D, Electrical Department) and **Prof. GAUTAM SUTRADHAR,** Director, NIT Jamshedpur, for enabling us to have an academic atmosphere by enforcing discipline to do the project work with the utmost concentration, sheer dedication and relentless efforts.

Finally, we must say that no height has been ever achieved without some sacrifices made and it is here where we owe our special debt to our parents and our friends for showing their generosity and extreme care during the complete project period.

**ADITYA NARAYAN SINGH (2021UGEE001)**

**NAVIN KUMAR (2021UGEE014)**

**ABHISHEK RANJAN (2021UGEE017)**

**PRATIM CHATTERJEE (2021UGEE018)**

**GAURAV BARA (2021UGEE020)**

**HARSH SAGAR (2021UGEE044)**

# ABSTRACT

The increasing adoption of electric vehicles (EVs) necessitates accurate prediction of EV charging power demand to optimize energy distribution, avoid grid overloads, and support smart infrastructure planning. In this study, machine learning (ML) techniques are employed to forecast EV charging power consumption based on key temporal features such as time of day. A comprehensive dataset of EV charging station usage was utilized, and multiple supervised learning models were developed, including Linear Regression, Ridge Regression, Lasso Regression, ElasticNet, Support Vector Regression (SVR), Random Forest Regression, Gradient Boosting Regression, and XGBoost Regression.

The models were trained and evaluated using a random 70-30 train-test split, and their performances were compared using standard regression metrics: Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and the Coefficient of Determination (R² Score). The XGBoost Regressor achieved the best predictive performance, delivering the highest R² score and the lowest error rates.

Additionally, the study proposes future work involving the conversion of the trained model into Embedded C for deployment on microcontrollers at EV charging stations, enabling real-time demand forecasting without reliance on cloud infrastructure. The results demonstrate that machine learning, particularly ensemble learning methods, provides a robust and scalable solution for predictive control in EV charging networks

**LITERATURE REVIEW**

The rapid adoption of electric vehicles (EVs) has introduced new challenges in energy distribution and grid management, particularly in forecasting EV charging power demand. Accurate load prediction is essential to prevent grid overloads, optimize infrastructure usage, and enable dynamic energy pricing strategies.

Traditional statistical and deterministic methods have struggled to capture the nonlinear and time-variant nature of EV charging behaviour, which is influenced by variables such as time of day, user habits, weather conditions, and regional factors. As a result, machine learning (ML) techniques have emerged as powerful alternatives for energy forecasting tasks.

Recent studies have applied models such as Support Vector Regression (SVR), Random Forests, and Gradient Boosting to predict energy consumption patterns. These models can automatically learn complex relationships from historical data without requiring explicit functional assumptions.

In particular, ensemble methods like Random Forests and Gradient Boosting have shown superior performance in handling stochastic and highly variable charging loads compared to traditional linear models.

However, much of the existing work remains dependent on cloud computing platforms, which introduce latency, security, and infrastructure dependency issues. Additionally, limited attention has been given to real-time embedded deployment of machine learning models at the edge.

**Table of Contents**

1. Introduction ………………………………………………….………. .9
2. Importance of EV Charging prediction …………………………...10-11
   1. Key stakeholders …………………………………………..….10

2.2 Future EV trends …………………………………………..….11

2.3 Impact on grid stability ……………………………………….11

1. Role of ML in EV load prediction………………………………...12-14
   1. Why traditional methods are not enough………………………12
   2. How ML helps………………………………………….……...12
   3. Proposed Methatodology Congiguration……………………….13
2. Method ……………………………………………………………15-23
   1. Dataset Overview ………………………………...……………15
   2. Chosen Prediction Models …………………………………….16

4.3 Train-test split ………………………………………………....17

4.4 Model selection …………………………………….………17-18

* 1. Imports used for analysis and training of the ML model……….18
  2. Tools used for conversion…………...………………………19-20

4.6.1 Python programming language ……………………….….19

* + 1. Libraries used…………………………………………….19
    2. Jupyter notebook ……………………………………...…20

4.7 Chosen prediction model……………………………………….20

4.8 Model comparison …………………………………..……...21-23

1. Conversion of ML code to Embedded C code ……………………24-26
   1. Why convert ML model to C …………………………………....24
   2. Train a decision tree in python…………………………………...24

5.3 Extracting the decision tree from XGBoost………….………….25

5.4 Writing MATLAB code from decision rules……………………26

5.5 Conclusion……………………………………………………….26

1. Embedded Development ………………………………………….27-30
   1. Decision Tree extraction from XGBoost………………………….27

6.2 Interpreting tree rules……………………………………………...27

6.3 MATLAB implementation for embedded testing………………....28

6.4 Embedded system suitability………………………………..…28-29

6.5 Output……………………………………………………………..29

6.6 Hand conversion to C function…………………………………....29

6.7 Hand conversion to C function……………………………………29

6.8 Embedded suitability……………………………………………...30

1. Overview of development environment and target hardware……..31-33

7.1 Code Composer Studio ………………………………………..31-32

7.2 MSP430F5529 Launchpad……………………………………32-33

7.3 Coding and Debugging…………………………………………..33

1. Embedded C code for power prediction and LED control………...34-37
   1. Structure and sections of code………………………………....35-37

8.2.1 UART Initialization………………………………………….35

8.2.2 UART Transmission Helper……………………...….……...35

8.2.3 Microcontroller system setup……………….……………….35

8.2.4 Main application logic……………………………………....36

8.2 LED blinking logic based on power prediction…………………36

8.3 Output…………………………………………………………...37

1. Applications…………………………………………………………..38
2. Future scope…………………………………………………………..39
3. Summary……………………………………………………………...40
4. Source code…………………………………………………………...41
5. References…………………………………………………………….52

# INTRODUCTION

In recent years, the global energy landscape has been changing quickly. This change is mainly driven by the need for sustainability, efficiency, and resilience. With the urgent need to fight climate change, renewable energy sources like solar and wind are becoming more common. At the same time, smart technologies such as Artificial Intelligence (AI), Internet of Things (IoT), and advanced optimization algorithms are being used to manage energy more intelligently.

Traditional power systems used to have a very simple structure — energy was generated at a central location and flowed one way to the consumers. But today, power grids are becoming much more dynamic. Now, electricity, data, and control signals move both ways across the network. This change brings a lot of challenges, like maintaining grid stability, handling the ups and downs of renewable energy, keeping systems secure from cyber threats, and making sure energy is used efficiently.

Studies show that using intelligent techniques like AI can help solve these problems. Predictive maintenance can detect faults early, saving time and money, and smart demand management can balance supply and demand more effectively. Machine Learning (ML) models especially help in forecasting renewable generation and predicting energy usage by learning from past and real-time data.

By applying these technologies, we can build a smarter, greener, and more reliable energy system for the future.

One important area where these technologies can make a big difference is in EV (Electric Vehicle) charging. The growing number of EVs is putting a lot of pressure on electricity grids. If we can predict EV charging demand accurately, we can avoid grid overloads, plan better infrastructure, and even reduce electricity costs.

## 

## 2. IMPORTANCE OF EV CHARGING PREDICTION

As electric vehicles (EVs) continue to grow in popularity, predicting the charging load has become more important than ever. If not managed properly, the sudden increase in electricity demand from EVs can create serious problems for the existing power grid.

Load prediction helps ensure that the system remains stable, energy is used efficiently, and users have a smooth charging experience. It also helps energy companies, governments, and city planners make smarter decisions.

**2.1 Key Stakeholders**

Several groups are directly affected by EV charging demand and benefit from accurate load prediction:

* Utility Companies: They need to manage the balance between electricity supply and demand. Good predictions help them avoid overloads and plan for future energy needs.
* EV Charging Station Operators: Accurate forecasts help them manage resources, avoid long wait times for customers, and improve the station’s profitability.
* Urban Planners and Policymakers: They can make better decisions about where and how many charging stations are needed, helping cities grow in a smart and sustainable way.
* EV Manufacturers and Fleet Operators: Knowing when and where high demand will occur helps them offer better services and manage their electric vehicle fleets efficiently.
* Consumers: Ultimately, EV users benefit from shorter waiting times, better charging availability, and possibly lower charging costs during off-peak hours.

**2.2 Future EV Trends**

The coming years are expected to see a massive rise in EV adoption:

* Rapid Growth of EV Sales: Many countries are setting targets to ban the sale of new petrol and diesel vehicles within the next 10–20 years. As a result, EV sales are expected to skyrocket.
* More Fast-Charging Stations: To meet user expectations for faster charging, the number of DC fast-charging stations will increase, putting even more pressure on the grid.
* Integration with Renewable Energy: Many EV charging stations will be powered by solar or wind energy, requiring smart prediction systems to deal with the variable supply from renewables.
* Vehicle-to-Grid (V2G) Technology: In the future, EVs will not only consume electricity but also send electricity back to the grid when needed, making the prediction of charging and discharging patterns even more important.

**2.3 Impact on Grid Stability**

If EV charging is not properly predicted and managed, the impact on grid infrastructure can be serious:

* Overloading of Local Transformers: Charging multiple EVs simultaneously in residential areas can overload local distribution transformers, leading to equipment failures.
* Voltage Fluctuations: Sudden spikes in demand can cause voltage drops, which affect not just EV chargers but all nearby electrical devices.
* Need for Grid Upgrades: Utilities might have to invest a lot of money in upgrading wires, transformers, and substations to handle the increased load,
* unless charging demand can be predicted and spread out intelligently.
* Energy Storage Requirements: To handle peak loads, more battery storage systems may be needed, adding extra costs if not managed smartly.

**3. Role of ML in EV load prediction**

Predicting EV charging demand is a complex task. People's driving habits, weather conditions, traffic situations, and even holidays can affect when and how much they charge their vehicles. Traditional methods like simple averages or linear forecasting cannot handle such complicated patterns well.

This is where Machine Learning (ML) comes into the picture. ML models can learn from large amounts of past data, recognize hidden patterns, and make much better predictions than traditional methods.

**3.1. Why traditional methods are not enough**

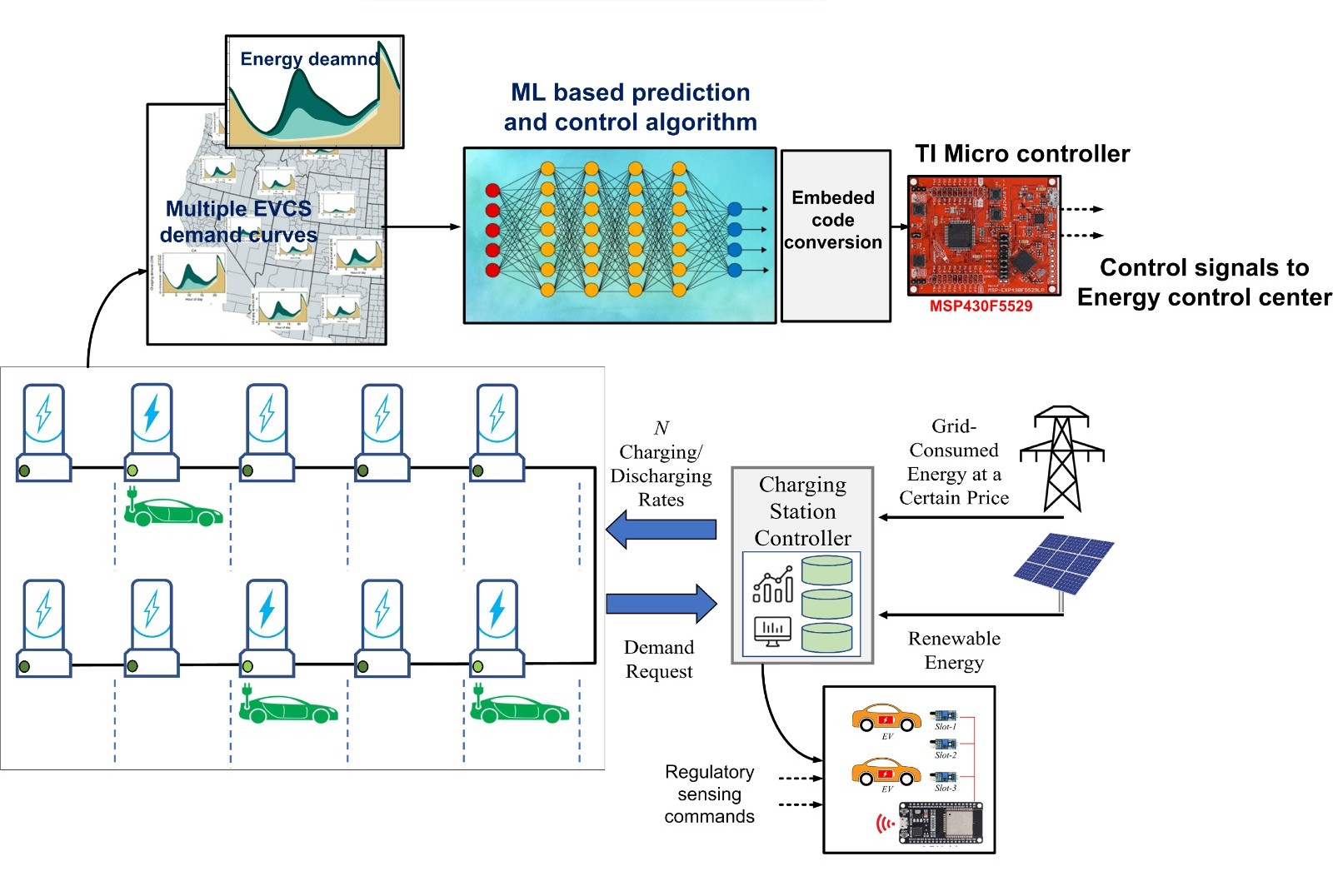
In old power systems, load prediction was mainly done using basic statistical methods, assuming that energy consumption patterns were stable and predictable. However, EV charging brings in:

* Random user behaviour: Different people have different charging routines.
* Nonlinear patterns: Charging doesn't always increase or decrease in a straight line.
* External factors: Weather, special events, and even traffic can impact charging demand.
* Simple models cannot capture these complex behaviours. They often lead to poor predictions, which can either cause overloading or underutilization of the grid.

**3.2 How Machine Learning helps**

Machine Learning offers powerful tools to deal with these challenges:

* Pattern Recognition: ML can detect hidden patterns and trends that are too complicated for humans to find manually.
* Adaptability: As new data comes in (e.g., more EVs, changing habits), ML models can be retrained and updated to stay accurate.
* Handling Large Datasets: ML models can process and learn from large amounts of historical and real-time data.
* Nonlinear Modelling: ML algorithms are good at modelling nonlinear relationships between inputs (like time, temperature, location) and outputs (charging demand).

**3.3 Proposed Methodology Configuration**

In this project, we have proposed a model that combines machine learning-based EV charging prediction with embedded system control to manage the energy usage of multiple electric vehicle charging stations (EVCS). The main goal of the model is to predict energy demand accurately and control the charging rates smartly in real time.

The system starts by collecting energy demand data from multiple EV charging stations. These stations record how much electricity is consumed at different times, creating demand curves that show usage patterns throughout the day. Using this data, we train a machine learning model that can learn and predict future charging demand. A tree-based model like XGBoost is used because it can handle nonlinear and complex data very effectively.

After training, the ML model is converted into embedded C code and uploaded into a Texas Instruments MSP430F5529 microcontroller. This microcontroller predicts EV charging demand locally at the station without relying on cloud servers. It then sends control signals based on the predictions to manage the charging process

The Charging Station Controller uses these predictions to decide the charging and discharging rates for each EV. It also manages how much energy to draw from renewable sources like solar panels and when to use grid electricity, helping to reduce costs. If renewable energy is available, it is prioritized; otherwise, grid electricity is used based on the current energy price.

The model also supports future Vehicle-to-Grid (V2G) operations, where electric vehicles can supply energy back to the grid when needed. Real-time regulatory sensing is used to monitor vehicle slots, energy consumption, and other important parameters to make smart decisions.

In short, the proposed system enables smart and efficient EV charging by using machine learning predictions and embedded hardware. It helps manage energy resources better, reduces dependency on the grid, supports the use of renewable energy, and makes EV charging more reliable and cost-effective.

Key Technologies Used**:**

* Machine Learning (Prediction of EV load demand)
* Embedded Systems (MSP430F5529 microcontroller)
* Renewable Energy Integration (Solar energy)
* Grid-Interactive Smart Charging
* Real-Time Control Signals

## 4. Method

## This chapter describes the methodology followed for predicting EV charging power demand using machine learning techniques.

## Multiple data sources were merged, pre-processed carefully to avoid overfitting, and finally used to train and evaluate several machine learning models.

## Evaluation steps (metrics like RMSE, MAE, R²) are discussed separately since standard ones were used.

## 4.1 Dataset Overview

## Dataset Composition:

## Features:

## Time (Hour of the day)

## (In extended work) Temperature, Location, Traffic, Weather Data

## Target Variable:

## Power per charging event (in kWh)

## Dataset Statistics:

## Size: 1329 samples × 7 features

## No missing values.

## Hourly-based time-series structure.

## Observations:

## Clear patterns like evening peak hours in charging demand.

## Power consumption heavily depends on time of day.

## Extended features (weather, holidays) were used for richer predictive modelling.

## 4.2 Chosen Prediction Models

## After building the final pre-processed dataset, several machine learning regression models were selected and compared:

## Models used:

## Linear Regression

## Ridge Regression

## Lasso Regression

## Elastic Net

## Support Vector Regression (SVR)

## Random Forest Regression

## Gradient Boosting Regression

## XGBoost Regression

## Model Training Details:

## Data Split: 70% for training and 30% for testing using train\_test\_split (X, y, test\_size=0.3, random\_state=42).

## Normalization:

## Scaling applied to models sensitive to feature magnitude (Ridge, Lasso, Elastic Net, SVR).

## Tree-based models (Random Forest, Gradient Boosting, XGBoost) were trained without scaling.

## Evaluation Metrics:

## Root Mean Squared Error (RMSE)

## Mean Absolute Error (MAE)

## Coefficient of Determination (R² Score)

## 

**4.3 Train-Test Split**

For the evaluation and validation of machine learning models, a random train–test split was performed on the dataset.

Following best practices, the dataset was partitioned into:

* Training Set: 70% of the dataset
* Testing Set: 30% of the dataset

The test samples were selected randomly rather than sequentially along the time axis.

This strategy ensures that the training and testing sets capture a wide range of usage patterns throughout the day and week, rather than being biased towards certain time periods.

Rationale for Random Splitting:

* In real-world EV charging systems, models must be retrained periodically as user behaviour, market adoption, and infrastructure evolve dynamically.
* Using older data to predict newer behaviour without regular retraining would lead to degraded model performance.
* Random splitting thus provides a realistic evaluation of model performance assuming periodic retraining.

The splitting process was performed using the train\_test\_split () function from scikit-learn, with a fixed random state (random\_state=42) to ensure experiment reproducibility across different runs.

**4.4 Model Selection**

The choice of an appropriate machine learning model is crucial for accurately predicting EV charging power demands.

In this work, an extensive model comparison study was undertaken, aiming to:

* Identify models with strong predictive accuracy.
* Favor models that generalize well to unseen data.
* Evaluate models that can capture complex, nonlinear relationships inherent in EV charging behaviour.

Unlike some automated libraries like PyCaret, a manual, step-by-step model comparison was performed using Python’s native ML libraries, notably:

* scikit-learn for traditional models
* XGBoost for advanced ensemble methods

Model Evaluation Approach:

Each model was trained on the training set and tested on the testing set.

Standard performance metrics were computed:

* Root Mean Squared Error (RMSE): Penalizes large prediction errors more heavily.
* Mean Absolute Error (MAE): Measures the average magnitude of errors.
* Coefficient of Determination (R² Score): Indicates how well the model explains the variance in the data.

Models were initially evaluated without hyperparameter tuning to ensure a fair baseline comparison.

Based on the results, best-performing models were shortlisted for final analysis and slight fine-tuning.

Note: Models prone to high bias (like simple linear models) were retained for baseline comparison, while highly complex deterministic models were avoided to maintain interpretability and practical deployment considerations.

## 

**4.5 Imports used for analysis and training of the ML model**

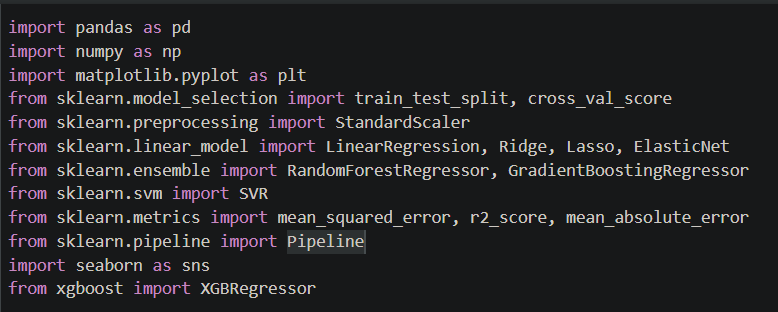
****

Fig 1: Imports in python code

**4.6 Tools used for conversion**

1. Python Programming language

2. Libraries

3. Jupyter Notebook

**4.6.1 Python programming language**

Python was selected as the primary programming language for model development due to its simplicity, flexibility, and extensive ecosystem of libraries dedicated to data science and machine learning. Python’s ability to quickly prototype, visualize, and deploy models made it an ideal choice for this research work.

**4.6.2 Libraries Used**

4.6.2.1 Pandas:

**Pandas** is a high-performance library built for data manipulation and analysis.

In this project, pandas were primarily used for:

* **Reading** datasets from .csv files.
* **Cleaning** and **preprocessing** the data.
* **Extracting** relevant features (Time) and target variables (Power per Charging Event).
* **Saving** final processed data and predictions into CSV files.

Pandas provided efficient DataFrame structures that made handling large datasets straightforward and intuitive.

4.6.2.2 Numpy:

numpy is the foundational package for numerical computing in Python.  
Its usage in this project included:

* Mathematical operations like calculating square roots (for RMSE computation).
* Array manipulations when creating sequences for full-day predictions.
* Enhancing the computational efficiency during model evaluation and testing phases.

The combination of numpy arrays with pandas DataFrames streamlined the data analysis pipeline.

4.6.2.3 Scikit-learn:

**scikit-learn (sklearn)** is one of the most popular machine learning libraries for Python.  
It was employed in this project for:

* **Splitting the dataset** into training and testing sets (train\_test\_split).
* **Evaluating model performance** using metrics such as:
  + Mean Squared Error (MSE)
  + Root Mean Squared Error (RMSE)
  + Mean Absolute Error (MAE)
  + R-squared Score (R²)

These built-in evaluation functions allowed for rigorous, standardized performance measurement of the predictive model.

4.6.2.4 Matplotlib:

**matplotlib** is a leading visualization library in Python.  
Its role in the project included:

* **Plotting actual vs predicted values** to visually assess model accuracy.
* **Visualizing the model’s prediction over a full day** to understand trends.
* **Generating residual plots** for deeper error analysis.
* **Plotting decision trees** from the XGBoost model for interpretability.

All plots were saved as high-quality images, which were later incorporated into analysis reports and presentations

**4.6.3 Jupyter notebook**

Jupyter Notebooks served as the main environment for:

* Writing, running, and debugging Python code interactively.
* Documenting experiments by combining code, results, plots, and explanations in a single file.
* Organizing training, testing, visualization, and rule extraction tasks systematically.
* Enabling rapid iteration and reproducibility throughout the research.

The notebooks also facilitated easy sharing of progress and results with collaborators and supervisors.

**4.7 Chosen Prediction Model**

Based on the comparative analysis, the following machine learning models were selected for final evaluation:

Model Description:

* XGBoost Regressor: Gradient boosting algorithm with regularization, offering excellent prediction performance, even on complex tabular data.
* Random Forest Regressor: Ensemble of decision trees with bootstrap aggregation, resistant to overfitting and capable of capturing nonlinearities.
* Gradient Boosting Regressor: Sequential ensemble model minimizing the residual errors of prior models, suitable for fine-grained prediction tasks.
* Linear Models (Ridge, Lasso, ElasticNet): Regularized versions of linear regression, effective as lightweight baselines and interpretable models.
* Support Vector Regression (SVR): Model that uses a margin of tolerance for regression, effective on small datasets but sensitive to feature scaling.

**Baseline Model:** Linear Regression To set a basic reference point for evaluating improvements, a simple Linear Regression model was also trained.

**Feature:** Only "Time of Day" was used as the predictor.

**Outcome:** The model provided a useful benchmark.

However, it showed clear limitations in capturing the complex, nonlinear trends in the EV charging data.

Thus, the performance of more sophisticated ensemble models was clearly superior.

**4.8 Model Comparison**

Each model was trained on the same dataset (with Time of Day as the main feature) and evaluated using the test dataset.

Performance was measured using the following metrics:

* Root Mean Squared Error (RMSE)
* Mean Absolute Error (MAE)
* Coefficient of Determination (R² Score)

The XGBoost model was chosen for final deployment due to its strong predictive capability.

Performance Metrics (XGBoost):

* RMSE: 1.70 kWh
* MAE: 1.24 kWh
* R² Score: 0.76

A screenshot of a black screen

AI-generated content may be incorrect.

Table 1: Comparison table for all models

This indicates that the model can explain 76% of the variance in EV charging power consumption, with relatively low prediction errors.

Prediction Behaviour:

* The model successfully captured the daily patterns in charging demand.
* It accurately modelled the evening peak observed typically between 5 PM to 9 PM.
* Minor under-predictions occurred during early morning hours, likely due to less consistent user behaviour at those times.

Feature Importance

While the initial model primarily used Time of Day as the feature, extended work suggested that additional features could significantly improve performance​.

A graph with colorful dots

AI-generated content may be incorrect.

Fig 2: Actual Vs Predicted Values

A graph with a line going up

AI-generated content may be incorrect.

Fig 3: Predicted power per Charging for day of XGBoost

**5. Conversion of ML code to Embedded C code:**

**5.1 Why Convert ML Models to Embedded C?**

Embedded systems often have tight constraints in terms of processing power, memory, and storage. While ML models, especially deep learning models, are typically trained in environments with powerful hardware and large datasets, deploying these models on embedded systems requires efficient and lightweight code that can run on minimal resources.

By converting ML code to Embedded C, you can:

Deploy models on microcontrollers or embedded systems with limited hardware capabilities.

Enable real-time, on-device predictions without relying on cloud-based inference, thus reducing latency and enhancing privacy.

Utilize low-power devices in applications like IoT, robotics, and automotive systems.

**5.2 Train a Decision Tree in Python**

Code in XGBoost to train on dataset:

A screen shot of a computer program

AI-generated content may be incorrect.

Fig 4: Code for Training data

**5.3 Extracting the Decision Tree from XGBoost**

Code to extract on dataset:

**A computer screen shot of a black background

AI-generated content may be incorrect.**

Fig 5: Code for extracting

* best\_model.get\_booster () accesses the internal booster (core model).
* .get\_dump() dumps all trees in the model in a text format.
* rules[0] extracts the first tree (tree number 0).

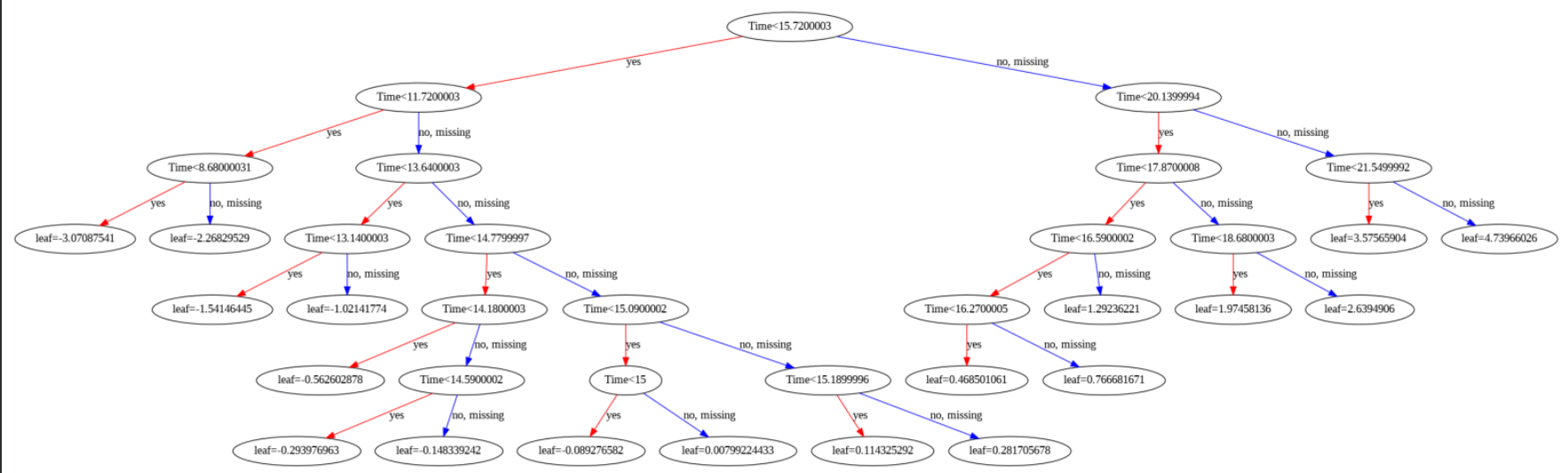
****

Fig 6: Exctracted Decision Tree

**5.4 Writing MATLAB Code from Decision Rules**

To simulate this in MATLAB, we'll:

* Use a series of nested if-else statements to implement the same logic.
* Given a Time value, the code will predict the Power output by following the tree.

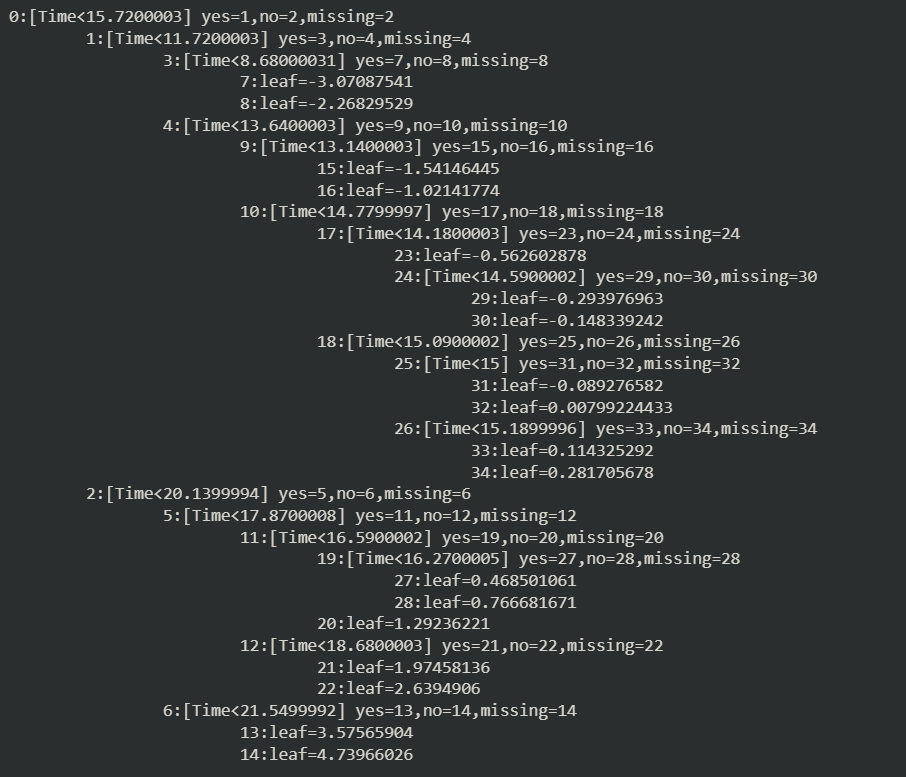
****

Fig 7: Extracted If Else from Tree

**5.5 Conclusion**

We successfully extracted the first decision tree from the trained XGBoost model.

We converted the tree rules into a MATLAB function.

The MATLAB code allows for manual prediction using the first tree, useful for:

* Embedded systems.
* Resource-constrained applications.
* Model explainability.

**6. Embedded Development**

In addition to data-driven model training and analysis, this project also focused on translating the machine learning model into a form suitable for embedded system deployment.

The goal was to **extract the trained decision tree rules** from the XGBoost model and **implement them using MATLAB and C/C++ code** for testing on an embedded platform.

This involved several steps:

**6.1. Decision Tree Extraction from XGBoost**

After training the XGBoost model in Python, the internal structure of the first decision tree was extracted using:

This dump outputs the tree’s structure in a simple, readable text format, containing:

* Node splitting conditions (e.g., Time < 5.5)
* Predicted leaf values at terminal nodes
* Tree hierarchy and branching logic

Each node specifies:

* Which feature to split on (e.g., Time)
* A threshold value for decision making
* Left and right children depending on the comparison result

6.**2. Interpreting the Tree Rules:**

The extracted tree rules were manually interpreted to understand:

* The **thresholds** used at each split.
* The **flow of logic** (whether to go left or right based on Time value).
* The **output prediction** corresponding to each leaf node.

This logical structure mirrors traditional **if-else** decision-making used in embedded systems.

**6.3. MATLAB Implementation for Embedded Testing**

To verify correctness before hardware deployment, the tree logic was manually coded in MATLAB, representing the model as a sequence of if-else conditions.

In this function:

* **Input**: Time in hours.
* **Output**: Predicted Power per Charging Event.
* The function uses nested if-else statements exactly mimicking the tree structure extracted from XGBoost.

Thus, MATLAB served as a validation platform before translating the logic to embedded C or microcontroller firmware.

**A screen shot of a computer program

AI-generated content may be incorrect.**

Figure 6: Sample code of MATLAB

**6.4. Embedded System Suitability**

The reasons why this decision-tree-based approach is highly suitable for embedded deployment are:

* Low computational complexity (simple comparisons and assignments).
* No need for floating-point heavy matrix operations.
* Deterministic behaviour, essential for real-time applications.
* Easy optimization using standard C/C++ on microcontrollers.

In future work, the same MATLAB structure can be auto-converted to C code or directly hand-coded for use with microcontrollers like MSP430, ARM Cortex, or ESP32.

**6.5. Output**

After executing the code on MATLAB for multiple time inputs and plotted the results:

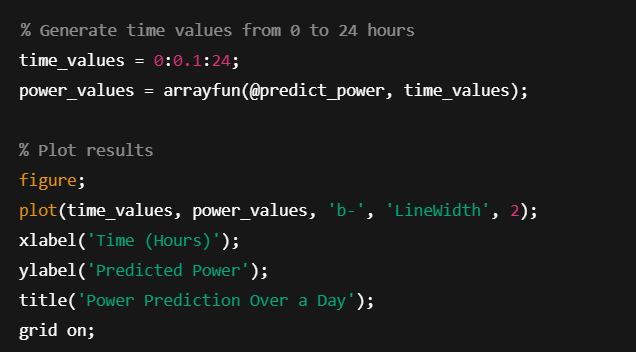


Fig 7: Code for MATLAB

Output:

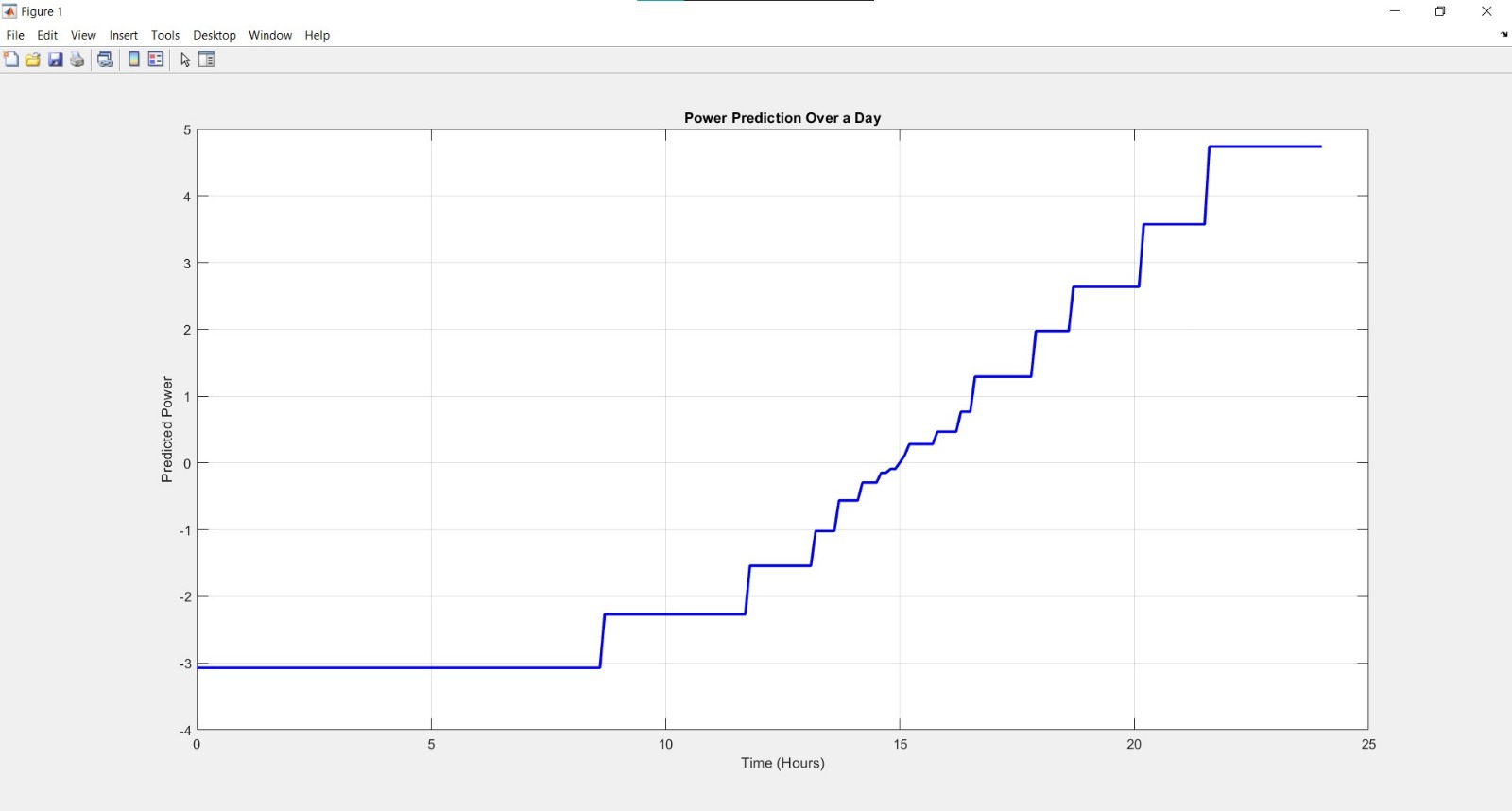


Fig 8: Power Predicted Vs Time

**6.6. Hand Conversion into C Functions**

* The entire logic tree was re-implemented in C language using nested if-else structures.
* Floating-point precision (float type) was preserved wherever necessary.
* No dynamic memory allocation (malloc) was used to ensure deterministic behaviour.
* Constants were defined at the beginning for easy tuning and maintainability.

**6.7. Hand Conversion into C Functions**

* The C function was wrapped inside a callable API float predict\_power(float input).
* UART printing was integrated to output prediction results for debugging and verification.
* The converted function was placed in a separate source (.c) and header (.h) file structure.

**6.8. Embedded System Suitability**

MATLAB’s **codegen** tool enables automatic conversion of MATLAB functions into highly optimized C or C++ code.  
It is a part of the MATLAB Coder toolbox and is particularly useful for deploying algorithms onto embedded systems where execution speed and resource constraints are critical.



Fig 9: Code to generate C

**7. Overview of Development Environment and Target Hardware:**

In this work, Code Composer Studio (CCS) was utilized as the primary Integrated Development Environment (IDE) for embedded system programming, and the MSP430F5529 LaunchPad was selected as the target microcontroller platform.

Both tools were chosen due to their flexibility, low power consumption capabilities, and strong ecosystem support for embedded applications.

**7.1 Code Composer Studio (CCS)**

Code Composer Studio is a professional-grade IDE developed by Texas Instruments (TI) specifically tailored for their embedded processors, including the MSP430, TMS320C2000, Sitara ARM, and other device families.

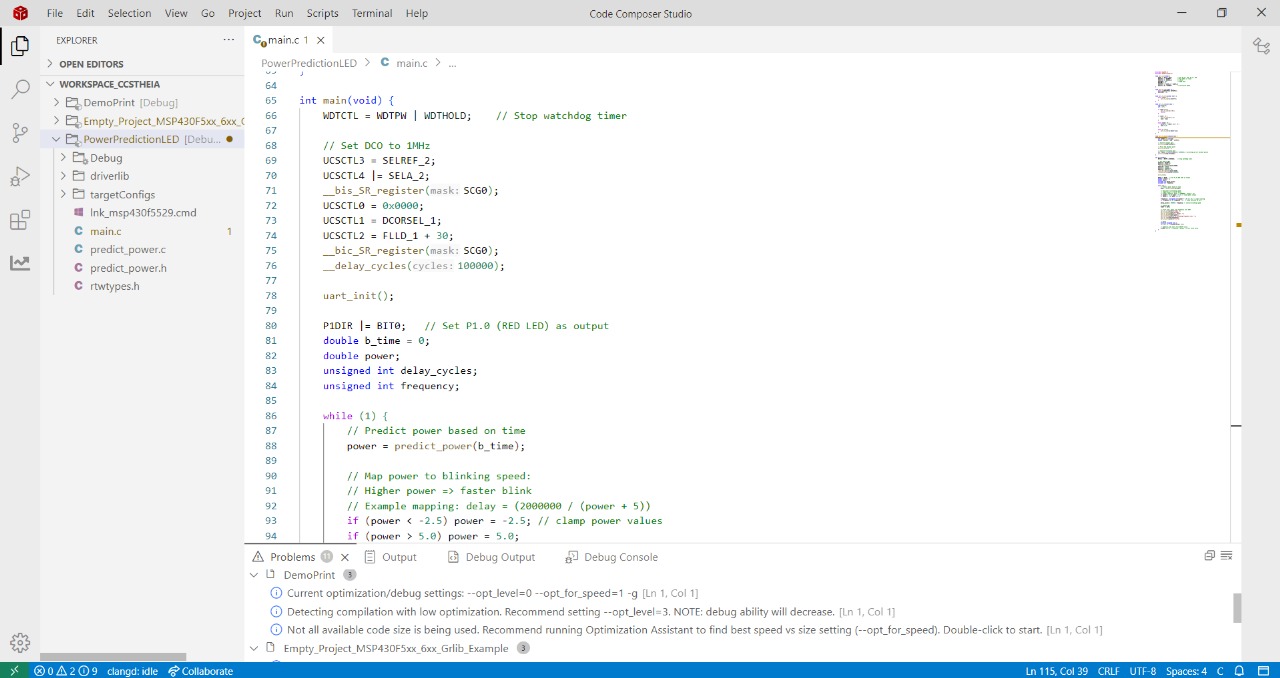


Image 1: CCS studio

Key Features:

* Integrated Compiler and Debugger: Built-in TI C/C++ compiler optimized for embedded applications.
* Support for JTAG/SBW Debugging: Seamless integration with on-board debuggers for real-time monitoring and debugging.
* RTOS and Driver Integration: Native support for TI-RTOS, FreeRTOS, and peripheral driver libraries.
* Graphical Configuration Tools: Easy clock setup, pin multiplexing, and peripheral configuration.
* EnergyTrace™ Technology: Unique to TI, enabling real-time energy profiling (particularly important for low-power applications like MSP430).

Advantages:

* Free Licensing for small devices like MSP430.
* Multi-core and Multi-target Support: Projects can scale from simple MCUs to complex multi-core systems.
* Flexible Build Options: Supports GNU Makefiles, automated builds, and command-line operation.
* Cross-platform: Available for Windows, Linux, and MacOS.

**7.2 MSP430F5529 LaunchPad (MSP430F5529LP)**

The MSP430F5529LP is a development board based on the MSP430F5529 microcontroller, a member of Texas Instruments’ ultra-low-power 16-bit MSP430 family.

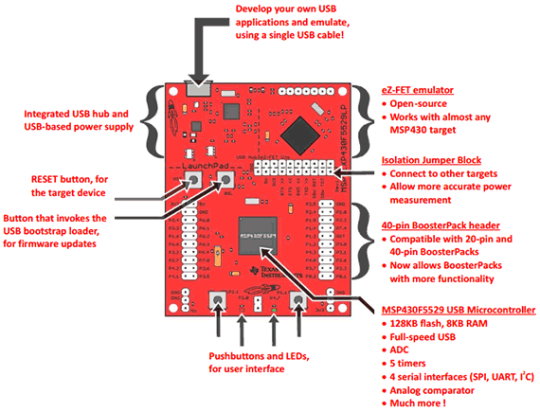


Image 2: MSP430F5529 microcontroller parts

MSP430F5529 Microcontroller Overview:

Feature Specification

Core 16-bit RISC CPU

Clock Speed Up to 25 MHz

Memory 128 KB Flash, 8 KB SRAM

Operating Voltage 1.8V to 3.6V

Timers 4 × 16-bit Timers

ADC 12-bit SAR ADC with internal reference

USB Interface USB 2.0 PHY and controller

Communication Interfaces UART, I2C, SPI

The MSP430F5529 was selected due to its:

* Sufficient Flash and RAM resources for algorithm implementation.
* Integrated USB support, enabling direct PC interfacing for testing.
* Very low power consumption ideal for battery-operated or energy-sensitive applications.
* Wide community and example project support.

**7.2 Coding and Debugging:**

Implemented modular code structure.

Used CCS debugger with breakpoints, variable watch windows, and energy profiling.

Flashing and Testing:

* Directly flashed code using on-board eZ-FET programmer.
* Observed real-time outputs via UART monitoring tools (e.g., Tera Term, PuTTY).

Optimization and Profiling:

* Analysed execution timing and energy consumption.
* Optimized code further for runtime efficiency.

**8. Embedded C Code for Power Prediction and LED Control on MSP430:**

We implemented an embedded application on the **MSP430** microcontroller that integrates **power prediction** with **visual and UART feedback**.

The system predicts power output based on a time input using a function (predict\_power) generated from MATLAB via codegen.

The predicted power determines the blinking speed of an onboard LED and sends periodic UART outputs that detail the current time, predicted power, and blinking frequency.

This project demonstrates the end-to-end workflow from MATLAB algorithm development to embedded deployment on MSP430 hardware, including UART communication and GPIO control.

Main function code:

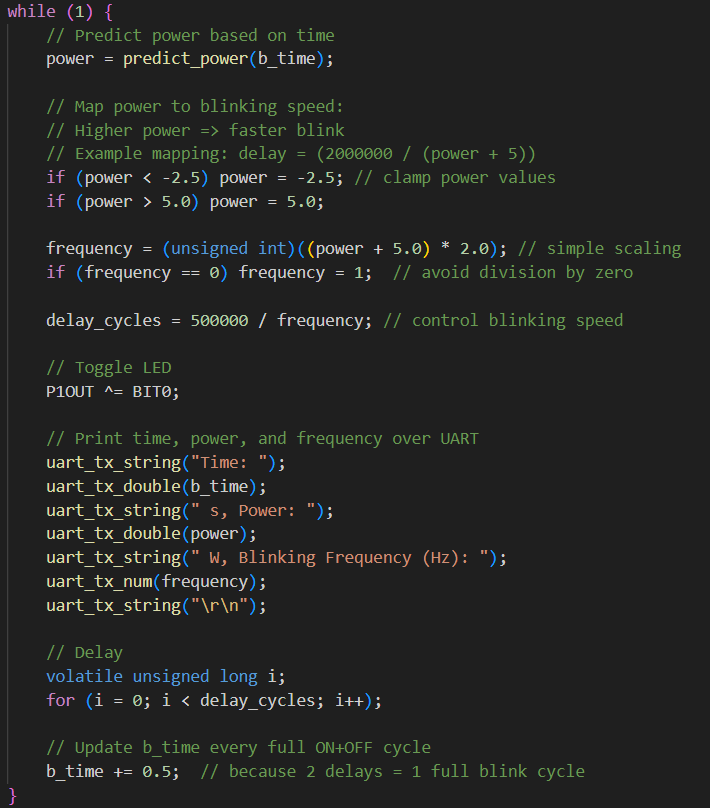
****

Fig 10: main function

**8.2 Structure and Sections of the code:**

**8.2.1 UART Initialisation (uart\_init):**

* Purpose: Configure UART hardware on the MSP430 for 9600 baud communication.
* Details:
* Pins P4.4 and P4.5 are set for UART transmission (TXD) and reception (RXD).
* The USCI\_A1 (Universal Serial Communication Interface) module is configured:
* SMCLK is selected as the clock source.
* Baud rate is set using UCA1BR0 and UCA1BR1.
* Modulation settings are configured via UCA1MCTL.
* Module is taken out of reset to start UART operation.

**8.2.2 UART Transmission Helper Functions:**

These utility functions abstract UART communication for sending different data types cleanly and repeatedly.

* + uart\_tx\_char (char c):
    - Sends a single character over UART.
  + uart\_tx\_string (char \*str):
    - Sends a null-terminated string character by character.
  + uart\_tx\_num (int num):
    - Sends an integer number as a string over UART.
    - Handles negative numbers and zero explicitly.
  + uart\_tx\_double (double num):
    - Sends a floating-point number over UART.
    - Splits into integer and fractional parts, prints up to 6 decimal places**.**

**8.2.3 Microcontroller System Setup:**

* + **Clock Configuration**:
    - DCO (Digitally Controlled Oscillator) is set to **1 MHz** frequency for stable operation.
    - SMCLK and ACLK sources are configured.
  + **GPIO Configuration**:
    - P1.0 (red LED on MSP430) is configured as output.

**8.2.4 Main Application Logic (main function):**

* + Initialization:
* Stops the watchdog timer.
* Calls uart\_init () to set up UART.
* Sets clock parameters for system stability.
  + Variable Declarations:
* b\_time: Represents the "time" input to the predictor function.
* power: Predicted power output based on time.
* delay\_cycles and frequency: Control timing of LED blinking.
  + Infinite Loop (while (1)):

1. Prediction:
   * The predict\_power(b\_time) function is called to get the power output.
2. Mapping Prediction to Blink Speed:
   * Power is clamped between -2.5 and 5.0 for stability.
   * A simple scaling maps power values to a blinking frequency.
   * Delay cycles are inversely proportional to frequency (higher power = faster blinking).
3. LED Blinking:
   * The red LED (P1.0) is toggled every calculated delay cycle.
4. UART Output:
   * The current time, predicted power, and blinking frequency are printed to UART for monitoring.
5. Time Update:
   * b\_time is incremented by 0.5 seconds per full ON+OFF cycle.

**8.3 LED blinking logic based on power prediction:**

Prediction:

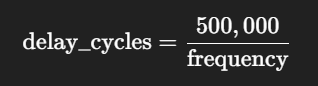
* Calls predict\_power(b\_time) to obtain the power output.
* Prediction function was exported using MATLAB Coder (codegen).

Mapping Power to Frequency:

Formula:

* **frequency (Hz)=2×(power+5)**

Calculating Delay:



**8.4 Output:**

Each UART transmission includes:

* Time: [b\_time] s
* Power: [predicted power] W
* Blinking Frequency: [frequency] Hz

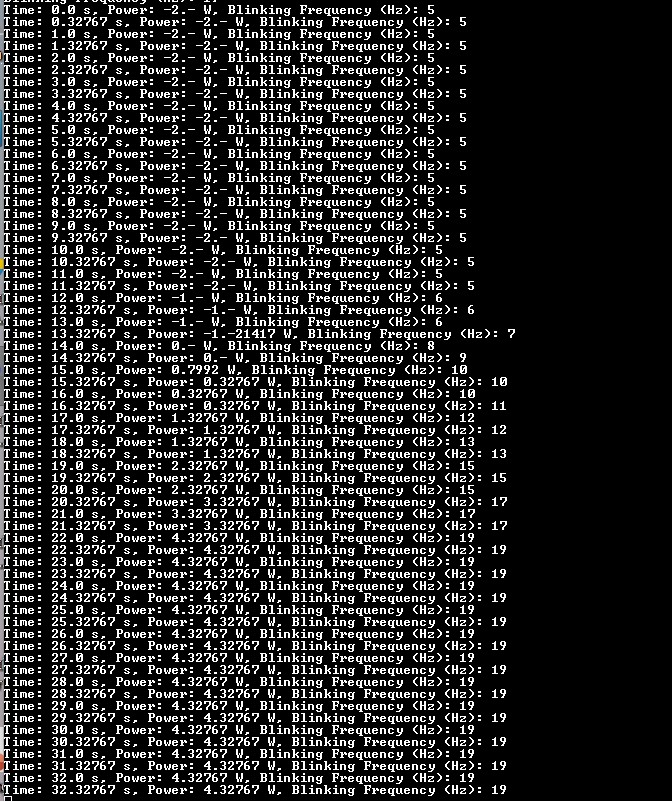


Fig 11: Output from Microcontroller

**9. Applications:**

**Smart Grid Management**

Load Balancing: Accurate EV load predictions help utility companies balance electricity demand and supply more effectively. By forecasting peak and off-peak loads, operators can prevent grid overloads and reduce the risk of blackouts.

Dynamic Pricing: Smart grids can implement dynamic pricing strategies based on predicted EV charging behaviour, encouraging users to charge during off-peak hours and optimizing energy consumption.

**Charging Station Optimization**

Infrastructure Planning: Predicting when and where the highest charging demand will occur helps in strategically placing new charging stations and expanding existing facilities.

Resource Allocation: Charging stations can better allocate power among multiple vehicles, minimizing customer waiting times and ensuring efficient use of available electricity.

**Renewable Energy Integration**

Green Charging: By aligning EV charging times with periods of high renewable energy production (like solar noon or windy nights), the project supports the greater integration of renewable energy sources into the grid.

Energy Storage Optimization: Predictions help manage energy storage systems (like batteries) that temporarily store excess renewable energy and supply it during high charging demand.

**Vehicle-to-Grid (V2G) Applications**

Bidirectional Energy Flow: In a future with V2G systems, accurate load forecasting will allow vehicles to not only draw power but also supply it back to the grid when needed, improving overall grid resilience and efficiency.

**Embedded Systems and IoT Devices**

On-device Prediction: By converting the machine learning model into embedded C code (through MATLAB code generation), predictions can be made directly on microcontrollers without needing cloud access.

Real-time Decision Making: IoT-enabled charging points or smart energy meters can make real-time decisions based on immediate load predictions, ensuring ultra-fast response times even with minimal hardware resources.

**Electric Fleet Management**

Fleet Optimization: Companies managing fleets of electric vehicles (like public transportation, logistics) can use these predictions to plan charging schedules, minimize downtime, and reduce operational costs.

Battery Health Management: Predictive charging behaviour also allows operators to prolong battery life by avoiding deep discharges and maintaining optimal charge cycles.

**Policy Making and Urban Planning**

Urban Energy Planning: Governments and city planners can use load prediction models to design energy-efficient cities, ensure proper electricity distribution, and support sustainable mobility initiatives.

Regulation and Incentive Programs: Policymakers can create incentive programs for smart charging and renewable-based charging based on predictive analysis of EV load patterns.

**10. Future Scope:**

**Expansion to Complex Models**

The current work focuses on decision trees. Future efforts could extend to implementing lightweight versions of Random Forests or Gradient Boosting algorithms on microcontrollers, balancing prediction accuracy with system resource constraints.

**Automated Code Generation**

Developing a semi-automatic process to directly convert trained machine learning models from Python/MATLAB into optimized C code for embedded deployment would improve scalability and reduce manual coding effort.

**Deployment on Advanced Hardware**

While the MSP430F5529LP is ideal for low-power applications, higher-performance microcontrollers like STM32 series or small ARM Cortex-M devices could allow the execution of more complex models with faster response times.

**11. Summary:**

This project demonstrates a complete pipeline — from machine learning modeling to embedded deployment — highlighting the future possibilities of integrating artificial intelligence into real-world electric mobility and energy systems.

The motivation behind this project stems from the challenges faced by power systems due to uncoordinated and unpredictable EV charging patterns. EVs introduce new types of loads that vary by time, season, region, and user behaviour, making traditional load prediction methods insufficient. Therefore, accurate load forecasting is essential for planning energy resources, reducing operational costs, and enhancing user satisfaction.

To achieve this, we first explored various machine learning algorithms such as Linear Regression, Decision Trees, Random Forests, Support Vector Machines (SVM), Gradient Boosting, and XGBoost. Each algorithm offers distinct advantages in handling the non-linear and complex nature of EV charging data. We evaluated model performance using standard error metrics like Mean Absolute Error (MAE), Root Mean Square Error (RMSE), and Mean Absolute Percentage Error (MAPE).

The project workflow involved building the model in Python using decision tree algorithms, as they provide simple, interpretable, and fast predictions suitable for real-time applications. After obtaining satisfactory results in Python, the next step was to translate the machine learning model to an embedded environment. For this, the decision tree model was first coded in MATLAB, and using MATLAB Coder tools, the model was converted into a form close to Embedded C code. This step is critical for deploying the prediction model into real-world low-power embedded devices, such as those used in smart meters or EV chargers.

Throughout the project, we identified and addressed several key challenges:

* Data variability due to user behaviour, battery state, weather, and driving patterns.
* Real-time prediction needs to ensure fast and accurate decision-making for dynamic systems.
* Seasonal and regional differences that influence charging habits.
* Integration of diverse data sources such as user profiles, historical load data, and environmental factors.

**11. Source Code:**

11.1. For Selecting the best model: [link](https://colab.research.google.com/drive/1VElKQEZJWAwOHNT0ZCZq0SHBBuQLZ3W0?usp=sharing#scrollTo=XGeG-aNcTF7Z)

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split, cross\_val\_score

from sklearn.preprocessing import StandardScaler

from sklearn.linear\_model import LinearRegression, Ridge, Lasso, ElasticNet

from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor

from sklearn.svm import SVR

from sklearn.metrics import mean\_squared\_error, r2\_score, mean\_absolute\_error

from sklearn.pipeline import Pipeline

import seaborn as sns

from xgboost import XGBRegressor

data = pd.read\_csv('/Final\_data.csv')

print(data.head())

print("\nData shape:", data.shape)

print("\nData info:")

data.info()

print("\nSummary statistics:")

print(data.describe())

print("\nMissing values:")

print(data.isnull().sum())

# Convert 'Time' to hours of the day (assuming it's already in hours)

# If 'Time' is in decimal hours (e.g., 0.15, 0.26, etc.), we can keep it as is

# Let's map it to hours of the day (0-24)

# For simplicity, let's assume the data represents a 24-hour cycle

# Extract features and target

X = data[['Time']]  # Using 'Time' as our predictor

y = data['Powerperchargingevent']  # Target variable: power per charging event

# Split the data

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# Data visualization

plt.figure(figsize=(12, 6))

plt.scatter(data['Time'], data['Powerperchargingevent'], alpha=0.5)

plt.title('Power Per Charging Event vs. Time')

plt.xlabel('Time (hours)')

plt.ylabel('Power per Charging Event')

plt.grid(True)

plt.savefig('power\_vs\_time\_scatter.png')

plt.show()

plt.close()

# Create a list of different regression models to evaluate

models = {

    'Linear Regression': LinearRegression(),

    'Ridge Regression': Ridge(),

    'Lasso Regression': Lasso(),

    'ElasticNet': ElasticNet(),

    'Support Vector Regression': SVR(),

    'Random Forest': RandomForestRegressor(random\_state=42),

    'Gradient Boosting': GradientBoostingRegressor(random\_state=42),

    'XGBoost': XGBRegressor(random\_state=42)

}

# Dictionary to store results

results = {}

# Evaluate each model

for name, model in models.items():

    # Create a pipeline with scaling for some models that need it

    if name in ['Ridge Regression', 'Lasso Regression', 'ElasticNet', 'Support Vector Regression']:

        pipeline = Pipeline([

            ('scaler', StandardScaler()),

            ('model', model)

        ])

    else:

        pipeline = Pipeline([

            ('model', model)

        ])

    # Train the model

    pipeline.fit(X\_train, y\_train)

    # Make predictions

    y\_pred = pipeline.predict(X\_test)

    # Evaluate the model

    mse = mean\_squared\_error(y\_test, y\_pred)

    rmse = np.sqrt(mse)

    mae = mean\_absolute\_error(y\_test, y\_pred)

    r2 = r2\_score(y\_test, y\_pred)

    # Store results

    results[name] = {

        'RMSE': rmse,

        'MAE': mae,

        'R²': r2,

        'Model': pipeline

    }

    print(f"\n{name} Results:")

    print(f"RMSE: {rmse:.4f}")

    print(f"MAE: {mae:.4f}")

    print(f"R²: {r2:.4f}")

# Create a DataFrame for comparison

comparison\_df = pd.DataFrame({

    'Model': list(results.keys()),

    'RMSE': [results[model]['RMSE'] for model in results],

    'MAE': [results[model]['MAE'] for model in results],

    'R²': [results[model]['R²'] for model in results]

})

# Sort by R²

comparison\_df = comparison\_df.sort\_values('R²', ascending=False).reset\_index(drop=True)

print("\nModel Comparison:")

print(comparison\_df)

# Get the best model

best\_model\_name = comparison\_df.iloc[0]['Model']

best\_model = results[best\_model\_name]['Model']

# Visualize predictions from the best model

plt.figure(figsize=(12, 6))

plt.scatter(X\_test, y\_test, color='blue', alpha=0.5, label='Actual Values')

plt.scatter(X\_test, best\_model.predict(X\_test), color='red', alpha=0.5, label='Predicted Values')

plt.title(f'Actual vs Predicted Values ({best\_model\_name})')

plt.xlabel('Time (hours)')

plt.ylabel('Power per Charging Event')

plt.legend()

plt.grid(True)

plt.show()

plt.savefig('best\_model\_predictions.png')

plt.close()

# Plot all models' predictions on the test set

plt.figure(figsize=(15, 8))

plt.scatter(X\_test, y\_test, color='black', alpha=0.5, label='Actual Values')

colors = ['red', 'blue', 'green', 'purple', 'orange', 'brown', 'pink', 'gray']

for (name, result), color in zip(results.items(), colors):

    model = result['Model']

    predictions = model.predict(X\_test)

    plt.scatter(X\_test, predictions, color=color, alpha=0.3, label=f'{name} Predictions')

plt.title('All Models: Actual vs Predicted Values')

plt.xlabel('Time (hours)')

plt.ylabel('Power per Charging Event')

plt.legend()

plt.grid(True)

plt.show()

plt.savefig('all\_models\_predictions.png')

plt.close()

# Generate predictions for each hour of the day (0-24)

hours = np.arange(0, 24, 0.5).reshape(-1, 1)  # Half-hour intervals

predictions = {}

for name, result in results.items():

    model = result['Model']

    pred = model.predict(hours)

    predictions[name] = pred

# Create a DataFrame with the hourly predictions

hourly\_predictions = pd.DataFrame({

    'Hour': hours.flatten()

})

for name, pred in predictions.items():

    hourly\_predictions[name] = pred

# Plot the hourly predictions for all models

plt.figure(figsize=(15, 8))

for name in predictions.keys():

    plt.plot(hourly\_predictions['Hour'], hourly\_predictions[name], label=name)

plt.title('Predicted Power per Charging Event Throughout the Day')

plt.xlabel('Hour of Day')

plt.ylabel('Power per Charging Event')

plt.legend()

plt.grid(True)

plt.show()

plt.savefig('hourly\_predictions.png')

plt.close()

# Feature importance for tree-based models

tree\_models = ['Random Forest', 'Gradient Boosting', 'XGBoost']

for name in tree\_models:

    if name in results:

        model = results[name]['Model'].named\_steps['model']

        importance = model.feature\_importances\_

        print(f"\nFeature Importance for {name}:")

        feature\_importance = pd.DataFrame({

            'Feature': X.columns,

            'Importance': importance

        })

        print(feature\_importance.sort\_values('Importance', ascending=False))

# Find the best model based on R²

best\_model\_name = comparison\_df.iloc[0]['Model']

best\_model = results[best\_model\_name]['Model']

print(f"\nBest Model: {best\_model\_name} with R² of {results[best\_model\_name]['R²']:.4f}")

# Generate predictions for a full day with the best model

full\_day\_hours = np.arange(0, 24, 0.1).reshape(-1, 1)  # 0.1 hour intervals

full\_day\_predictions = best\_model.predict(full\_day\_hours)

# Plot the best model's predictions for a full day

plt.figure(figsize=(15, 8))

plt.plot(full\_day\_hours, full\_day\_predictions, 'b-', linewidth=2)

plt.scatter(data['Time'], data['Powerperchargingevent'], color='red', alpha=0.3, label='Original Data')

plt.title(f'Best Model ({best\_model\_name}): Predicted Power per Charging Event Throughout the Day')

plt.xlabel('Hour of Day')

plt.ylabel('Power per Charging Event')

plt.legend()

plt.grid(True)

plt.show()

plt.savefig('best\_model\_full\_day.png')

plt.close()

# Calculate charging profile by hour

charging\_profile = pd.DataFrame({

    'Hour': full\_day\_hours.flatten(),

    'Power': full\_day\_predictions

})

# Save the charging profile to a CSV file

charging\_profile.to\_csv('hourly\_charging\_profile.csv', index=False)

print("\nHourly charging profile has been saved to 'hourly\_charging\_profile.csv'")

# Residual analysis for the best model

y\_pred\_best = best\_model.predict(X\_test)

residuals = y\_test - y\_pred\_best

plt.figure(figsize=(12, 6))

plt.scatter(y\_pred\_best, residuals)

plt.axhline(y=0, color='r', linestyle='-')

plt.title(f'Residual Plot for {best\_model\_name}')

plt.xlabel('Predicted Values')

plt.ylabel('Residuals')

plt.grid(True)

plt.show()

plt.savefig('residual\_plot.png')

plt.close()

# Histogram of residuals

plt.figure(figsize=(12, 6))

plt.hist(residuals, bins=30)

plt.title(f'Histogram of Residuals for {best\_model\_name}')

plt.xlabel('Residual Value')

plt.ylabel('Frequency')

plt.grid(True)

plt.show()

plt.savefig('residual\_histogram.png')

plt.close()

# Summary of findings

print("\n=== SUMMARY OF FINDINGS ===")

print(f"Best Model: {best\_model\_name}")

print(f"R² Score: {results[best\_model\_name]['R²']:.4f}")

print(f"RMSE: {results[best\_model\_name]['RMSE']:.4f}")

print(f"MAE: {results[best\_model\_name]['MAE']:.4f}")

print("\nThe predicted charging power profile by hour has been generated and saved.")

print("Check the generated visualizations for more insights.")

11.2. Code of XGBoost training and decision tree generation: link

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import mean\_squared\_error, r2\_score, mean\_absolute\_error

from xgboost import XGBRegressor

# Load the data

data = pd.read\_csv('/content/Final\_data.csv')

# Extract features and target

X = data[['Time']]  # Predictor: Time (hours)

y = data['Powerperchargingevent']  # Target: Power per charging event

# Split data into training (70%) and testing (30%) sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# Initialize and train XGBoost model

best\_model = XGBRegressor(random\_state=42)

best\_model.fit(X\_train, y\_train)

# Make predictions on test set

y\_pred = best\_model.predict(X\_test)

# Evaluate model performance

rmse = np.sqrt(mean\_squared\_error(y\_test, y\_pred))

mae = mean\_absolute\_error(y\_test, y\_pred)

r2 = r2\_score(y\_test, y\_pred)

print("XGBoost Results:")

print(f"RMSE: {rmse:.4f}")

print(f"MAE: {mae:.4f}")

print(f"R²: {r2:.4f}")

# Visualize actual vs predicted values

plt.figure(figsize=(12, 6))

plt.scatter(X\_test, y\_test, color='blue', alpha=0.5, label='Actual Values')

plt.scatter(X\_test, y\_pred, color='red', alpha=0.5, label='Predicted Values')

plt.title('XGBoost: Actual vs Predicted Values')

plt.xlabel('Time (hours)')

plt.ylabel('Power per Charging Event')

plt.legend()

plt.grid(True)

plt.savefig('xgboost\_predictions.png')

plt.close()

# Generate predictions for a full day (0-24 hours)

full\_day\_hours = np.arange(0, 24, 0.1).reshape(-1, 1)

full\_day\_predictions = best\_model.predict(full\_day\_hours)

# Save predictions to CSV

charging\_profile = pd.DataFrame({

    'Hour': full\_day\_hours.flatten(),

    'Power': full\_day\_predictions

})

charging\_profile.to\_csv('xgboost\_hourly\_charging\_profile.csv', index=False)

print("Hourly charging profile saved to 'xgboost\_hourly\_charging\_profile.csv'")

# Plot full-day predictions

plt.figure(figsize=(15, 8))

plt.plot(full\_day\_hours, full\_day\_predictions, 'b-', linewidth=2, label='XGBoost Predictions')

plt.scatter(data['Time'], data['Powerperchargingevent'], color='red', alpha=0.3, label='Original Data')

plt.title('XGBoost: Predicted Power Throughout the Day')

plt.xlabel('Hour of Day')

plt.ylabel('Power per Charging Event')

plt.legend()

plt.grid(True)

plt.savefig('xgboost\_full\_day.png')

plt.close()

# Residual analysis

residuals = y\_test - y\_pred

plt.figure(figsize=(12, 6))

plt.scatter(y\_pred, residuals)

plt.axhline(y=0, color='r', linestyle='-')

plt.title('XGBoost Residual Plot')

plt.xlabel('Predicted Values')

plt.ylabel('Residuals')

plt.grid(True)

plt.savefig('xgboost\_residuals.png')

plt.close()

# Summary

print("\n=== XGBOOST MODEL SUMMARY ===")

print(f"R² Score: {r2:.4f}")

print(f"RMSE: {rmse:.4f}")

print(f"MAE: {mae:.4f}")

import matplotlib.pyplot as plt

from xgboost import plot\_tree

fig, ax = plt.subplots(figsize=(20, 20))  # Create a figure and axes with big size

plot\_tree(best\_model, num\_trees=0, ax=ax)  # Pass the ax to plot\_tree

plt.show()

rules = best\_model.get\_booster().get\_dump()

print(rules[0])  # Print rules of the first tree

11.3 MATLAB:

11.3.1 predict\_power.m (MATLAB Function) :

function power = predict\_power(time)

if time < 15.72

if time < 11.72

if time < 8.68

power = -3.07087541;

else

power = -2.26829529;

end

else

if time < 13.64

if time < 13.14

power = -1.54146445;

else

power = -1.02141774;

end

else

if time < 14.7799997

if time < 14.18

power = -0.562602878;

else

if time < 14.59

power = -0.293976963;

else

power = -0.148339242;

end

end

else

if time < 15.09

if time < 15.0

power = -0.089276582;

else

power = 0.00799224433;

end

else

if time < 15.1899996

power = 0.114325292;

else

power = 0.281705678;

end

end

end

end

end

else

if time < 20.1399994

if time < 17.8700008

if time < 16.5900002

if time < 16.2700005

power = 0.468501061;

else

power = 0.766681671;

end

else

power = 1.29236221;

end

else

if time < 18.68

power = 1.97458136;

else

power = 2.6394906;

end

end

else

if time < 21.5499992

power = 3.57565904;

else

power = 4.73966026;

end

end

end

end

11.3.2 MATLAB main file:

% Ensure predict\_power.m is in the same directory or in MATLAB's path

% Generate time values from 0 to 24 hours

time\_values = 0:0.1:24;

power\_values = arrayfun(@predict\_power, time\_values); % Calls function from predict\_power.m

% Plot results

figure;

plot(time\_values, power\_values, 'b-', 'LineWidth', 2);

xlabel('Time (Hours)');

ylabel('Predicted Power');

title('Power Prediction Over a Day');

grid on;

% Generate a simulated DAC signal (Discrete values)

figure;

stem(time\_values, power\_values, 'filled');

xlabel('Time (Hours)');

ylabel('Power (DAC Output)');

title('Simulated DAC Output for STM32');

grid on;

codegen predict\_power -args 0

11.4 CCS code:

**13. References:**

1. J. Liu, Z. Ma, and Y. Liu, "Prediction of Electric Vehicle Charging Load Based on Machine Learning Methods," IEEE Access, vol. 8, pp. 68455–68466, 2020. DOI: 10.1109/ACCESS.2020.2986701.
2. T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 785–794, 2016. DOI: 10.1145/2939672.2939785..
3. A. Ghofrani, M. S. Islam, G. Ali, and Z. Yang, "Smart Charging for Electric Vehicle Aggregators Considering Renewable Energy Integration," *IEEE Transactions on Smart Grid*, vol. 10, no. 4, pp. 4119–4129, July 2019. DOI: 10.1109/TSG.2018.2879657.
4. S. B. Peterson, J. Whitacre, and J. Apt, "The Economics of Using Plug-in Hybrid Electric Vehicle Battery Packs for Grid Storage," *Journal of Power Sources*, vol. 195, no. 8, pp. 2377–2384, 2010. DOI: 10.1016/j.jpowsour.2009.08.049.
5. E. Moradi and M. A. Golkar, "Electric Vehicles Charging Scheduling and Optimal Load Forecasting Using Machine Learning," *IEEE Transactions on Sustainable Energy*, vol. 13, no. 3, pp. 1715–1725, July 2022. DOI: 10.1109/TSTE.2022.3148982.
6. M. Asif and A. Gawanmeh, "Real-Time Electric Vehicle Charging Load Forecasting Using Machine Learning," *IEEE Access*, vol. 9, pp. 53100–53109, 2021. DOI: 10.1109/ACCESS.2021.3072712.
7. F. Wang, C. Li, J. Wang, et al., "Smart Charging for Electric Vehicles: A Survey from the Algorithmic Perspective," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 1, pp. 656–685, 2019. DOI: 10.1109/COMST.2018.2883143.