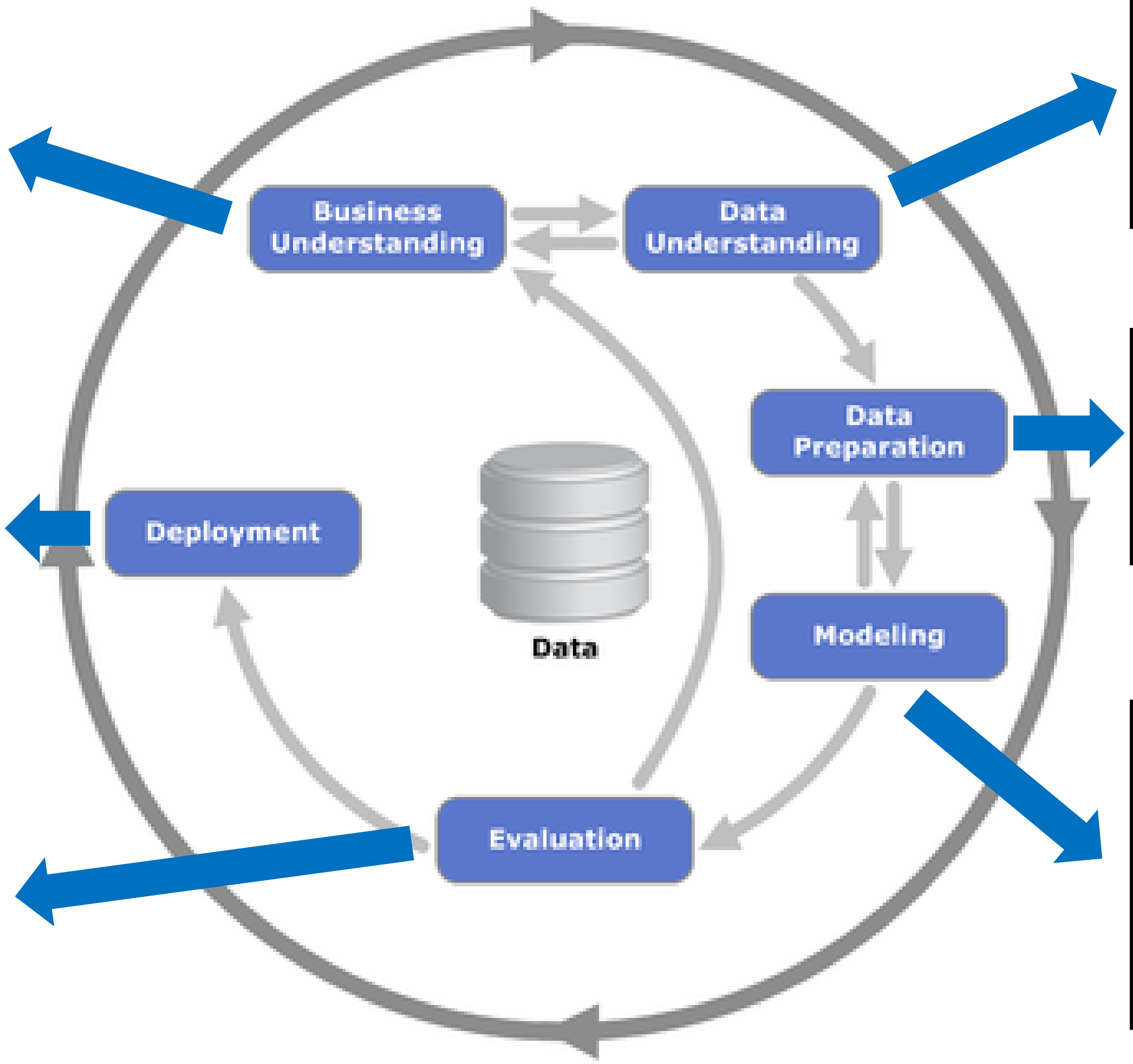
State of Charge (SOC) Estimation For LiFePO4 (LFP) Battery ZTE ZXDC48100C1

Methodology: Cross Industry Standard Process for Data Mining (CRISP – DM)



Business Understanding
- Research Objective
- Success criteria (or criteria to achieve)

Data Understanding
- Object Research and Testing Device
- Data Collection Methodology
- Exploratory Data Analysis (EDA)

Deployment
- Model Implementation
- Integration with the system
- Trials

Data Preparation
- Data Cleaning
- Data Transformation
- Feature Engineering
- Data Splitting

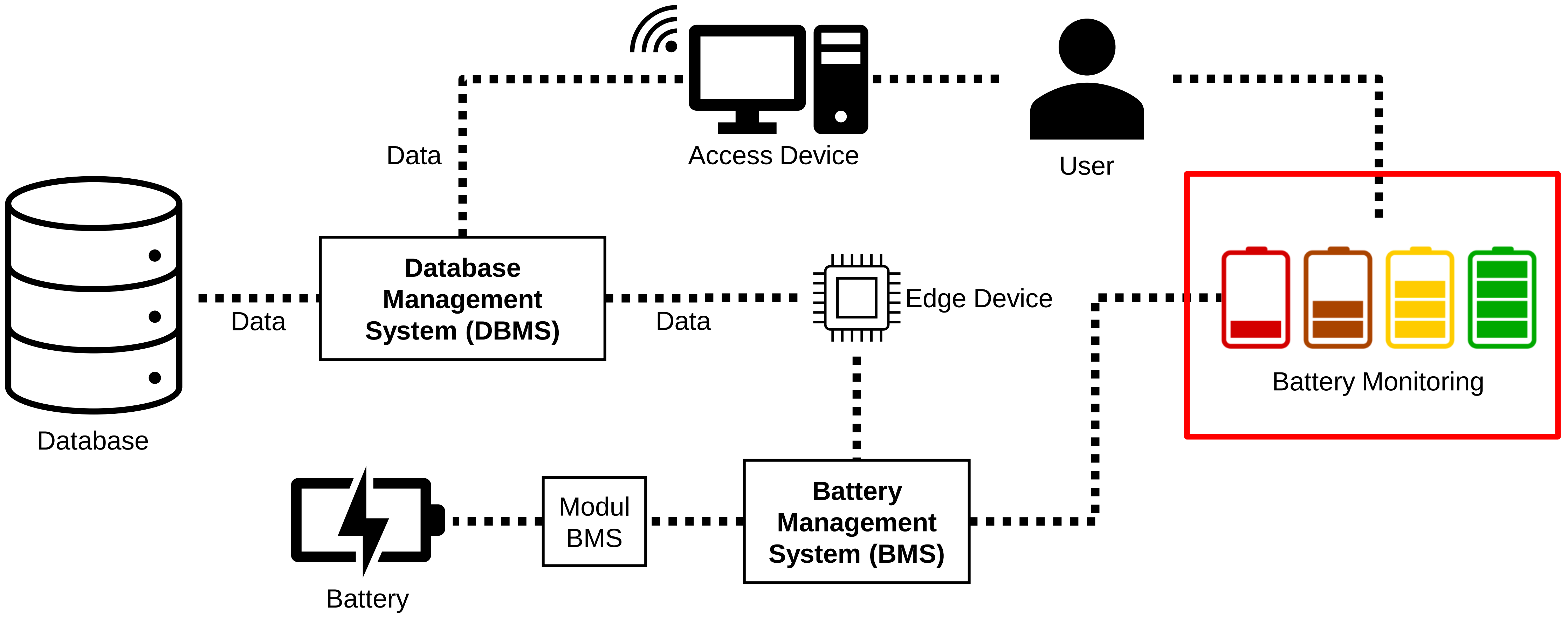
Evaluation
- Model Performance Evaluation
- Cross-Vlidation
- Conclusion

Modeling
- Model Selection
- Model Tuning (parameters and hyperparameters)
- Model Evaluation
- Model Validation
- Model Documentation

Business Understanding

Research Objective

- Building a ZXDC48 lithium-ion battery State of Charge (SOC) estimation model for Battery Management System (BMS).
- Create a BMS to monitor battery operational conditions.
- Deploy the SOC model prediction to the BMS using Edge Device.
- Sending operational data to the Database Management System (DBMS).



Data Understanding

Object Research

Battery Specification

Battery Type	Lithium Iron Phospate (LiFePO4)
Model	ZXDC48 100C1
Manufacture	ZTE
Capacity	100 Ah
Voltage	48 V
Weight	≤ 50

Operational Performance Parameters

Discharging voltage	**42 V to 53.2 V**
Discharging current	0 A to 50 A
Charging voltage	**52 V to 54 V**
Charging current	0 A to 50 A

Operational Environment Parameters

Recommanded operating temperature	15 °C to 35 °C
Charging temperature	0 °C to 60 °C
Discharging temperature	-20 °C to 65 °C
Storage temperature	-40 °C to 60 °C
Relative humidity	5% to 95%
Altitude	4000 m
Air pressure	70 kPa to 100 kPa

Data Understanding

Testing Device

Programmable Power Supply (for charging)

Specification

Manufacture	BK Precision
Model	9104 Series
Power	320 W
Operating Voltage	0 – 84 V
Rated Current	0 – 10 A

Electronic Load Controller (for discharging)

Image represents Models 8610, 8612, 8614, 8616

Specification

Manufacture	BK Precision
Model	8600 Series (8614)
Power	1500 W
Operating Voltage	0 – 120 V
Rated Current	0 – 240 A

Data Understanding

Data Collection Methodology

In this study, we conducted a Battery Cycle Test method consisting of charging and discharging test scenarios.

Charging

Constant Voltage (CV) : a method where the voltage remains constant throughout the charging process. In this method, once the battery reaches a predetermined voltage level, the charging current decreases gradually until it reaches zero, allowing the battery to be fully charged while preventing overcharging.

Scenario	**Charging**
Constant Voltage (CV)	54 V
Testing Temperature	+- 25 °C
Current	5 A
Initial Voltage	+- 44 V
Maximum Voltage	54 V

Discharging

Constant Current (CC) : method where a load is supplied with a steady current throughout the discharging process. In this method, the current flowing through the load remains constant, regardless of changes in voltage or load resistance.

Scenario	**Discharging**
Constant Current (CC)	5 A, 8 A, 10 A
Testing Temperature	+- 25 °C
Initial Voltage	+- 52 V
Cut-off Voltage (Minimum Voltage)	42 V (44 V for safety based on internal BMS battery device)

Data Understanding

Exploratory Data Analysis

Element	Charging Data	Discharging Data
Number of features	2	7
Features	Voltage, Current	Time, Voltage, Current, Power, Capacity, Energy Density, Power Density
Number of cycles	3 cycle	7 cycle
Cycle scenarios	3 cycles CV 54 V	3 cycles C/20 (CC 5 A), 3 cycles C/10 (CC 10 A), 1 cycles C/12.5 (CC 8 A)
Length of data rows	54003	372812
Size of data	1.6+ MB	25.6 MB
Format data	CSV	CSV
Null values	None	None
NaN values	None	None

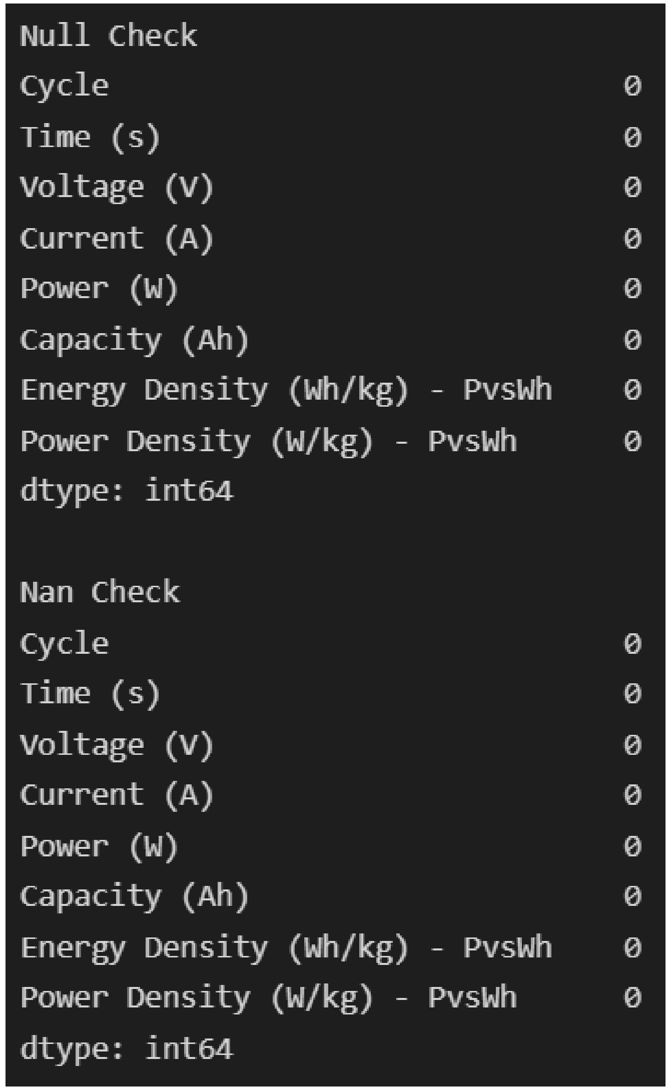
Data Understanding

Exploratory Data Analysis: Missing Values Detection

We use this code to check **Missing Values** with Python

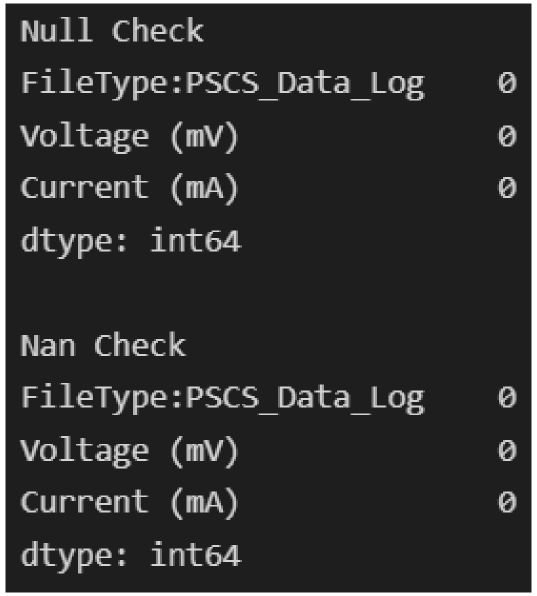
```python
# Import Python Library
import pandas as pd

# Use Function
null_check = pd.isnull().sum()
nan_check = pd.isna().sum()

# Execute the Function
print("Null Check")
print(null_check)
print()
print("Nan Check")
print(nan_check)
```

Dataset Discharge



```
Null Check
Cycle                              0
Time (s)                           0
Voltage (V)                        0
Current (A)                        0
Power (W)                          0
Capacity (Ah)                      0
Energy Density (Wh/kg) - PvsWh     0
Power Density (W/kg) - PvsWh       0
dtype: int64

Nan Check
Cycle                              0
Time (s)                           0
Voltage (V)                        0
Current (A)                        0
Power (W)                          0
Capacity (Ah)                      0
Energy Density (Wh/kg) - PvsWh     0
Power Density (W/kg) - PvsWh       0
dtype: int64
```

Dataset Charge



```
Null Check
FileType:PSCS_Data_Log    0
Voltage (mV)              0
Current (mA)              0
dtype: int64

Nan Check
FileType:PSCS_Data_Log    0
Voltage (mV)              0
Current (mA)              0
dtype: int64
```

Data Understanding

Exploratory Data Analysis: Descriptive Statistics

Dataset Discharge

	Time (s)	Voltage (V)	Current (A)	Power (W)	Capacity (Ah)	Energy Density (Wh/kg) - PvsWh	Power Density (W/kg) - PvsWh
count	372812	372812	372812	372812	372812	372812	372812
mean	29360.522269	48.352966	6.830148	329.763447	50.498084	2463.811779	329.763447
std	19511.419573	0.919486	2.252673	107.314736	29.155364	1412.818368	107.314736
min	1.00000	0.00000	0.00000	0.00000	0.00100	0.07075	0.00000
25%	13315	48.0164	4.99718	243.069	25.24875	1242.167500	243.069
50%	26630	48.5813	4.99823	246.556	50.49800	2471.175000	246.556
75%	42583	48.8845	9.99896	471.344	75.74700	3692.265000	471.344
max	72733	52.1700	10.0000	517.150	101.2920	4901.420000	517.150

We use this code to check **Descriptive Statistics** with Python

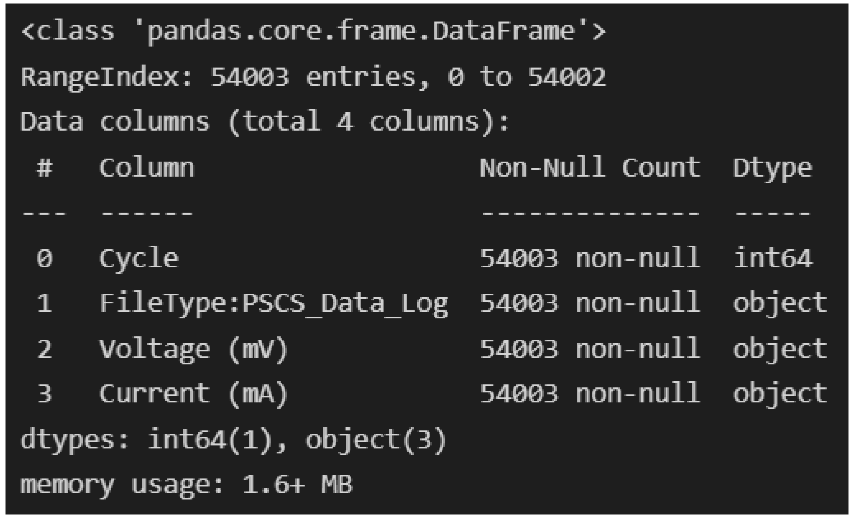
```
# Use describe() function to check descriptive statistics
statistics = data.describe()

# Print the descriptive statistics
print(statistics)
```

Data Understanding

Exploratory Data Analysis: Descriptive Statistics

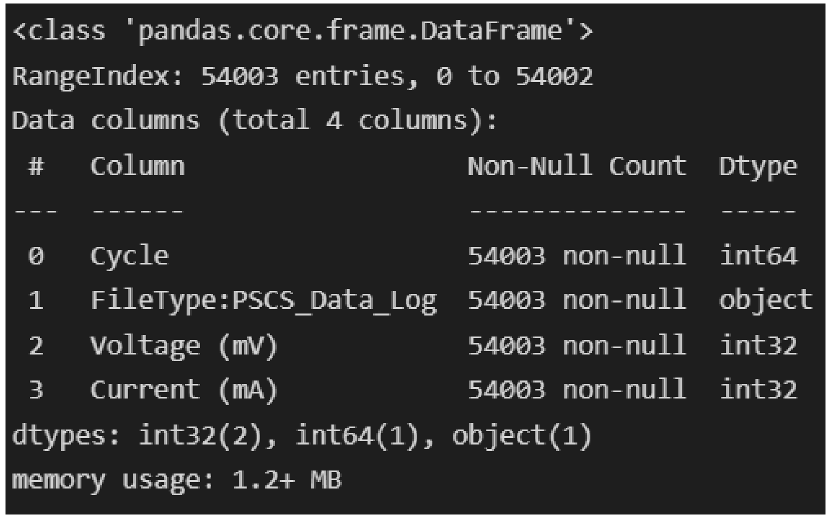
Dataset Charge

Before converting



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 54003 entries, 0 to 54002
Data columns (total 4 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   Cycle                  54003 non-null  int64
 1   FileType:PSCS_Data_Log 54003 non-null  object
 2   Voltage (mV)           54003 non-null  object
 3   Current (mA)           54003 non-null  object
dtypes: int64(1), object(3)
memory usage: 1.6+ MB
```

After converting



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 54003 entries, 0 to 54002
Data columns (total 4 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   Cycle                  54003 non-null  int64
 1   FileType:PSCS_Data_Log 54003 non-null  object
 2   Voltage (mV)           54003 non-null  int32
 3   Current (mA)           54003 non-null  int32
dtypes: int32(2), int64(1), object(1)
memory usage: 1.2+ MB
```

Since the **voltage** and **current** data types in dataset are objects, we need to convert the data dtype to integer by using this python code

```
# Convert column from object to integer
data['Voltage (mV)'] = data['Voltage (mV)'].astype(int)
data['Current (mA)'] = data['Current (mA)'].astype(int)
```

Data Understanding

Exploratory Data Analysis: Descriptive Statistics

Dataset Charge

	Voltage (mV)	Current (mA)
count	54003.000000	54003.000000
mean	52844.684555	2036.161324
std	1472.110331	2458.568872
min	50740.000000	0.000000
25%	51020.000000	0.000000
50%	54020.000000	10.000000
75%	54020.000000	5030.000000
max	54050.000000	5030.000000

We use this code to check **Descriptive Statistics** with Python

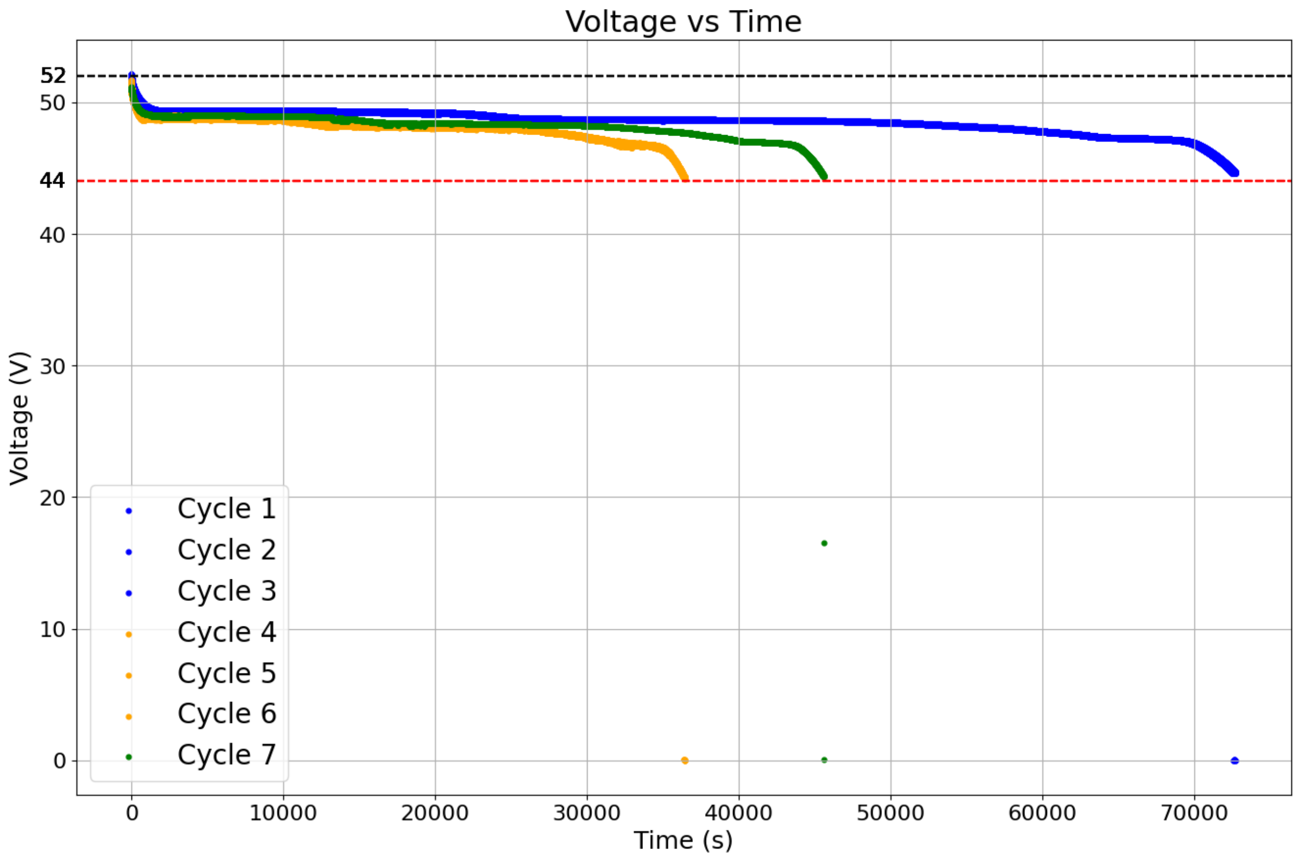
```
# Use describe() function to check descriptive statistics
statistics = data.describe()

# Print the descriptive statistics
print(statistics)
```

Data Understanding

Exploratory Data Analysis: Data Visualization

We use this code to visualize data with Python

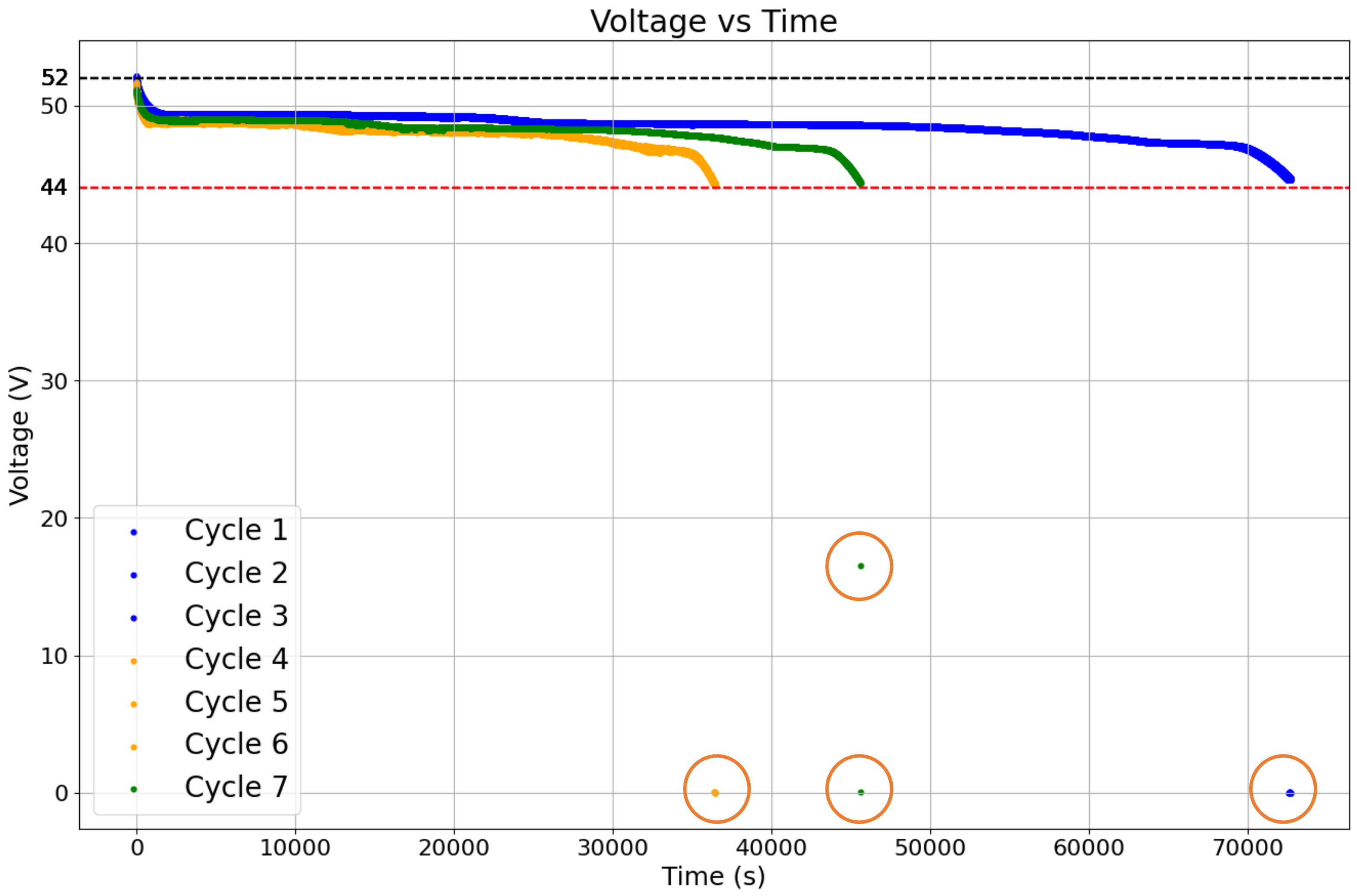
```
# To separate the plots of each cycle
dfs = []
for i in range(1, 8):
    data = df[df['Cycle'] == i]

    dfs.append(data)

# To show the graph
plt.figure(figsize = (16,10))

# Skema warna untuk setiap siklus
colors = ['blue', 'blue', 'blue', 'orange', 'orange', 'orange', 'green']

# Plotting the graph
for i, data in enumerate(dfs):
    plt.title('Voltage vs Time', fontsize = 20)
    x = data['Time (s)']
    y = data['Voltage (V)']
    plt.scatter(x, y, color=colors[i], label=f'Cycle {i+1}', s=12)
    plt.xticks(fontsize = 16)
    plt.yticks(fontsize = 16)
    plt.axhline(y=44, color='red', linestyle='--')
    plt.axhline(y=52, color='black', linestyle='--')
    plt.text(-4250, 44, '44', color='black', fontsize=16, ha='right', va='center')
    plt.text(-4250, 52, '52', color='black', fontsize=16, ha='right', va='center')
    plt.xlabel("Time (s)", fontsize = 18)
    plt.ylabel("Voltage (V)", fontsize = 18)
    plt.grid()
    plt.legend()

plt.show()
```



Data Understanding

Exploratory Data Analysis: Data Visualization

The following is a graph plotting voltage against time on the discharging dataset.



- The test results for each scenario are separated by color as shown in the following table.

Color	Cycle	Scenario
Blue	1, 2, 3	CC 10 A
Green	7	CC 8 A
Blue	4, 5, 6	CC 5 A

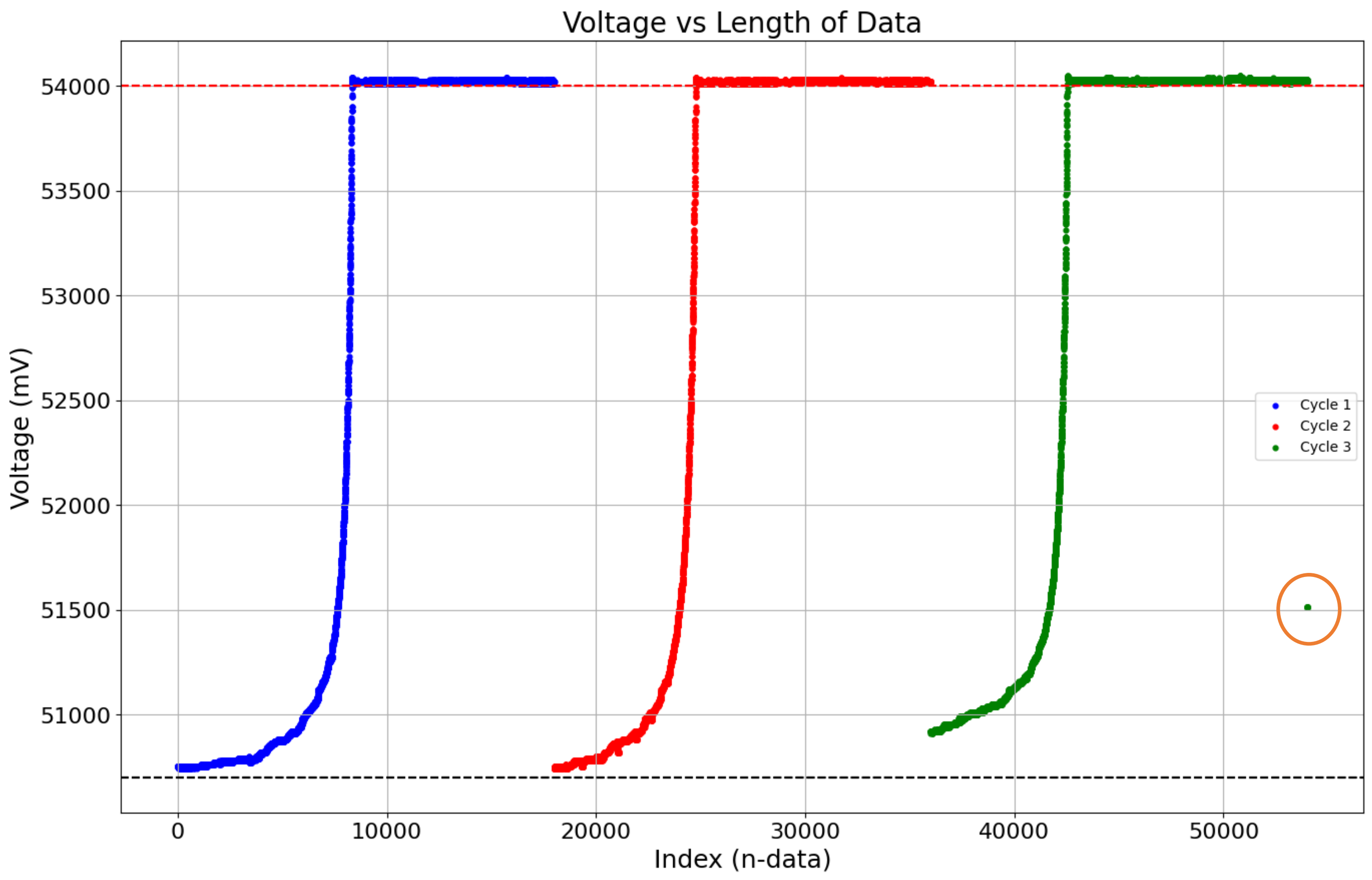
CC : Constant Current

- It can be seen in the graph that the test scenario is in the voltage range of 44 V to 52 V.
- There are **outliers** in the voltage range 0 to 20 V.

Data Understanding

Exploratory Data Analysis: Data Visualization

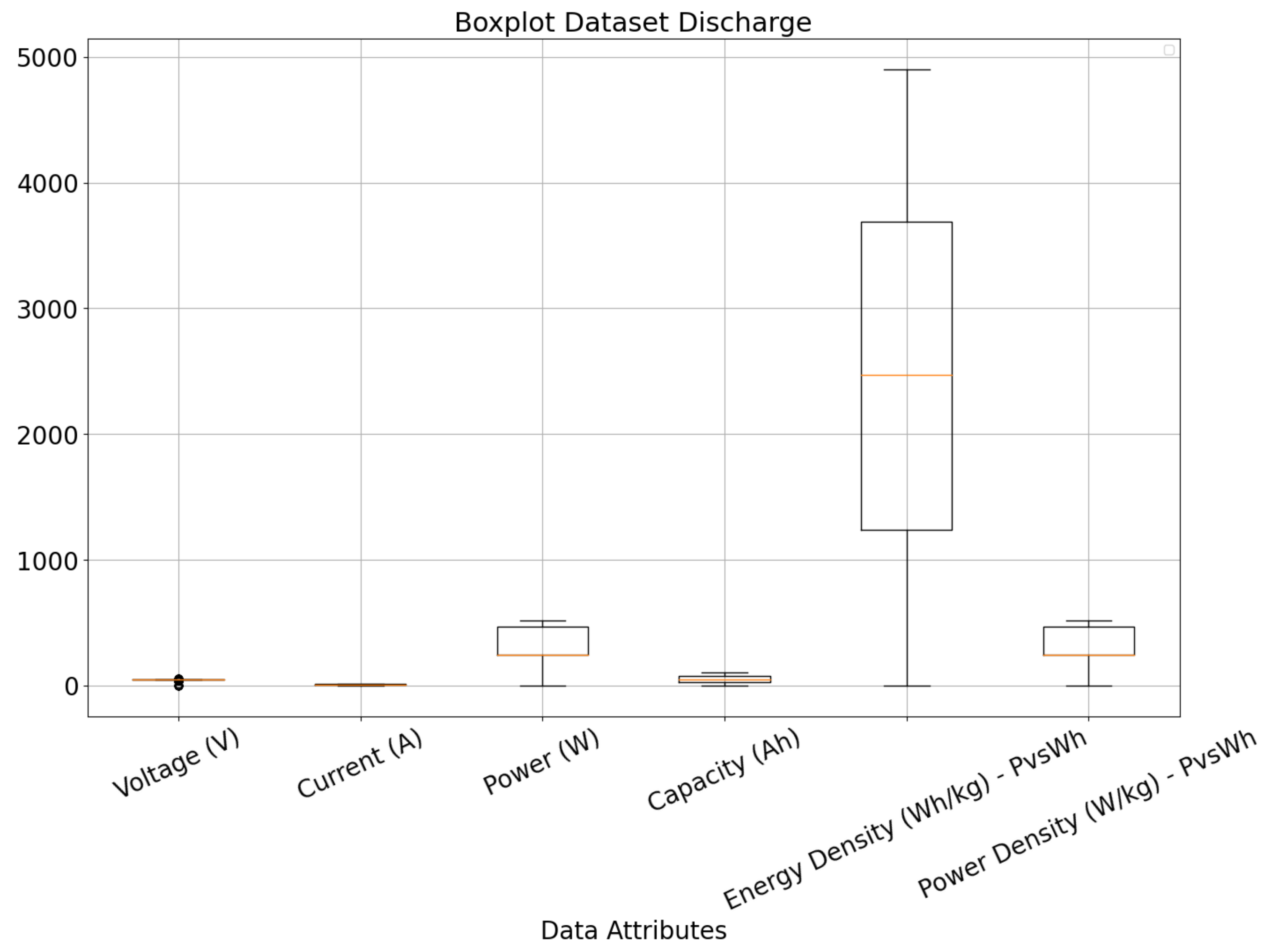
The following is a graph plotting voltage against time on the charging dataset.

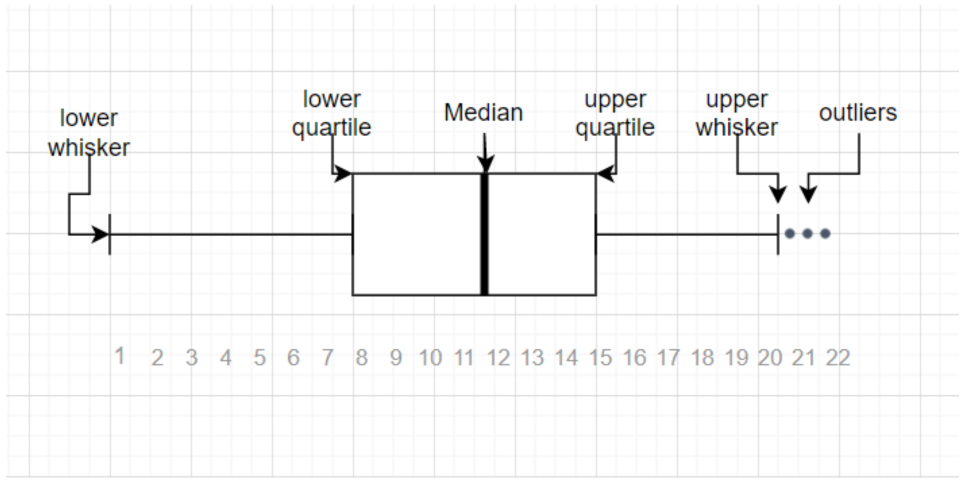


- The test results for each scenario are separated by color as shown in the following table.
- It can be seen in the graph that the test scenario is in the voltage range of **50 V to 54 V**.
- There are **outliers** in the cycle 3 with voltage range 51.5 V to 52 V.

Data Understanding

Exploratory Data Analysis: Outliers Identification with Boxplot

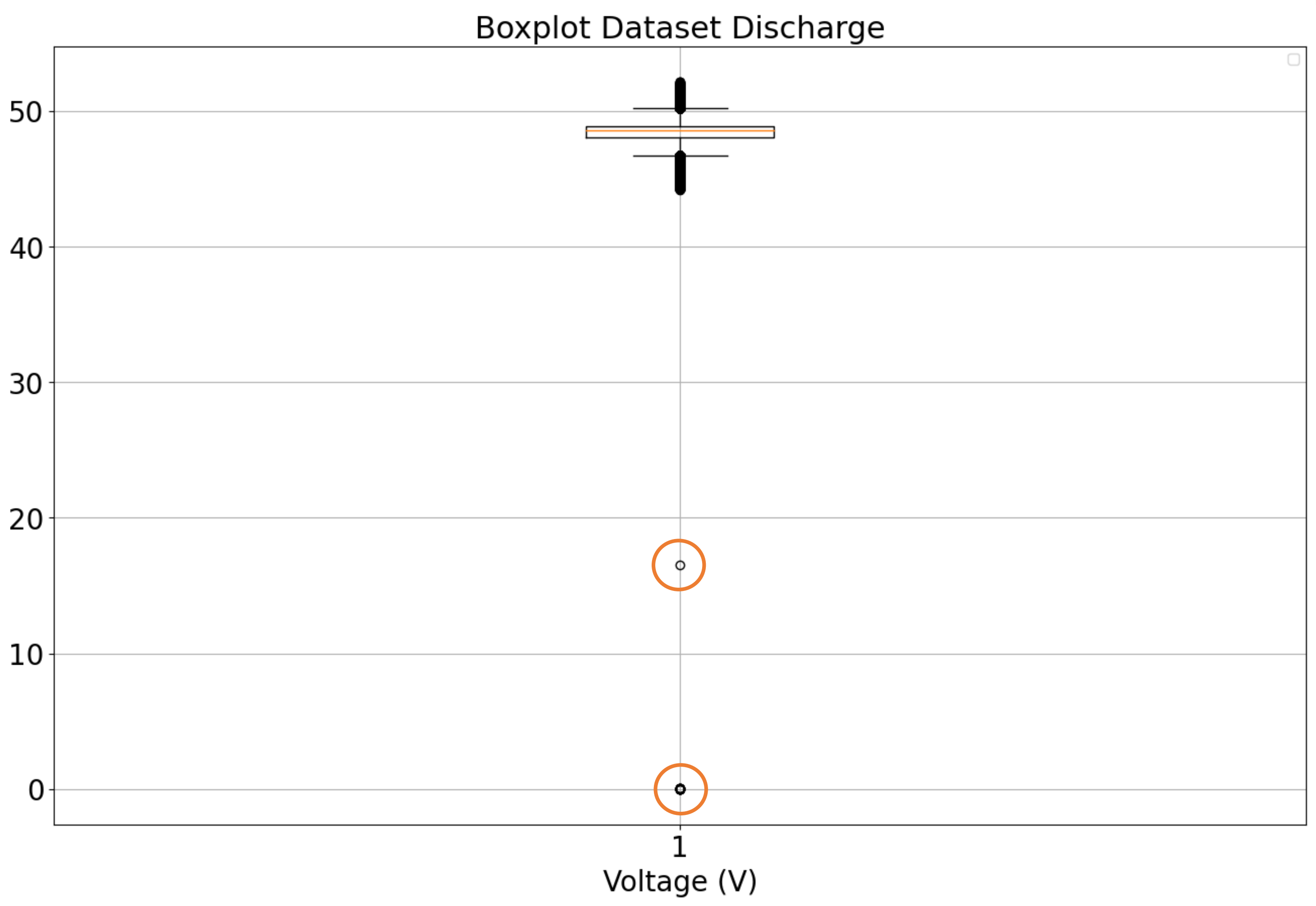


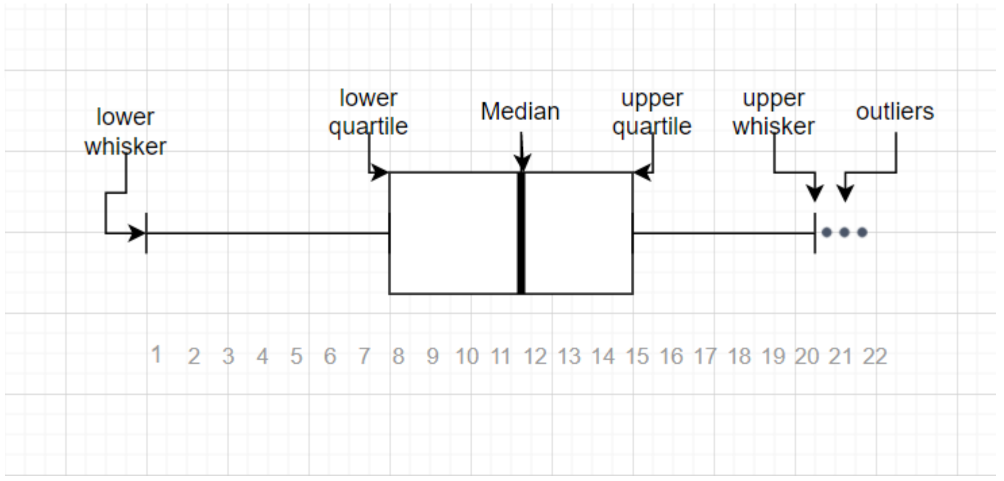


Seems like there are no **outliers** detected in boxplot dataset discharge, but we need to check Voltage (V) feature using quartile.

Data Understanding

Exploratory Data Analysis: Outliers Identification with Boxplot





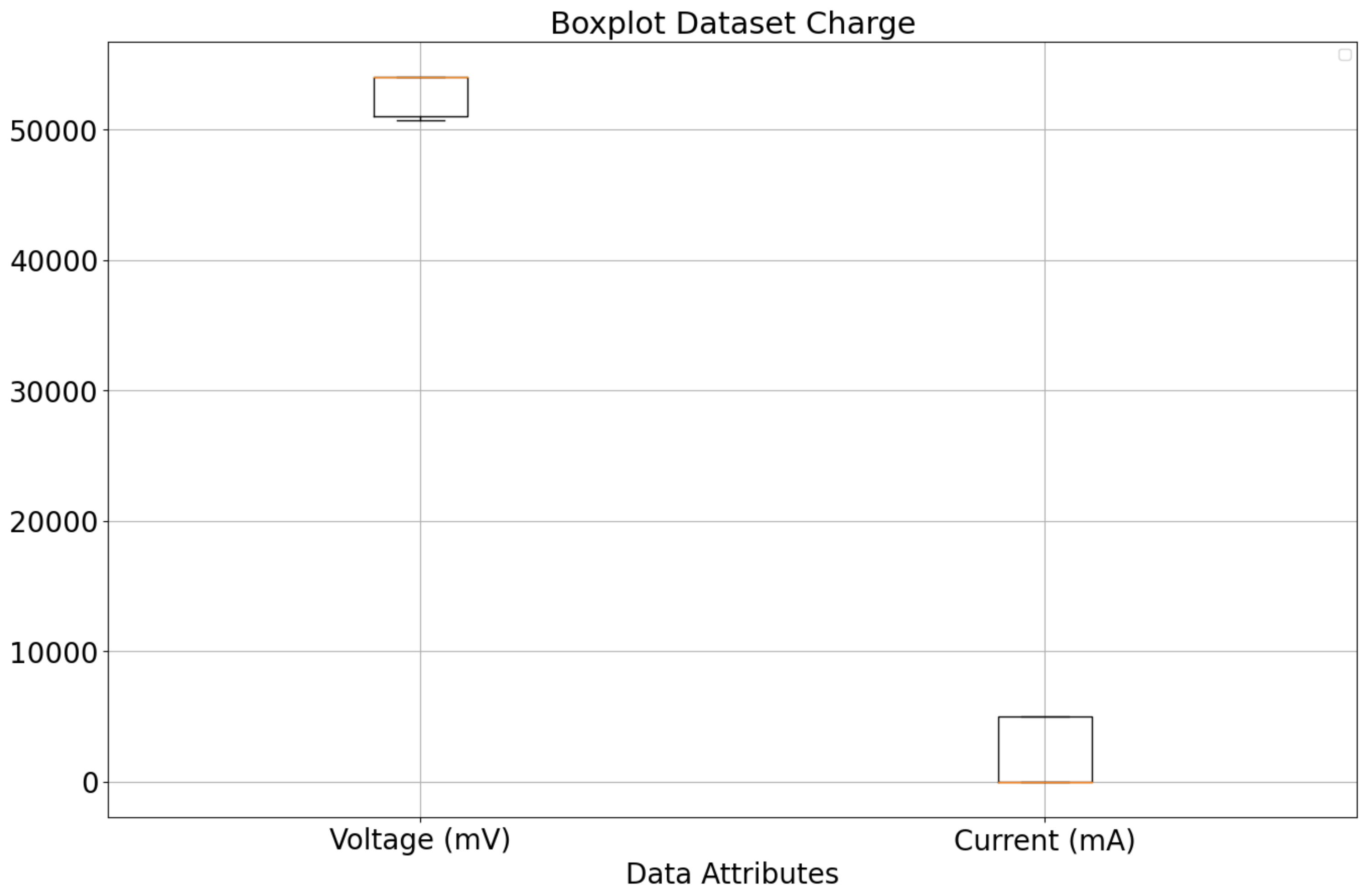
Using **MinMaxScaler()** we can calculate Inter Quartile, Lower Quartile, and Upper Quartile with Python, We get

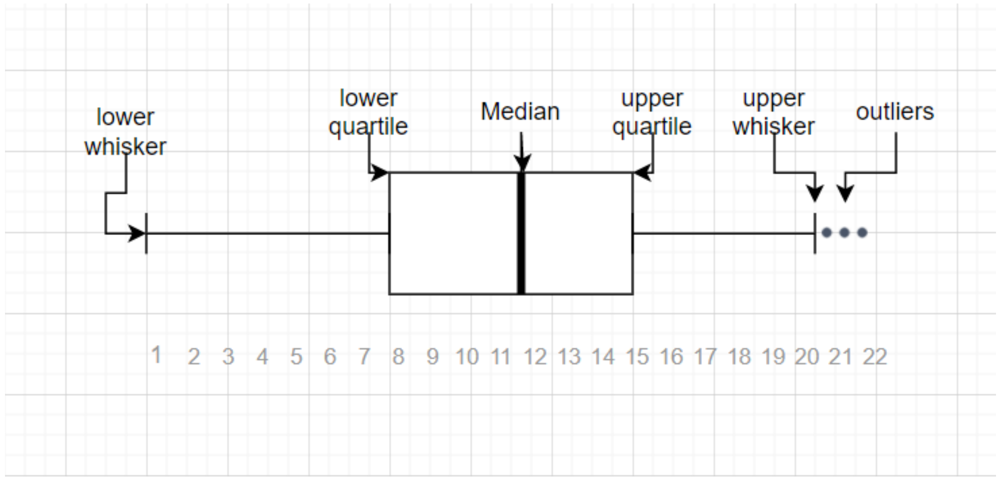
Inter Quartile	0.8681000000000054
Median	48.5813
Upper	50.18665000000014
Lower	46.714249999999999

But in this case, we use **52 V** as Upper and **44 V** as Lower

Data Understanding

Exploratory Data Analysis: Outliers Identification with Boxplot

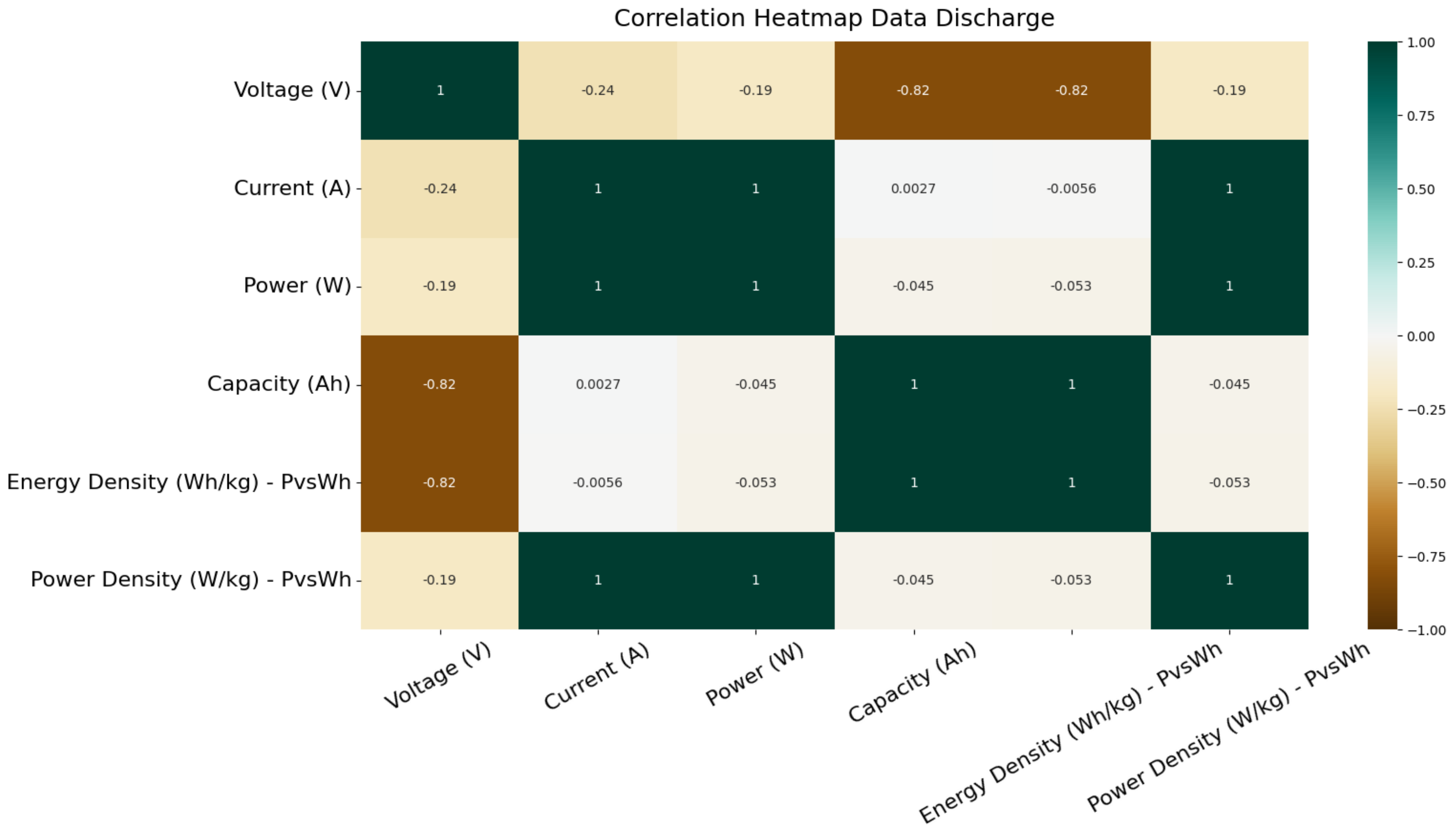




There are no **outliers** detected in boxplot dataset charge

Data Understanding

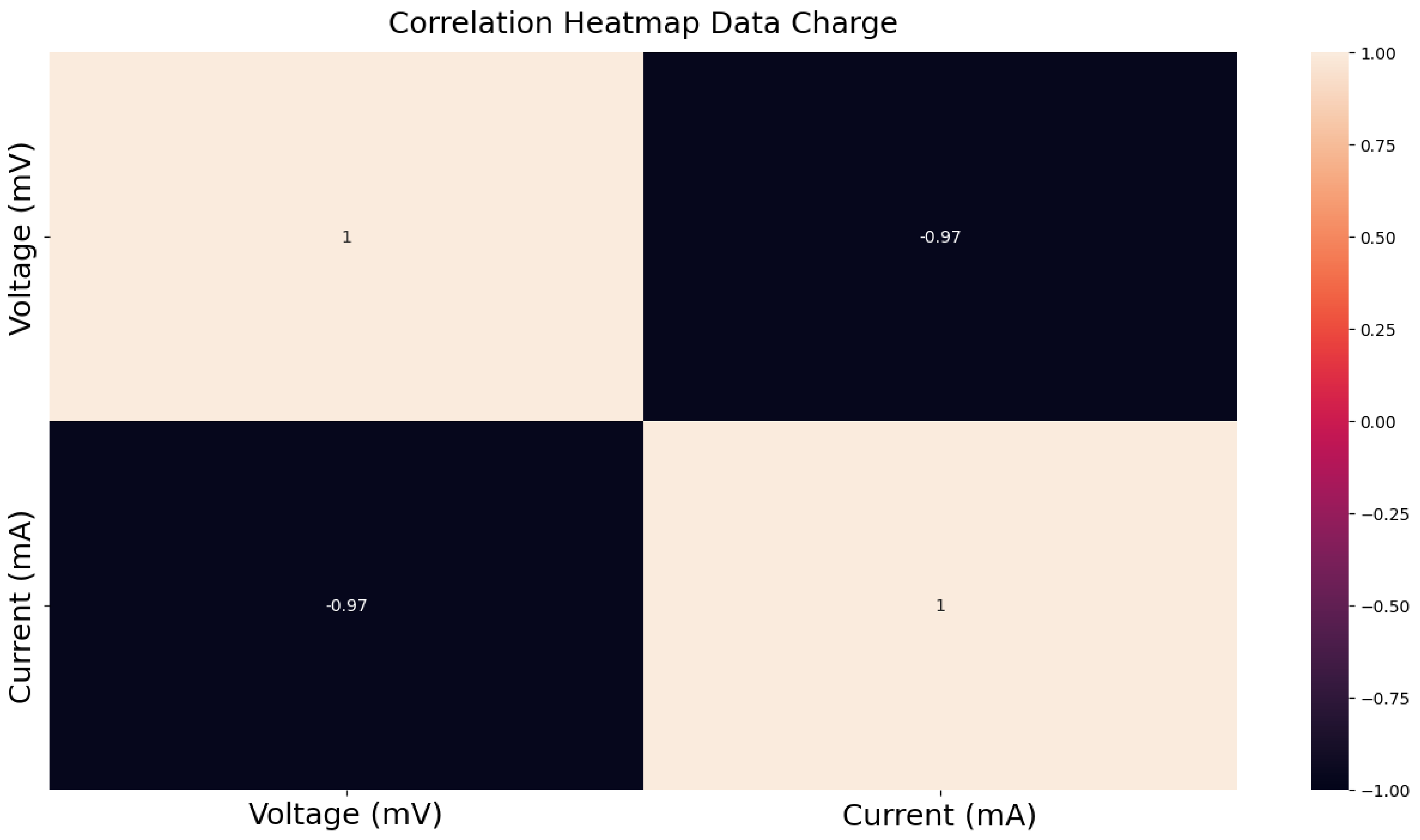
Exploratory Data Analysis: Correlation Analysis with Seaborn



	Voltage (V)	Current (A)	Power (W)	Capacity (Ah)	Energy Density (Wh/kg) - PvsWh	Power Density (W/kg) - PvsWh
Voltage (V)	1	-0.24	-0.19	-0.82	-0.82	-0.19
Current (A)	-0.24	1	1	0.0027	-0.0056	1
Power (W)	-0.19	1	1	-0.045	-0.053	1
Capacity (Ah)	-0.82	0.0027	-0.045	1	1	-0.045
Energy Density (Wh/kg) - PvsWh	-0.82	-0.0056	-0.053	1	1	-0.053
Power Density (W/kg) - PvsWh	-0.19	1	1	-0.045	-0.053	1

Data Understanding

Exploratory Data Analysis: Correlation Analysis with Seaborn

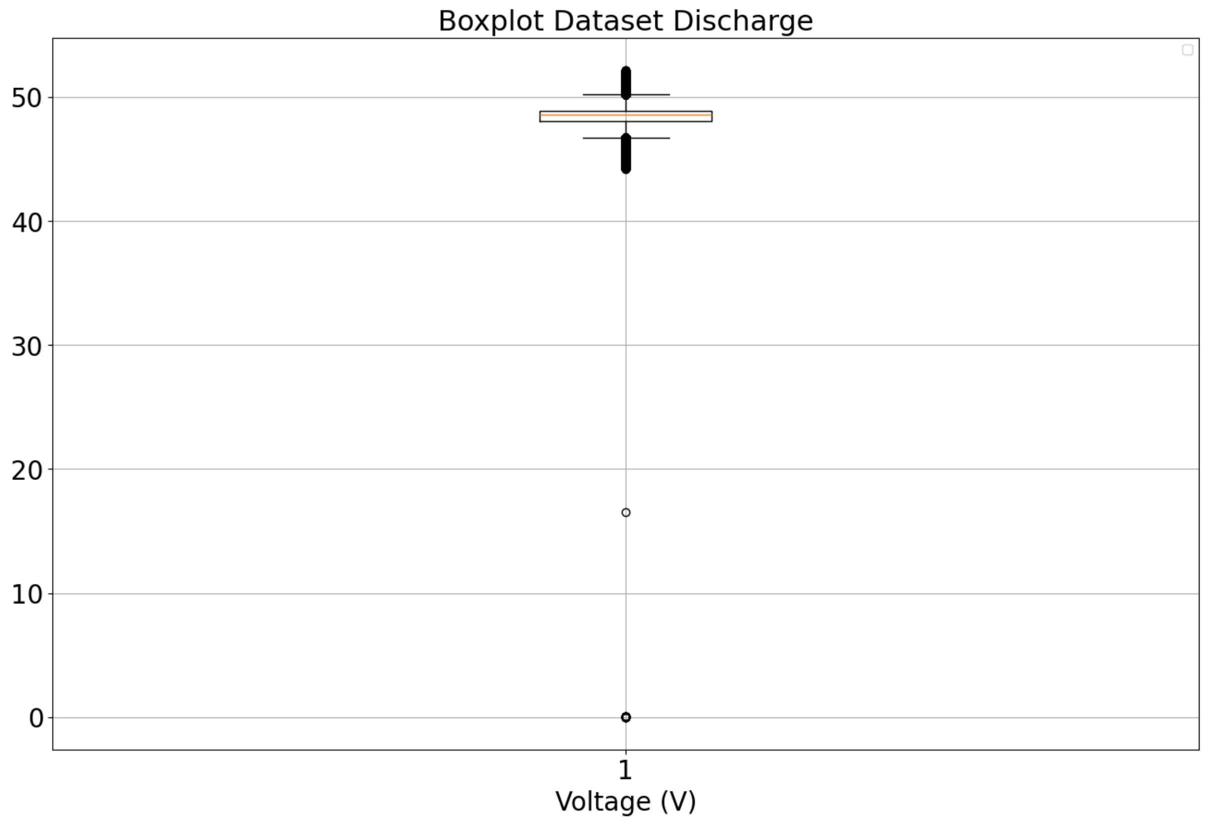


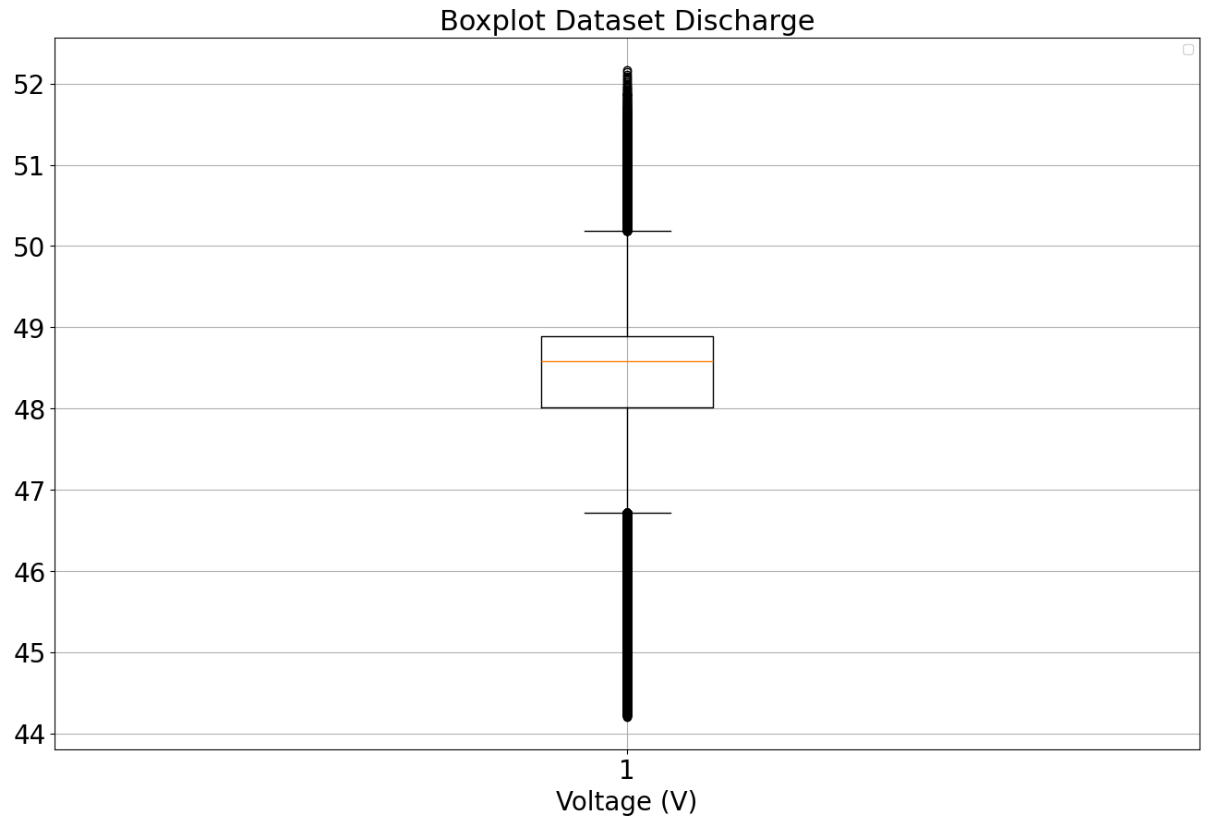
Data Preparation

Exploratory Data Analysis: Cleaning Outlier

To clean the outliers, we use this Python code:

```
df = df[~(df['Voltage (V)'] < 44)]
```



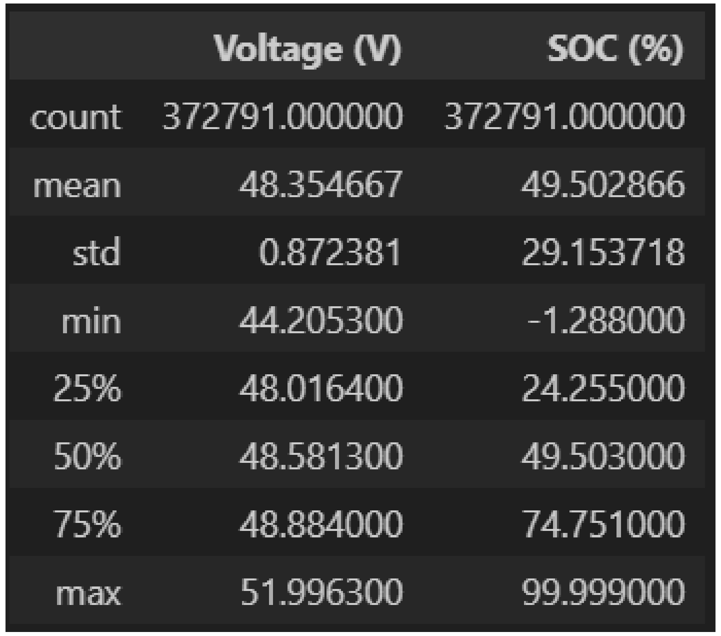


Data Preparation

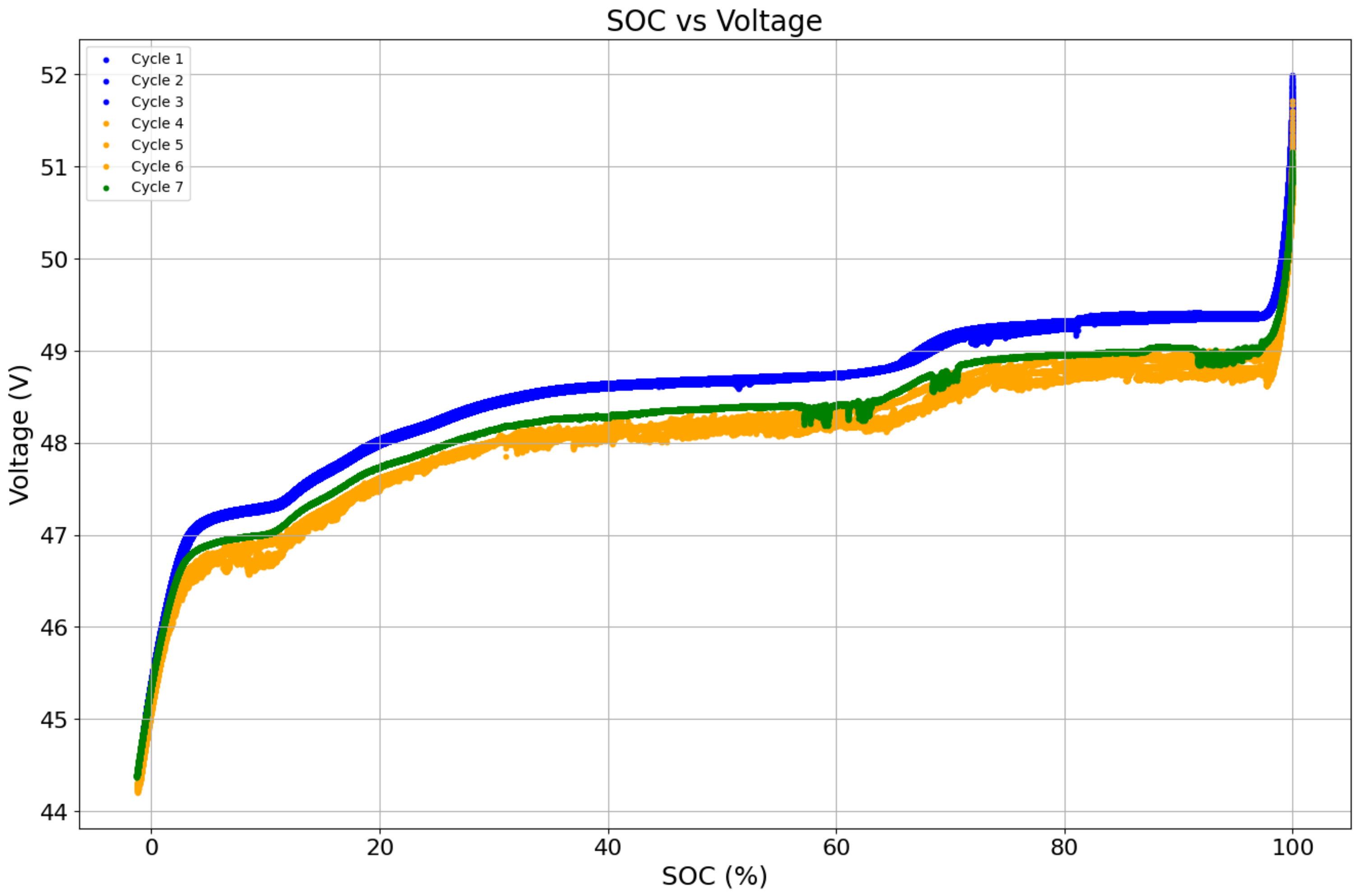
Feature Engineering: Create SOC references as Dependent Data

To clean the outliers, we use this Python code:

$$\mathrm{SOC(t)} = \mathrm{SOC}(t_0) + \frac{1}{C}\sum_{i=0}^{n} I_i \Delta t$$

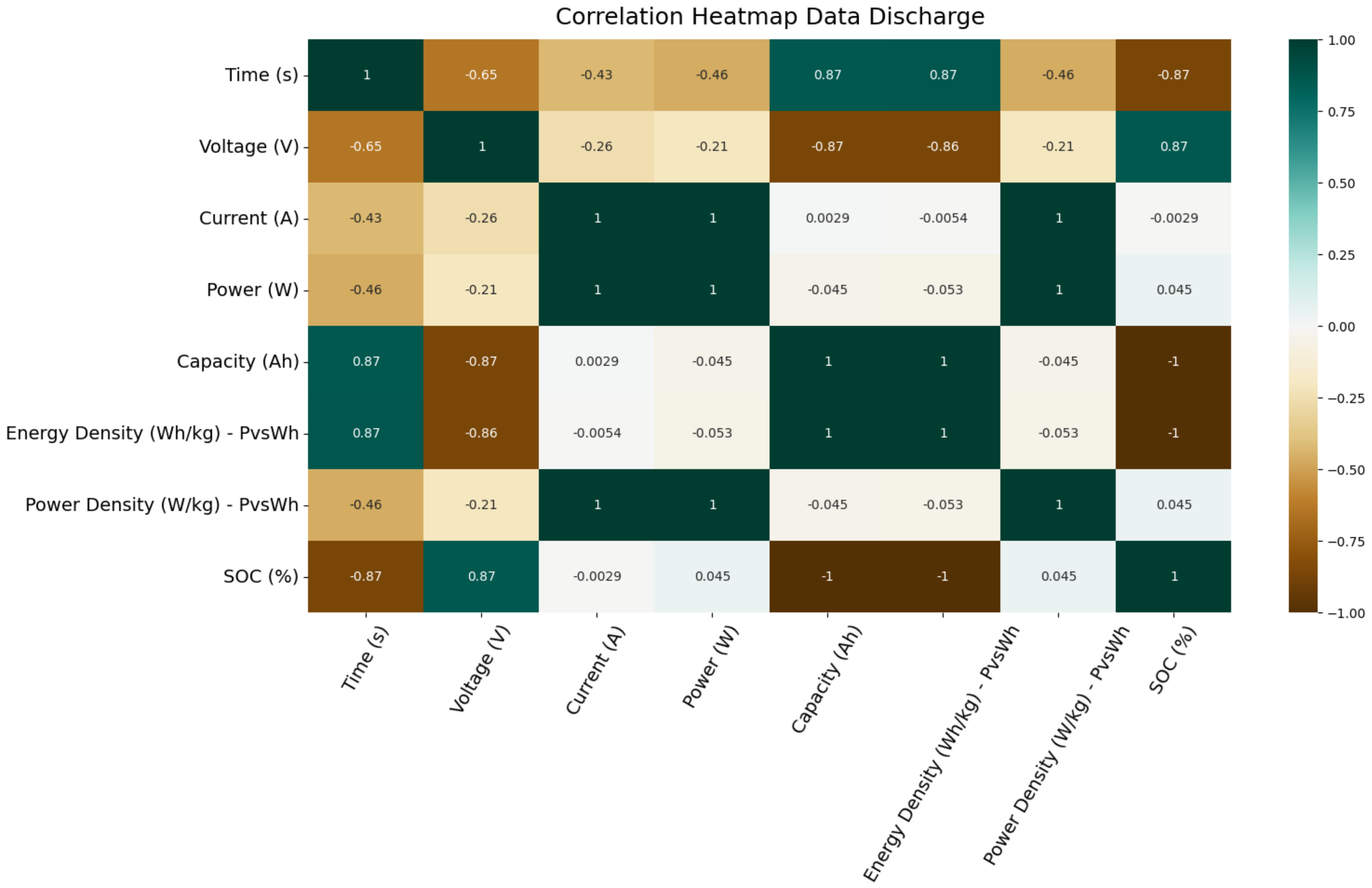


	Voltage (V)	SOC (%)
count	372791.000000	372791.000000
mean	48.354667	49.502866
std	0.872381	29.153718
min	44.205300	-1.288000
25%	48.016400	24.255000
50%	48.581300	49.503000
75%	48.884000	74.751000
max	51.996300	99.999000



Data Preparation

Feature Selection: Correlation Analysis with Seaborn



	Time (s)	Voltage (V)	Current (A)	Power (W)	Capacity (Ah)	Energy Density (Wh/kg) - PvsWh	Power Density (W/kg) - PvsWh	SOC (%)
Time (s)	1	-0.65	-0.43	-0.46	0.87	0.87	-0.46	-0.87
Voltage (V)	-0.65	1	-0.26	-0.21	-0.87	-0.86	-0.21	0.87
Current (A)	-0.43	-0.26	1	1	0.0029	-0.0054	1	-0.0029
Power (W)	-0.46	-0.21	1	1	-0.045	-0.053	1	0.045
Capacity (Ah)	0.87	-0.87	0.0029	-0.045	1	1	-0.045	-1
Energy Density (Wh/kg) - PvsWh	0.87	-0.86	-0.0054	-0.053	1	1	-0.053	-1
Power Density (W/kg) - PvsWh	-0.46	-0.21	1	1	-0.045	-0.053	1	0.045
SOC (%)	-0.87	0.87	-0.0029	0.045	-1	-1	0.045	1

0 – 0.25: Very weak correlation
0.25 – 0.5: Fair correlation
0.5 – 0.75: Strong correlation
0.75 – 0.99: Very strong correlation
1: Perfect positive correlation
-1: Perfect negative correlation

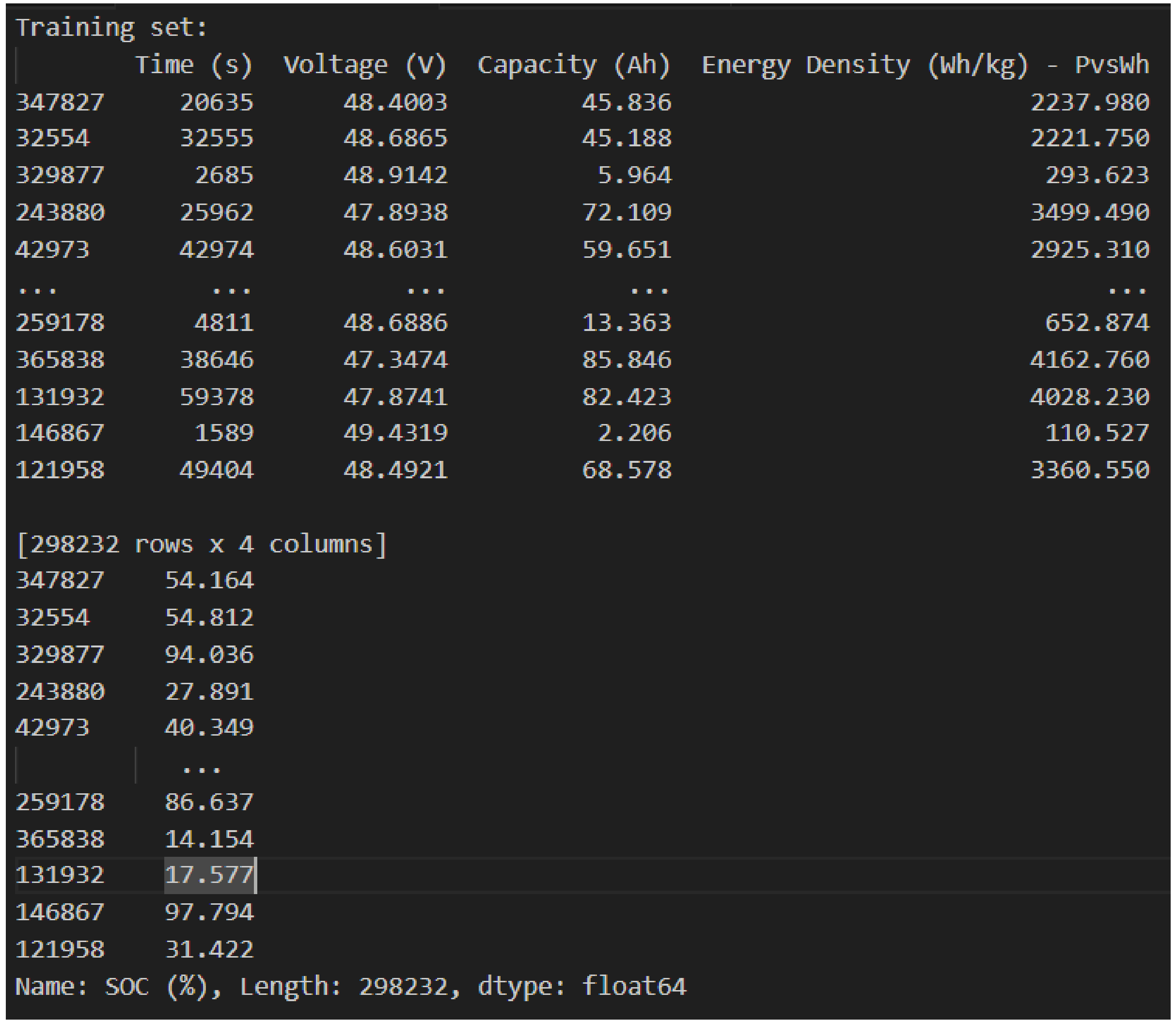
In this case, we chose the features with strong correlation, specifically **Time**, **Voltage**, **Capacity**, and **Energy Density**.

Data Preparation

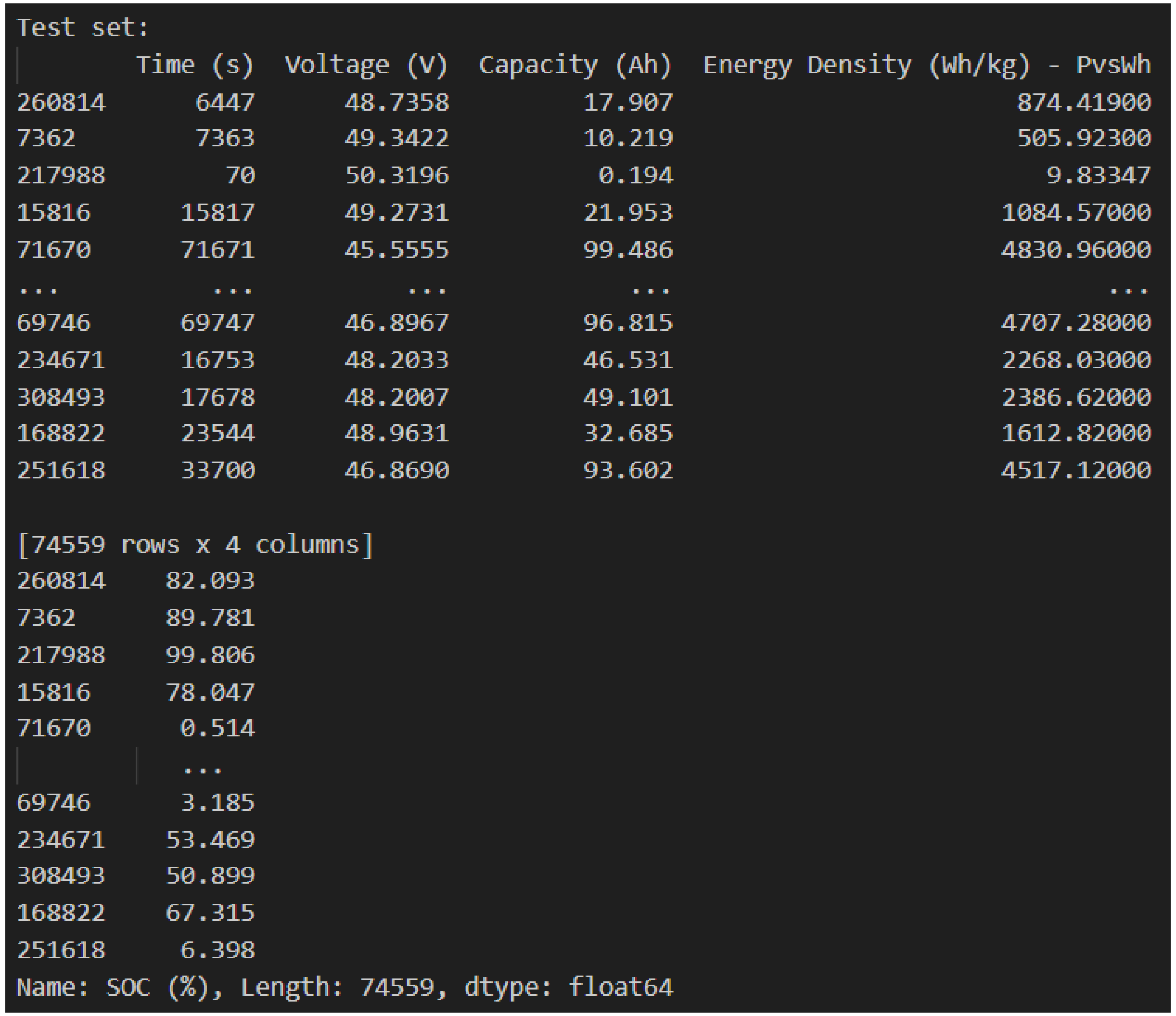
Data Splitting

In this section, we separate data to data training and data testing. We chose 80% of the data for training and 20% for testing. We use Scikit-Learn library to split the data.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```



```
Training set:
        Time (s)  Voltage (V)  Capacity (Ah)  Energy Density (Wh/kg) - PvsWh
347827     20635      48.4003         45.836                        2237.980
32554      32555      48.6865         45.188                        2221.750
329877      2685      48.9142          5.964                         293.623
243880     25962      47.8938         72.109                        3499.490
42973      42974      48.6031         59.651                        2925.310
...          ...          ...            ...                             ...
259178      4811      48.6886         13.363                         652.874
365838     38646      47.3474         85.846                        4162.760
131932     59378      47.8741         82.423                        4028.230
146867      1589      49.4319          2.206                         110.527
121958     49404      48.4921         68.578                        3360.550

[298232 rows x 4 columns]
347827    54.164
32554     54.812
329877    94.036
243880    27.891
42973     40.349
           ...
259178    86.637
365838    14.154
131932    17.577
146867    97.794
121958    31.422
Name: SOC (%), Length: 298232, dtype: float64
```



```
Test set:
        Time (s)  Voltage (V)  Capacity (Ah)  Energy Density (Wh/kg) - PvsWh
260814      6447      48.7358         17.907                       874.41900
7362        7363      49.3422         10.219                       505.92300
217988        70      50.3196          0.194                         9.83347
15816      15817      49.2731         21.953                      1084.57000
71670      71671      45.5555         99.486                      4830.96000
...          ...          ...            ...                             ...
69746      69747      46.8967         96.815                      4707.28000
234671     16753      48.2033         46.531                      2268.03000
308493     17678      48.2007         49.101                      2386.62000
168822     23544      48.9631         32.685                      1612.82000
251618     33700      46.8690         93.602                      4517.12000

[74559 rows x 4 columns]
260814    82.093
7362      89.781
217988    99.806
15816     78.047
71670      0.514
           ...
69746      3.185
234671    53.469
308493    50.899
168822    67.315
251618     6.398
Name: SOC (%), Length: 74559, dtype: float64
```

Modeling: Deep Learning

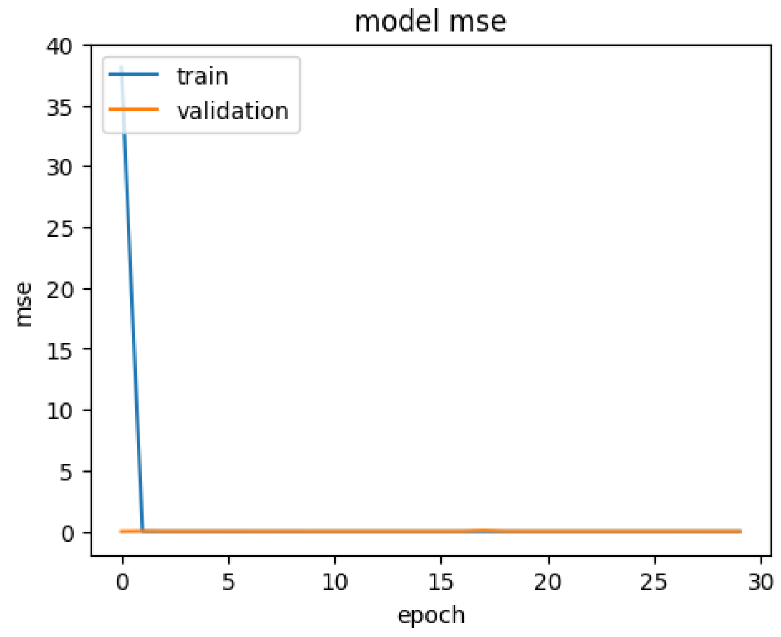
Hyperparameter Tuning

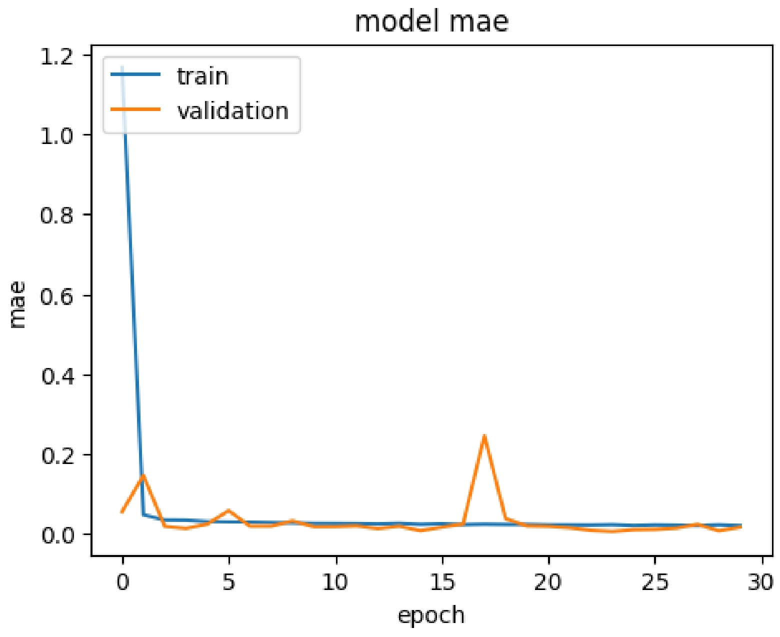
Hyperparameters are variables whose values are determined before the model training process and cannot be changed during training. They play a role in regulating the learning process and model structure, such as learning rate, number of epochs, number of layers, and number of neurons in each layer.

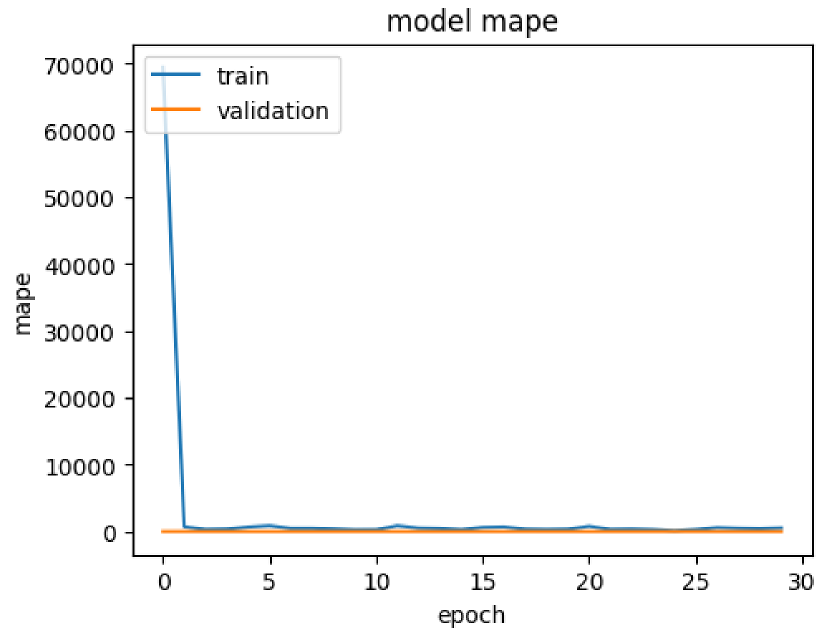
Hyperparameter	Content
Model	Long-Short Term Memory (LSTM)
Optimizer	Adam
Activation	ReLU
Batch Size	32
Input Layer	Count: 1 (4 features input)
Hidden Layer	Count: 2 Hidden Layer 1: 64 Neuron Hidden Layer 2: 32 Neuron
Output Layer	Count: 1
Epoch	30

Modeling: Deep Neural Network (DNN)

Training Performance



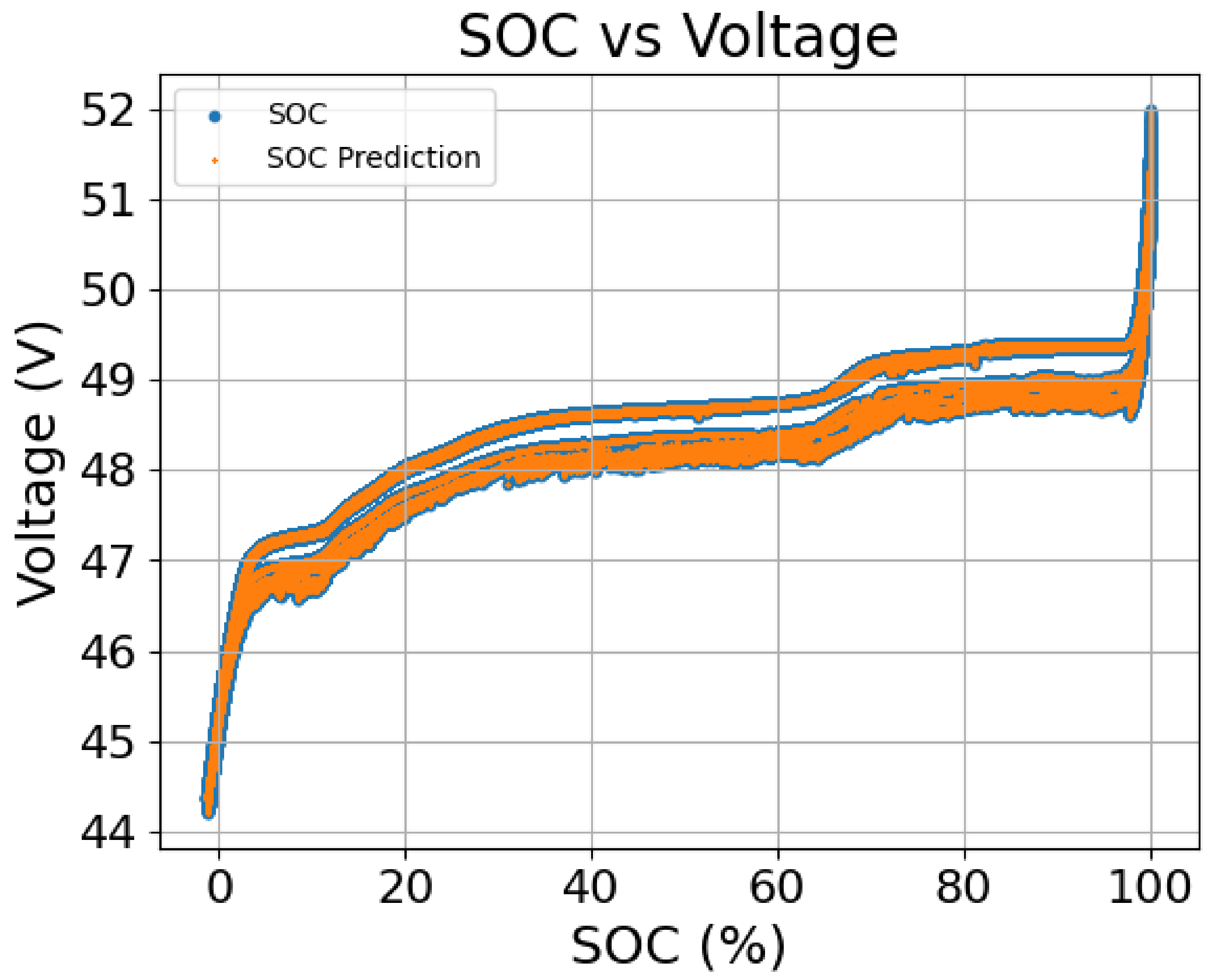




Training Time: 246.35202360153198 seconds
Prediction Time: 1.9037957191467285 seconds

Modeling: Deep Neural Network (DNN)

Model Evaluation



R2 training data = 0.9999995091519014
RMSE training data = 0.020388552990806424
MAE training data = 0.01781698805385481
MAPE training data = 0.5093883714043871 %

Modeling: Deep Learning

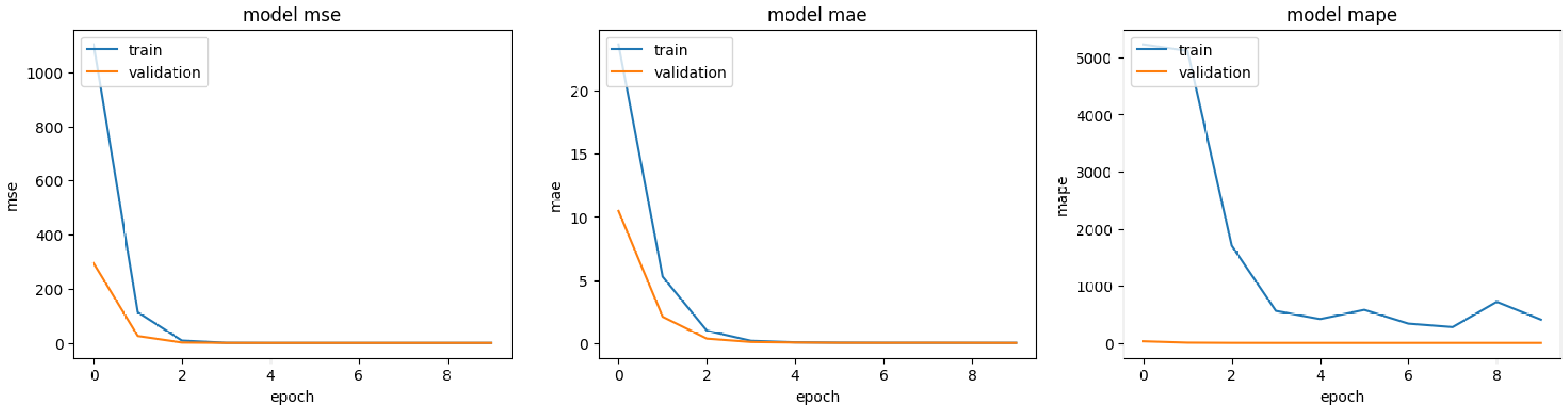
Hyperparameter Tuning

Hyperparameters are variables whose values are determined before the model training process and cannot be changed during training. They play a role in regulating the learning process and model structure, such as learning rate, number of epochs, number of layers, and number of neurons in each layer.

Hyperparameter	Content
Model	Long-Short Term Memory (LSTM)
Optimizer	Adam
Activation	ReLU
Batch Size	128
Input Layer	Count: 1 (4 features input)
Hidden Layer	Count: 2 Hidden Layer 1: 64 Neuron Hidden Layer 2: 32 Neuron
Output Layer	Count: 1
Epoch	10

Modeling: Long-Short Term Memory (LSTM)

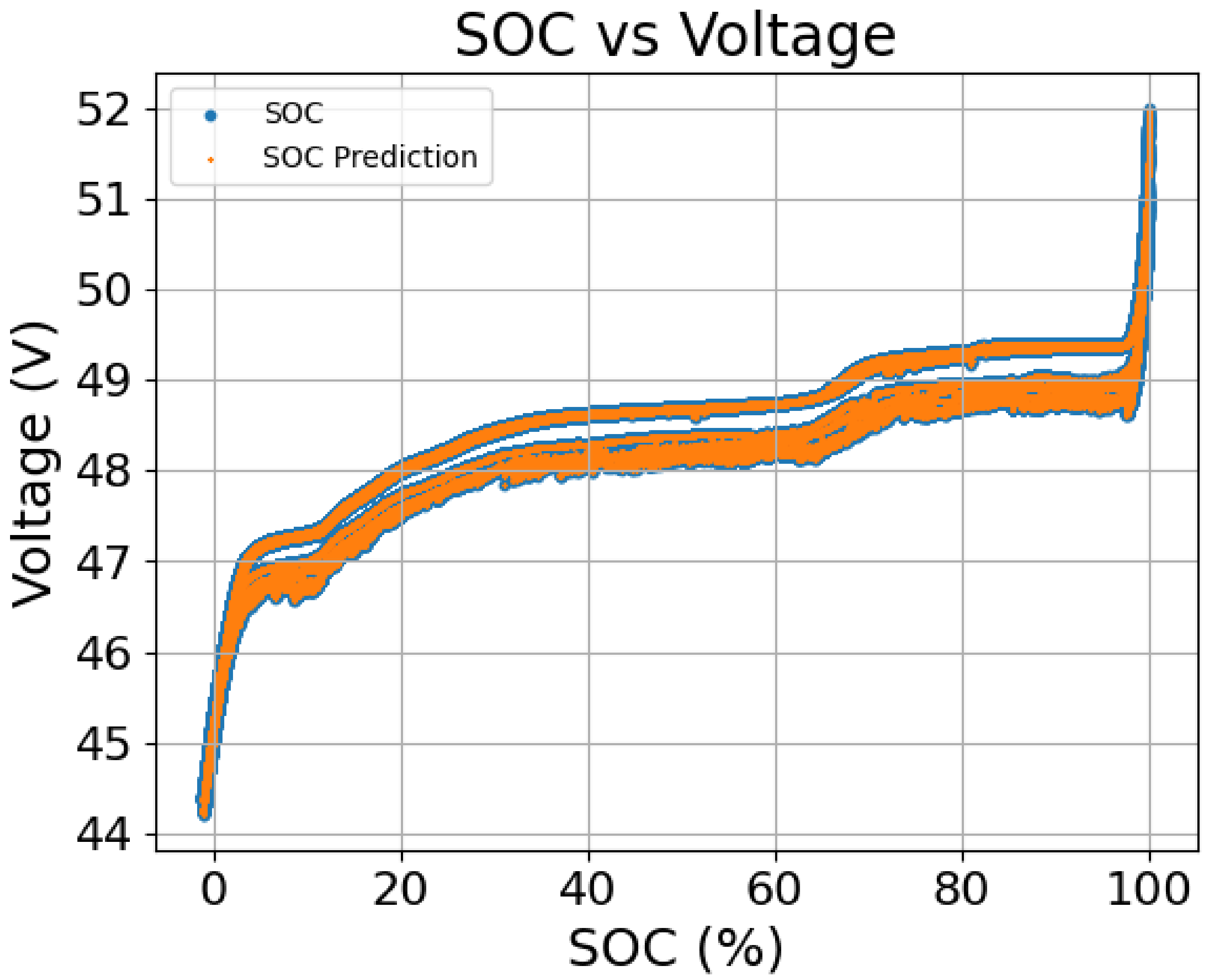
Training Performance



Training Time: `60.21635413169861 seconds`
Prediction Time: `3.7524471282958984 seconds`

Modeling: Long-Short Term Memory (LSTM)

Model Evaluation



R2 training data = 0.9999994309424449
RMSE training data = 0.021952851750538564
MAE training data = 0.015979483044945775
MAPE training data = 0.47290916751185985 %

Conclusion

- Both models, DNN and LSTM, exhibit exceptional performance with R2 values very close to 1, indicating their excellent ability to explain the variability in the training data. However, there are slight differences in their error metrics. The LSTM model demonstrates slightly lower Mean Absolute Error (MAE) and Mean Absolute Percentage Error (MAPE) compared to the DNN model, suggesting it has a marginally better performance in minimizing average prediction errors and percentage errors.
- Although the Root Mean Square Error (RMSE) for LSTM is slightly higher than that for DNN, it still indicates very low prediction error overall. These differences are minimal, indicating that both models are highly effective, but the LSTM model might have a slight edge in handling more complex time-series data.
- Ultimately, the choice between DNN and LSTM may depend on the specific needs of the project. For handling more intricate time-series patterns, LSTM might be more appropriate. However, in this particular case, both models provide highly satisfactory results.

Github Link

https://github.com/wilywho/Portfolio