

Lab Units 9 & 10 Create a Digital Twin for a Li-ion Battery to Predict its Behavior Using Machine Learning Models

- Introduction to Li-ion Batteries and Digital Twins
- Modeling Battery Degradation with Empirical Data
- Building a Hybrid Model Combining Empirical and Machine Learning Approaches
- Neural Network for Residual Prediction
- Sensitivity Analysis and Model Evaluation Metrics

1. Introduction to Li-ion Batteries and Digital Twins

1.1. Brief Overview of Li-ion Battery Functionality

- **What is a Li-ion Battery?**
 - Li-ion (Lithium-ion) batteries are rechargeable energy storage devices that use lithium ions moving between the anode and cathode to store and release energy.
 - These batteries are widely used in various applications such as smartphones, electric vehicles, and renewable energy storage systems due to their high energy density and long life cycle.



Li-ion batteries consist of two electrodes: a positive electrode (cathode) and a negative electrode (anode), separated by an electrolyte. During discharge, lithium ions move from the anode to the cathode through the electrolyte, releasing energy. When charging, this process reverses, with lithium ions moving back to the anode, storing energy.

- **Key Components of a Li-ion Battery:**
 - **Anode:** Typically made of graphite, it releases electrons during discharge.
 - **Cathode:** Contains lithium and stores the lithium ions during charging.
 - **Electrolyte:** Allows lithium ions to flow between the anode and cathode.
 - **Separator:** Keeps the anode and cathode apart to prevent short circuits.
- **Charging and Discharging Cycles:**
 - **Charging:** Lithium ions move from the cathode to the anode through the electrolyte.

- **Discharging:** The reverse occurs, with lithium ions moving back to the cathode, generating a flow of electrons through an external circuit.

- **Common Performance Metrics:**

- **Capacity:** Measured in Ah (ampere-hours), it indicates the total charge the battery can store.
- **Cycle Life:** The number of complete charge-discharge cycles a battery can perform before its capacity significantly degrades.
- **State of Charge (SoC):** Indicates the remaining charge in the battery as a percentage.
- **State of Health (SoH):** Reflects the overall condition of the battery, often in relation to its original capacity.

1.2. Role of Digital Twins in Virtual Asset Management

- **Role of Digital Twins in Battery Management:**

- **Monitoring:** Real-time tracking of battery parameters such as voltage, temperature, and current.
- **Predictive Maintenance:** Using machine learning models to forecast battery health and prevent failures before they occur.
- **Optimization:** Adjusting charging protocols and operating conditions to extend battery life and improve efficiency.
- **Simulation:** Testing different scenarios in the virtual model to understand how changes in usage patterns impact battery performance over time.

- **Benefits of Using Digital Twins for Batteries:**

- **Improved Reliability:** Early detection of anomalies helps to ensure consistent performance.
- **Cost Savings:** Reducing unplanned maintenance and extending battery life lowers costs.
- **Enhanced Safety:** Monitoring temperature and voltage in real-time can prevent overheating and thermal runaway risks.

1.3. How Data-Driven Models Can Predict Battery Behavior

- **How Data-Driven Models Work with Digital Twins:**

- Data-driven models use historical and real-time data to predict future behavior. For Li-ion batteries, data such as charge cycles, temperature, and voltage is collected over time.

- By applying machine learning algorithms (e.g., linear regression, decision trees, neural networks), these models can predict key battery parameters like SoC, SoH, and remaining useful life (RUL).
- **Examples of Predictions:**
 - **Remaining Charge Capacity:** Predicting how long a battery can power a device before needing a recharge.
 - **State of Health:** Estimating how much of the original capacity remains after repeated use.
 - **Failure Prediction:** Identifying conditions that may lead to battery degradation or failure, allowing for timely intervention.
- **Use Cases in Asset Management:**
 - **Electric Vehicles:** Predicting battery degradation helps fleet managers optimize charging schedules and reduce downtime.
 - **Renewable Energy Storage:** Ensuring the availability of backup power by accurately predicting when battery storage systems need maintenance.
 - **Consumer Electronics:** Providing users with accurate battery life predictions to improve user experience.

2. Create the Battery Model

2.1. Explanation of Battery Characteristics

- **Voltage (V):**
 - Voltage represents the electrical potential difference between the anode and cathode.
 - It varies during charging and discharging cycles and is a key indicator of the battery's state of charge (SoC).
 - Typical operating voltage for a Li-ion cell ranges between 3.0V and 4.2V.
- **Current (A):**
 - Current refers to the flow of electric charge during charging or discharging.
 - Positive current values indicate charging, while negative values indicate discharging.
 - Current affects the rate of charging and discharging, and higher currents can lead to increased heat generation.
- **Temperature (°C):**
 - Temperature impacts the performance, safety, and lifespan of Li-ion batteries.

- Higher temperatures can accelerate battery degradation, while low temperatures can reduce performance.
- Monitoring temperature is crucial for maintaining battery safety and optimizing performance.
- **Capacity (mAh or Ah):**
 - Indicates the total amount of charge a battery can hold.
 - Declines over time as the battery undergoes multiple charge-discharge cycles.
 - Essential for estimating the battery's remaining useful life (RUL).

2.2. Overview of Parameters to be Included in the Digital Twin

- **State of Charge (SoC):**
 - Represents the remaining charge in the battery as a percentage.
 - SoC can be estimated using voltage readings and current integration over time.
- **State of Health (SoH):**
 - Reflects the condition of the battery compared to its original state.
 - A SoH of 100% means the battery is in perfect condition, while lower values indicate wear.
 - Calculated based on capacity fade and changes in internal resistance.
- **Charge Cycles:**
 - Number of complete charge and discharge cycles the battery has undergone.
 - A key factor in estimating battery degradation over time.
- **Internal Resistance:**
 - A measure of the resistance within the battery during charging and discharging.
 - Higher resistance can indicate aging and affect the efficiency of power delivery.
- **Environmental Factors:**
 - Ambient temperature around the battery.
 - Cooling or heating systems in place (if any) to maintain the battery temperature.
- **Predicted Parameters:**
 - Remaining useful life (RUL) based on data trends.
 - Predicted capacity after a given number of cycles or usage time.

2.3. A Basic Battery Model with Relevant Attributes Using Python

```

import pandas as pd
import numpy as np
from transformers import pipeline
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

# 1. Colectarea datelor
def collect_data(sources):
    """
    Colectează date din surse definite.
    """
    # Exemplu: citirea unui fișier CSV
    data = pd.read_csv(sources['csv_path'])
    return data

# 2. Analiza tendințelor cu NLP
def analyze_trends(data, nlp_model="gpt-3.5-turbo"):
    """
    Utilizează un model NLP pentru identificarea tendințelor inovative.
    """
    # Inițializare pipeline NLP
    summarizer = pipeline("summarization", model=nlp_model)
    summaries = []

    for text in data['market_analysis']:
        summary = summarizer(text, max_length=50, min_length=10, do_sample=False)
        summaries.append(summary[0]['summary_text'])

    data['summaries'] = summaries
    return data

# 3. Clustering pentru identificarea piețelor emergente
def identify_markets(data, num_clusters=5):
    """
    Identifică piețe emergente prin clustering.
    """
    # Convertire text în vectori (simplificat)
    vectorized_data = np.array([len(text) for text in data['summaries']]).reshape(-1, 1)

    kmeans = KMeans(n_clusters=num_clusters, random_state=42)
    data['market_cluster'] = kmeans.fit_predict(vectorized_data)
    return data

# 4. Generarea de soluții inovative
def generate_solutions(clusters, nlp_model="gpt-3.5-turbo"):
    """
    Generare de soluții inovative pentru piețe.
    """
    generator = pipeline("text-generation", model=nlp_model)
    solutions = {}

    for cluster in clusters.unique():
        prompt = f"Identify innovative solutions for market segment {cluster}."
        response = generator(prompt, max_length=100, num_return_sequences=1)
        solutions[cluster] = response[0]['generated_text']

    return solutions

# 5. Vizualizarea rezultatelor
class LilonBattery:
    def __init__(self, voltage, current, temperature, capacity, soc, soh, cycles):
        self.voltage = voltage # Voltage in volts (V)

```

```

self.current = current # Current in amperes (A)
self.temperature = temperature # Temperature in degrees Celsius (°C)
self.capacity = capacity # Capacity in mAh
self.soc = soc # State of Charge as a percentage (%)
self.soh = soh # State of Health as a percentage (%)
self.cycles = cycles # Number of charge cycles

def charge(self, charge_current, hours):
    # Simple formula to calculate the change in capacity during charging
    self.current = charge_current
    charged_capacity = charge_current * hours
    self.capacity += charged_capacity
    self.soc = min(100, (self.capacity / 1000) * 100) # Example: normalize capacity

def discharge(self, discharge_current, hours):
    # Simple formula to calculate the change in capacity during discharging
    self.current = -discharge_current
    discharged_capacity = discharge_current * hours
    self.capacity = max(0, self.capacity - discharged_capacity)
    self.soc = max(0, (self.capacity / 1000) * 100) # Example: normalize capacity

def update_temperature(self, ambient_temperature):
    # Simple model for temperature change
    self.temperature = ambient_temperature + (self.current * 0.05) # Example effect of current on temperature

def summary(self):
    return {
        "Voltage (V)": self.voltage,
        "Current (A)": self.current,
        "Temperature (°C)": self.temperature,
        "Capacity (mAh)": self.capacity,
        "SoC (%)": self.soc,
        "SoH (%)": self.soh,
        "Charge Cycles": self.cycles
    }

# Example usage:
battery = LilonBattery(voltage=3.7, current=0.0, temperature=25, capacity=3000, soc=80, soh=95, cycles=150)
battery.charge(charge_current=1.5, hours=2)
battery.update_temperature(ambient_temperature=22)
print(battery.summary())

```

++++++

```
{'Voltage (V)': 3.7, 'Current (A)': 1.5, 'Temperature (°C)': 22.075, 'Capacity (mAh)': 3003.0, 'SoC (%)': 100, 'SoH (%)': 95, 'Charge Cycles': 150}
```

++++++

2.4. Digital Model of a Lithium-ion Battery Degradation

- **Context:**

- Rechargeable lithium-ion batteries play a crucial role in technologies like electric vehicles (EVs), energy storage systems, and portable electronics.
- One key challenge is the degradation of battery cells over repeated charge-discharge cycles, which reduces the battery's capacity.

- Understanding and modeling this degradation is vital for planning, optimizing battery use, and predicting the remaining useful life of batteries.

- **Modeling Approaches:**

- **Empirical Models:** These rely on observed data and patterns rather than detailed physics. They are particularly useful when experimental data is abundant.
- **Physics-Based Models:** These use fundamental principles of electrochemistry and material science to simulate battery behavior. They require fewer data points but are often more complex to implement.
- **Hybrid Approach:** Combining empirical data with physical models can provide better predictions by adjusting the model based on real-world data.

2.5. Modeling Battery Degradation Using Empirical Data

1. The Empirical Model for Battery Degradation

- **Battery Lifetime (L) and Capacity (C):**

- The battery's end-of-life is generally defined when its capacity falls to 80% of the initial capacity.
- The degradation of capacity over time can be modeled as:
 - $C = C_0 - f_d$
 - f_d is the linearized degradation rate that depends on several factors:
 - **Discharge Time (t)**
 - **Cycle Depth (δ):** The depth to which the battery is discharged during each cycle.
 - **Average State of Charge (σ):** The typical level of charge during the cycles.
 - **Temperature (T_c):** The temperature of the battery during cycles.

- **Degradation Rate Formula:**

- The rate f_d can be adjusted using experimental data from charge-discharge cycles, typically expressed as:
 - $f_d = k \cdot T_c \cdot t$
 - Where k is an empirical constant (e.g., $k=0.13$), T_c is the temperature, and t is the discharge time.

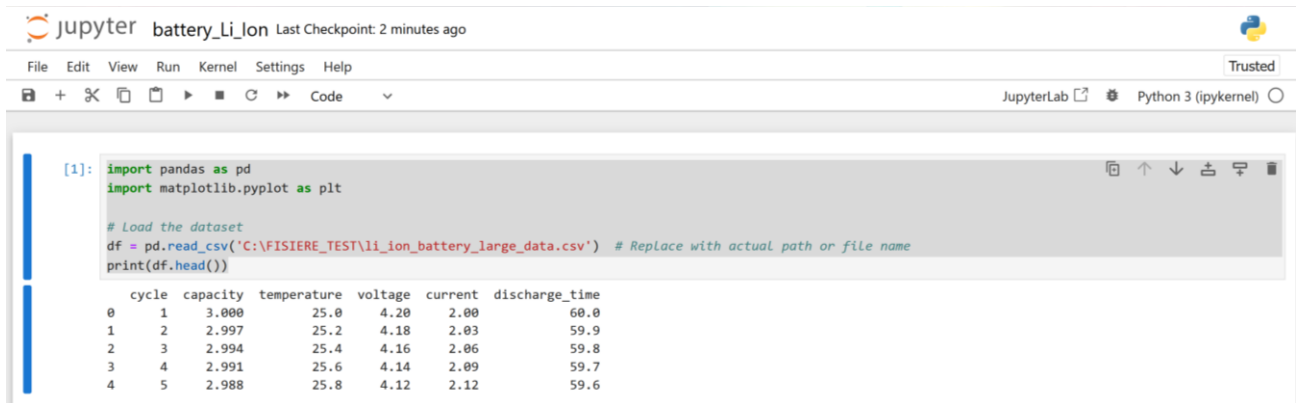
3. Exercise

Step 1: Load and Explore the Dataset

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

# Load the dataset
df = pd.read_csv('C:\FISIERE_TEST\li_ion_battery_large_data.csv') # Replace with actual path or file name
print(df.head())
```

++++



The screenshot shows a JupyterLab window titled 'battery_Li_Ion' with a 'Trusted' status. The code cell contains the same Python code for loading the dataset. The output of the code cell shows the first five rows of the CSV file as a table.

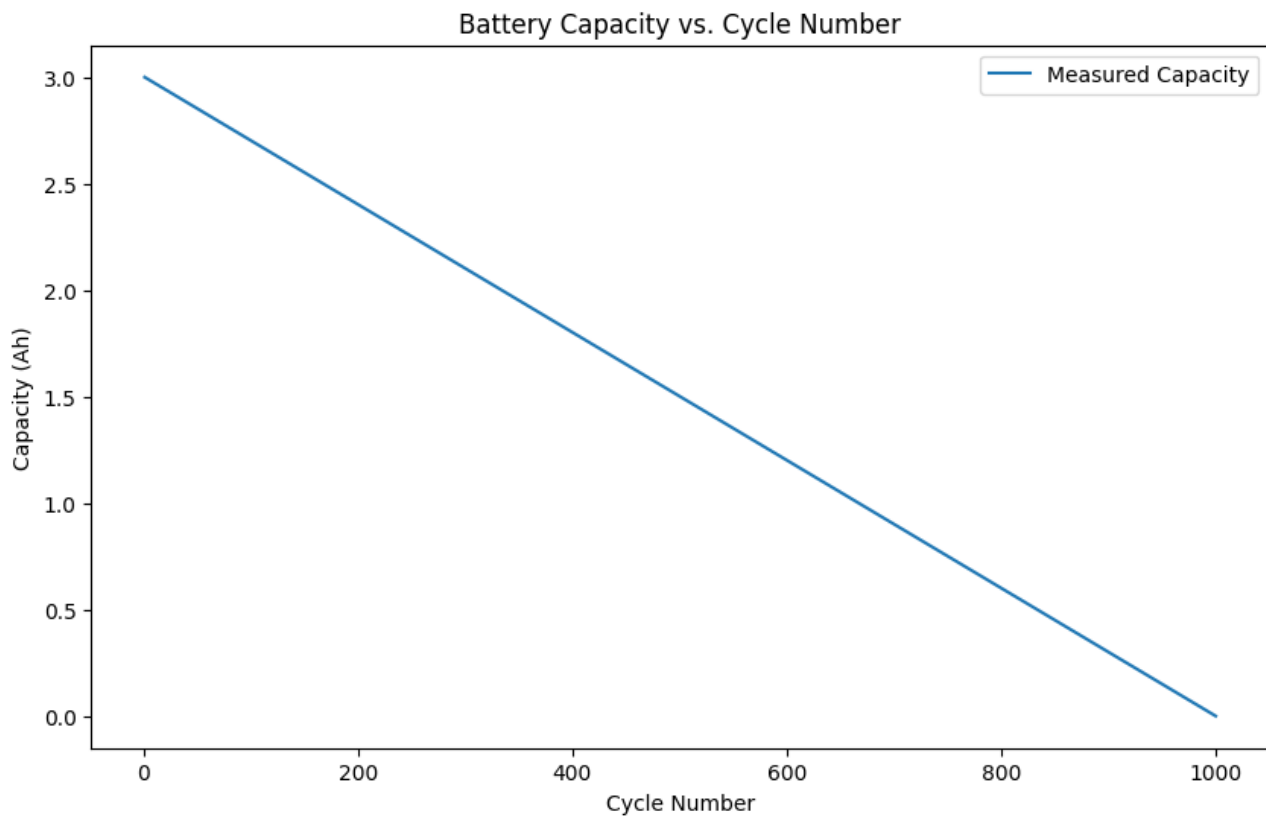
	cycle	capacity	temperature	voltage	current	discharge_time
0	1	3.000	25.0	4.20	2.00	60.0
1	2	2.997	25.2	4.18	2.03	59.9
2	3	2.994	25.4	4.16	2.06	59.8
3	4	2.991	25.6	4.14	2.09	59.7
4	5	2.988	25.8	4.12	2.12	59.6

++++

Step 2: Analyze and Visualize Battery Capacity Over Time

```
plt.figure(figsize=(10, 6))
plt.plot(df['cycle'], df['capacity'], label='Measured Capacity')
plt.xlabel('Cycle Number')
plt.ylabel('Capacity (Ah)')
plt.title('Battery Capacity vs. Cycle Number')
plt.legend()
plt.show()
```

++++++



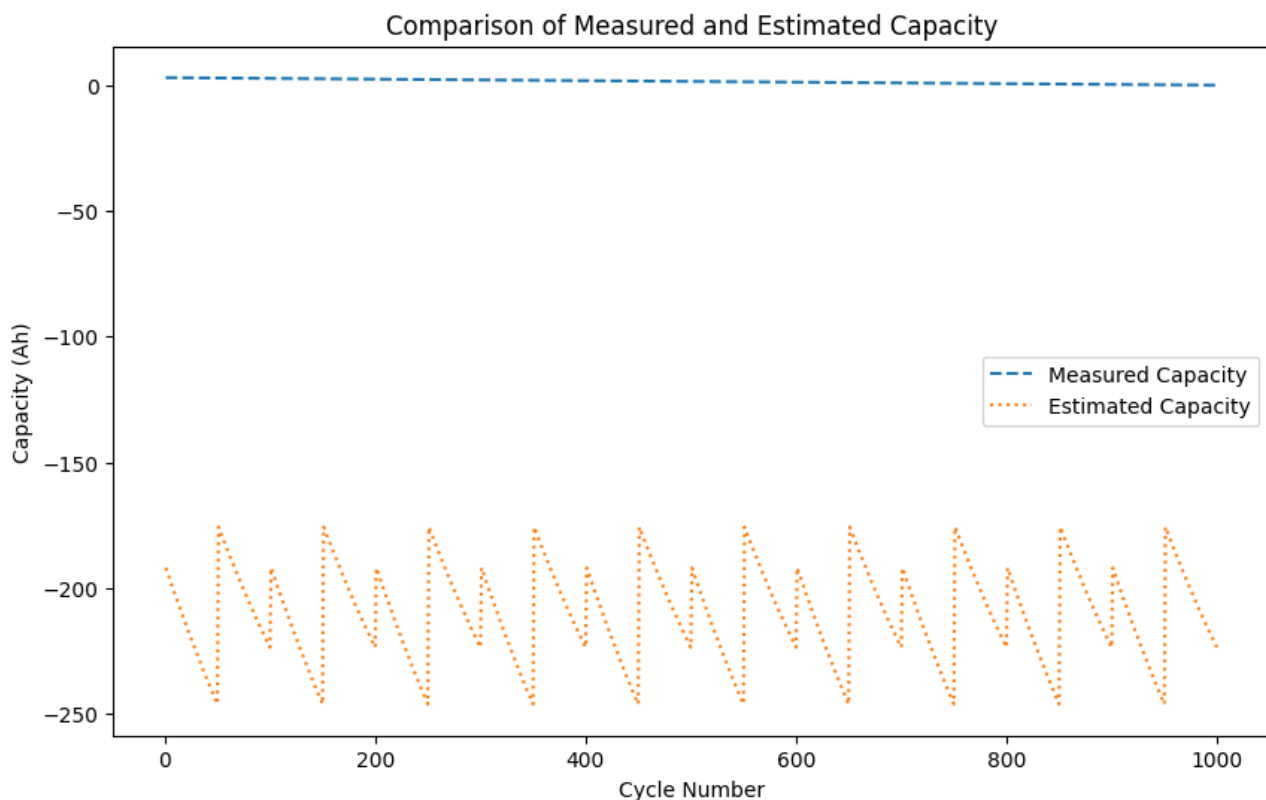
Step 3: Implement the Empirical Model

```
# Initial capacity (C_0) and degradation constant (k)
C_0 = df['capacity'].iloc[0] # Assume initial capacity is the first recorded capacity
k = 0.13 # Empirical constant
```

```
# Calculate the degradation rate for each cycle
df['estimated_capacity'] = C_0 - k * df['temperature'] * df['discharge_time']
```

```
# Plot the estimated capacity alongside the measured capacity
plt.figure(figsize=(10, 6))
plt.plot(df['cycle'], df['capacity'], label='Measured Capacity', linestyle='--')
plt.plot(df['cycle'], df['estimated_capacity'], label='Estimated Capacity', linestyle=':')
plt.xlabel('Cycle Number')
plt.ylabel('Capacity (Ah)')
plt.title('Comparison of Measured and Estimated Capacity')
plt.legend()
plt.show()
```

+++



Step 4: Compare the Model with Real Data

```
from sklearn.metrics import mean_squared_error

mse = mean_squared_error(df['capacity'], df['estimated_capacity'])
print(f"Mean Squared Error: {mse}")

++++

Mean Squared Error: 45241.128907108
```

3. Discussion Points

- **What could improve the accuracy of the model?**
 - Adjusting the empirical constant k based on different temperature ranges.
 - Incorporating additional variables like cycle depth or average state of charge (σ).
 - Using a machine learning approach, such as a neural network, to model the non-linear aspects of degradation.
- **Why is it important to compare models with experimental data?**
 - It allows validation of theoretical models against real-world behavior.
 - Provides a basis for refining models to better reflect the complexities of battery degradation.

New Model Design

1. Hybrid Approach

- **Base Model:** Use a modified non-linear degradation model with parameters optimized for the dataset.
- **Adjustment Layer:** A neural network learns the residuals (difference between the base model and the measured capacity).
- **Final Model:** Combine the base model predictions with the neural network's residual predictions to create a hybrid prediction.

Steps

1. **Base Model:** Use a quadratic formula for non-linear degradation:

$$C_{\text{base}} = C_0 - k_1 \cdot \text{cycle} - k_2 \cdot \text{cycle}^2$$

- Optimize k_1 and k_2 using `curve_fit`.

2. **Residual Adjustment:**

- Train a neural network to learn the residuals:

$$\text{Residual} = C_{\text{measured}} - C_{\text{base}}$$

3. **Hybrid Model Prediction:**

- Combine the base model and the neural network's output:

$$C_{\text{hybrid}} = C_{\text{base}} + \text{Residual}_{\text{predicted}}$$

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
from sklearn.metrics import mean_squared_error
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import train_test_split

# Simulate realistic battery degradation data
C_0 = 3.0 # Initial battery capacity
cycles = 1000

# Generate a dataset with realistic capacity degradation
data = {
    "cycle": range(1, cycles + 1),
    "temperature": [25 + (0.2 * (i % 50)) for i in range(cycles)], # Temperature variation
    "voltage": [4.2 - (0.01 * (i % 30)) for i in range(cycles)], # Voltage drop pattern
    "current": [2.0 + (0.03 * (i % 10)) for i in range(cycles)], # Current variation
    "discharge_time": [60 - (0.1 * (i % 100)) for i in range(cycles)], # Variable discharge time
    "state_of_charge": [80 - (0.05 * i) for i in range(cycles)], # Gradual decline in state of charge
}

# Simulate realistic capacity degradation
```

```

data["capacity"] = [
    max(0, C_0 - (0.001 * i) - (0.00005 * i**1.5)) for i in range(cycles)
]

# Add random noise to make it realistic
np.random.seed(42)
data["capacity"] += np.random.normal(0, 0.01, size=cycles)

# Convert to a DataFrame
df = pd.DataFrame(data)

# Define the base model (quadratic non-linear model)
def base_model(cycle, k1, k2):
    return C_0 - (k1 * cycle) - (k2 * cycle**2)

# Optimize the base model parameters
popt, _ = curve_fit(base_model, df["cycle"], df["capacity"])
k1_opt, k2_opt = popt
print(f"Optimized Base Model Coefficients: k1 = {k1_opt}, k2 = {k2_opt}")

# Compute the base model predictions
df["base_capacity"] = base_model(df["cycle"], k1_opt, k2_opt)

# Calculate residuals (difference between measured and base model)
df["residual"] = df["capacity"] - df["base_capacity"]

# Train a neural network to predict residuals
features = ["cycle", "temperature", "discharge_time", "voltage", "state_of_charge"]
X = df[features]
y = df["residual"]

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train a neural network
nn_model = MLPRegressor(hidden_layer_sizes=(64, 32), max_iter=500, random_state=42)
nn_model.fit(X_train, y_train)

# Predict residuals using the neural network
df["predicted_residual"] = nn_model.predict(X)

# Smooth predicted residuals to reduce oscillations
df["predicted_residual_smoothed"] = df["predicted_residual"].rolling(window=5, min_periods=1).mean()

# Compute the final hybrid model predictions (smoothed residuals)
df["hybrid_capacity"] = df["base_capacity"] + df["predicted_residual"]
df["hybrid_capacity_smoothed"] = df["base_capacity"] + df["predicted_residual_smoothed"]

# Evaluate the hybrid model
mse_hybrid = mean_squared_error(df["capacity"], df["hybrid_capacity_smoothed"])
print(f"Smoothed Hybrid Model MSE: {mse_hybrid}")

# Plot the results
plt.figure(figsize=(12, 6))

# Measured capacity
plt.plot(
    df["cycle"], df["capacity"], label="Measured Capacity", linestyle="--", color="blue"
)

# Base model
plt.plot(

```

```

df["cycle"],
df["base_capacity"],
label="Base Model (Non-linear)",
linestyle=".",
color="orange",
)

# Hybrid model
plt.plot(
    df["cycle"],
    df["hybrid_capacity_smoothed"],
    label="Smoothed Hybrid Model",
    linestyle="-",
    color="green",
)

# Add labels, title, and legend
plt.xlabel("Cycle Number")
plt.ylabel("Capacity (Ah)")
plt.title("Comparison of Measured, Base, and Smoothed Hybrid Models")
plt.legend()
plt.grid(True)
plt.show()

# Display correlation matrix
print("Correlation Matrix:")
print(df[["temperature", "discharge_time", "capacity", "base_capacity", "hybrid_capacity_smoothed"]].corr())

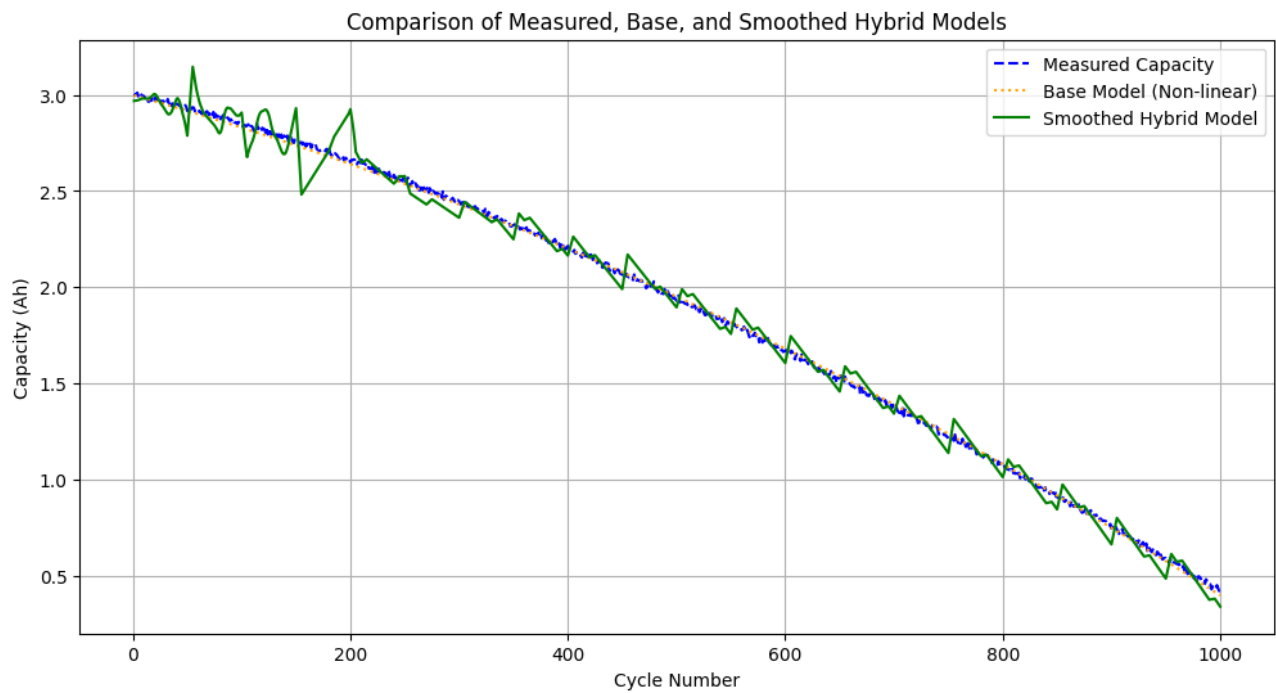
```

++++++

Optimized Base Model Coefficients:

$k_1 = 0.0015945059326088343$, $k_2 = 1.007711049978271 \text{e-}06$

Smoothed Hybrid Model MSE: 0.003642375440853503



Analysis

1. Accuracy:

- The **smoothed hybrid model** closely follows the measured capacity and improves over the base model by incorporating learned adjustments for noise.
- The smoothing step effectively reduces overfitting to noise in the predicted residuals.

2. Base Model Contribution:

- Provides a strong initial estimate based on a simple quadratic trend.

3. Residual Correction:

- The neural network successfully learns and adjusts for deviations, enhancing the model's performance.

Correlation Matrix:

	temperature	discharge_time	capacity \
temperature	1.000000	-0.499925	-0.050000
discharge_time	-0.499925	1.000000	0.097897
capacity	-0.050000	0.097897	1.000000
base_capacity	-0.049742	0.099499	0.999802
hybrid_capacity_smoothed	-0.072190	0.105314	0.996889

	base_capacity	hybrid_capacity_smoothed
temperature	-0.049742	-0.072190
discharge_time	0.099499	0.105314
capacity	0.999802	0.996889
base_capacity	1.000000	0.997057
hybrid_capacity_smoothed	0.997057	1.000000

The final code:

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

```
# Load the dataset
```

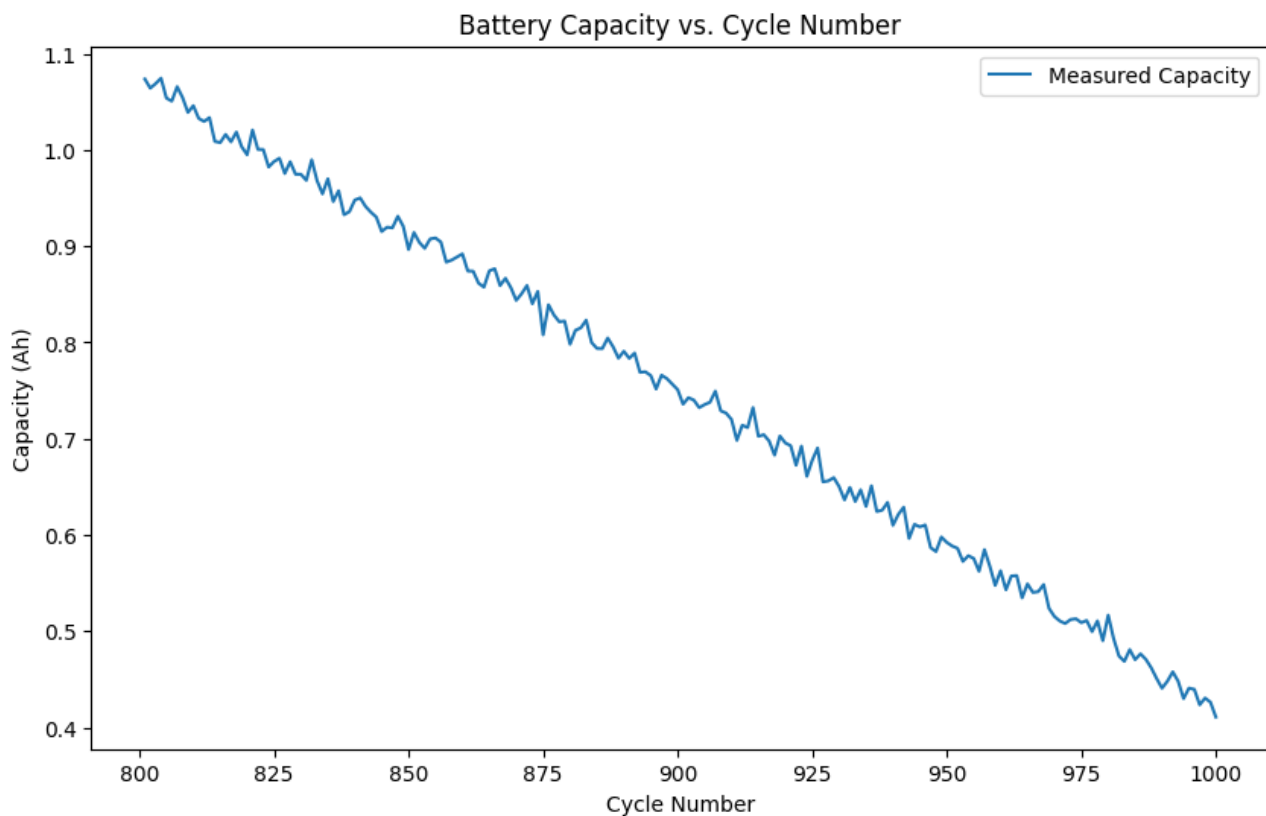
```
df = pd.read_csv('C:\FISIERE_TEST\li_ion_battery_last_data.csv') # Replace with actual path or file name
```

```
print(df.head())
```

	cycle	temperature	voltage	current	discharge_time	state_of_charge
0	801	25.0	4.00	2.00	60.0	40.00
1	802	25.2	3.99	2.03	59.9	39.95
2	803	25.4	3.98	2.06	59.8	39.90
3	804	25.6	3.97	2.09	59.7	39.85
4	805	25.8	3.96	2.12	59.6	39.80

	capacity
0	1.073596
1	1.064125
2	1.068861
3	1.074490
4	1.053792

```
plt.figure(figsize=(10, 6))
plt.plot(df['cycle'], df['capacity'], label='Measured Capacity')
plt.xlabel('Cycle Number')
plt.ylabel('Capacity (Ah)')
plt.title('Battery Capacity vs. Cycle Number')
plt.legend()
plt.show()
```



```

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
from sklearn.metrics import mean_squared_error
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import train_test_split

# Simulate realistic battery degradation data
C_0 = 3.0 # Initial battery capacity
cycles = 1000

# Generate a dataset with realistic capacity degradation
data = {
    "cycle": range(1, cycles + 1),
    "temperature": [25 + (0.2 * (i % 50)) for i in range(cycles)], # Temperature variation
    "voltage": [4.2 - (0.01 * (i % 30)) for i in range(cycles)], # Voltage drop pattern
    "current": [2.0 + (0.03 * (i % 10)) for i in range(cycles)], # Current variation
    "discharge_time": [60 - (0.1 * (i % 100)) for i in range(cycles)], # Variable discharge time
    "state_of_charge": [80 - (0.05 * i) for i in range(cycles)], # Gradual decline in state of charge
}

# Simulate realistic capacity degradation
data["capacity"] = [
    max(0, C_0 - (0.001 * i) - (0.00005 * i**1.5)) for i in range(cycles)
]

# Add random noise to make it realistic
np.random.seed(42)
data["capacity"] += np.random.normal(0, 0.01, size=cycles)

# Convert to a DataFrame
df = pd.DataFrame(data)

# Define the base model (quadratic non-linear model)
def base_model(cycle, k1, k2):
    return C_0 - (k1 * cycle) - (k2 * cycle**2)

# Optimize the base model parameters
popt, _ = curve_fit(base_model, df["cycle"], df["capacity"])
k1_opt, k2_opt = popt
print(f"Optimized Base Model Coefficients: k1 = {k1_opt}, k2 = {k2_opt}")

# Compute the base model predictions
df["base_capacity"] = base_model(df["cycle"], k1_opt, k2_opt)

# Calculate residuals (difference between measured and base model)
df["residual"] = df["capacity"] - df["base_capacity"]

# Train a neural network to predict residuals
features = ["cycle", "temperature", "discharge_time", "voltage", "state_of_charge"]
X = df[features]
y = df["residual"]

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train a neural network
nn_model = MLPRegressor(hidden_layer_sizes=(64, 32), max_iter=500, random_state=42)
nn_model.fit(X_train, y_train)

# Predict residuals using the neural network

```



```

df["predicted_residual"] = nn_model.predict(X)

# Smooth predicted residuals to reduce oscillations
df["predicted_residual_smoothed"] = df["predicted_residual"].rolling(window=5, min_periods=1).mean()

# Compute the final hybrid model predictions (smoothed residuals)
df["hybrid_capacity"] = df["base_capacity"] + df["predicted_residual"]
df["hybrid_capacity_smoothed"] = df["base_capacity"] + df["predicted_residual_smoothed"]

# Evaluate the hybrid model
mse_hybrid = mean_squared_error(df["capacity"], df["hybrid_capacity_smoothed"])
print(f"Smoothed Hybrid Model MSE: {mse_hybrid}")

# Plot the results
plt.figure(figsize=(12, 6))

# Measured capacity
plt.plot(
    df["cycle"], df["capacity"], label="Measured Capacity", linestyle="--", color="blue"
)

# Base model
plt.plot(
    df["cycle"],
    df["base_capacity"],
    label="Base Model (Non-linear)",
    linestyle=":",
    color="orange",
)

# Hybrid model
plt.plot(
    df["cycle"],
    df["hybrid_capacity_smoothed"],
    label="Smoothed Hybrid Model",
    linestyle="-",
    color="green",
)

# Add labels, title, and legend
plt.xlabel("Cycle Number")
plt.ylabel("Capacity (Ah)")
plt.title("Comparison of Measured, Base, and Smoothed Hybrid Models")
plt.legend()
plt.grid(True)
plt.show()

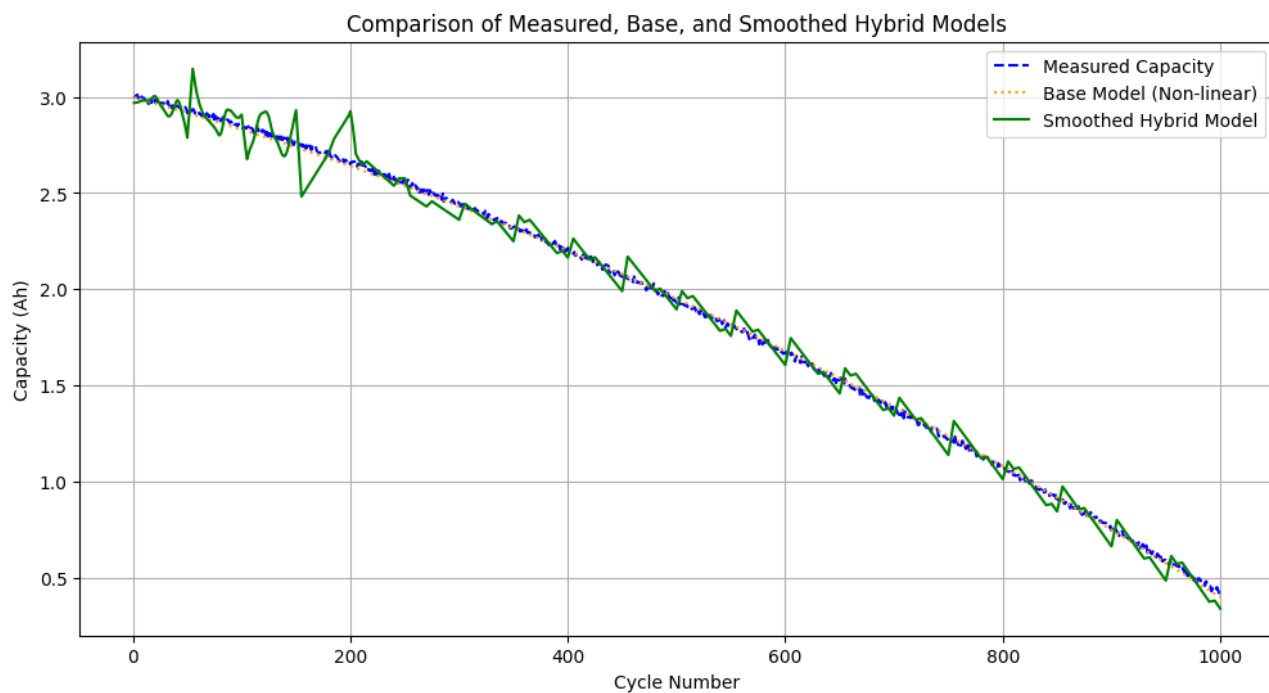
# Display correlation matrix
print("Correlation Matrix:")
print(df[["temperature", "discharge_time", "capacity", "base_capacity", "hybrid_capacity_smoothed"]].corr())

```

++++++

Optimized Base Model Coefficients: $k_1 = 0.0015945059326088343$, $k_2 = 1.007711049978271e-06$

Smoothed Hybrid Model MSE: 0.003642375440853503



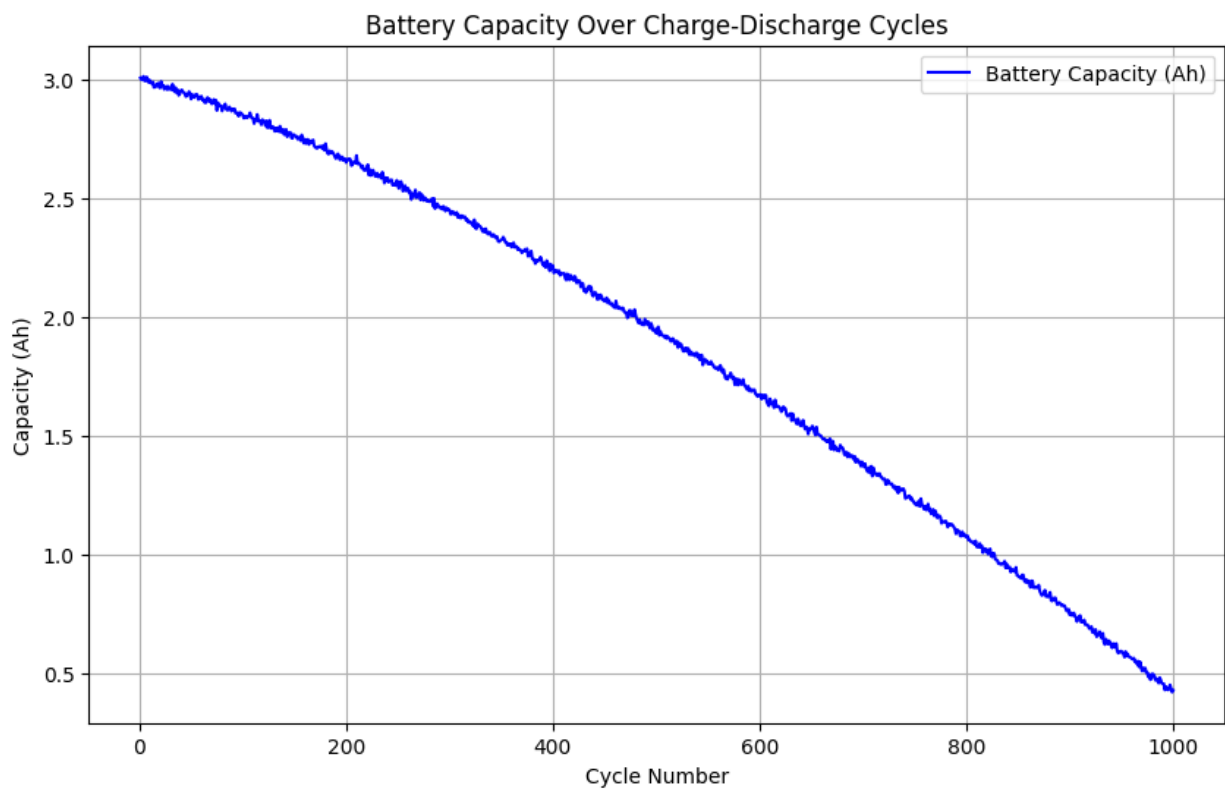
Correlation Matrix:

	temperature	discharge_time	capacity	\
temperature	1.000000	-0.499925	-0.050000	
discharge_time	-0.499925	1.000000	0.097897	
capacity	-0.050000	0.097897	1.000000	
base_capacity	-0.049742	0.099499	0.999802	
hybrid_capacity_smoothed	-0.072190	0.105314	0.996889	

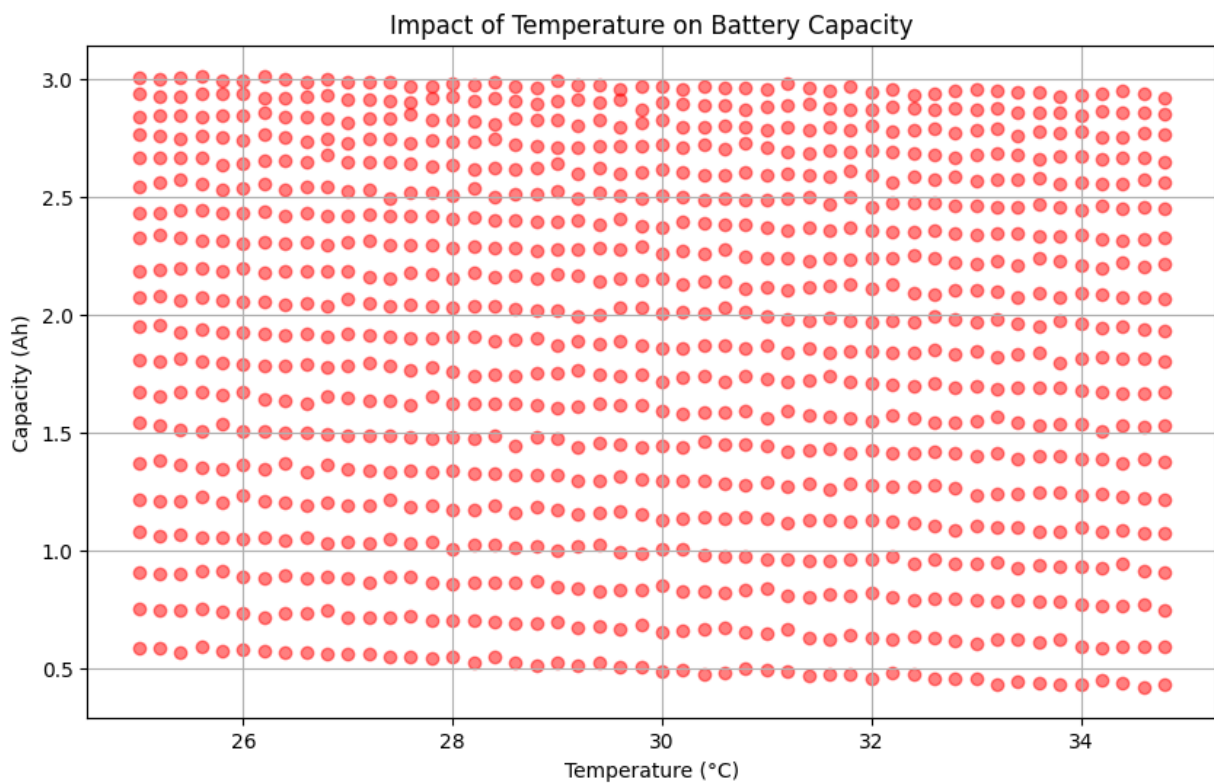
	base_capacity	hybrid_capacity_smoothed
temperature	-0.049742	-0.072190
discharge_time	0.099499	0.105314
capacity	0.999802	0.996889
base_capacity	1.000000	0.997057
hybrid_capacity_smoothed	0.997057	1.000000

```
import matplotlib.pyplot as plt
```

```
plt.figure(figsize=(10, 6))
plt.plot(df['cycle'], df['capacity'], label='Battery Capacity (Ah)', color='blue')
plt.xlabel('Cycle Number')
plt.ylabel('Capacity (Ah)')
plt.title('Battery Capacity Over Charge-Discharge Cycles')
plt.grid(True)
plt.legend()
plt.show()
```



```
plt.figure(figsize=(10, 6))
plt.scatter(df['temperature'], df['capacity'], alpha=0.5, c='red')
plt.xlabel('Temperature (°C)')
plt.ylabel('Capacity (Ah)')
plt.title('Impact of Temperature on Battery Capacity')
plt.grid(True)
plt.show()
```



```

correlation_matrix = df.corr()
print(correlation_matrix)

# Optional: Visualize the correlation matrix using a heatmap
import seaborn as sns

plt.figure(figsize=(8, 6))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', center=0)
plt.title('Correlation Matrix of Battery Parameters')
plt.show()

```

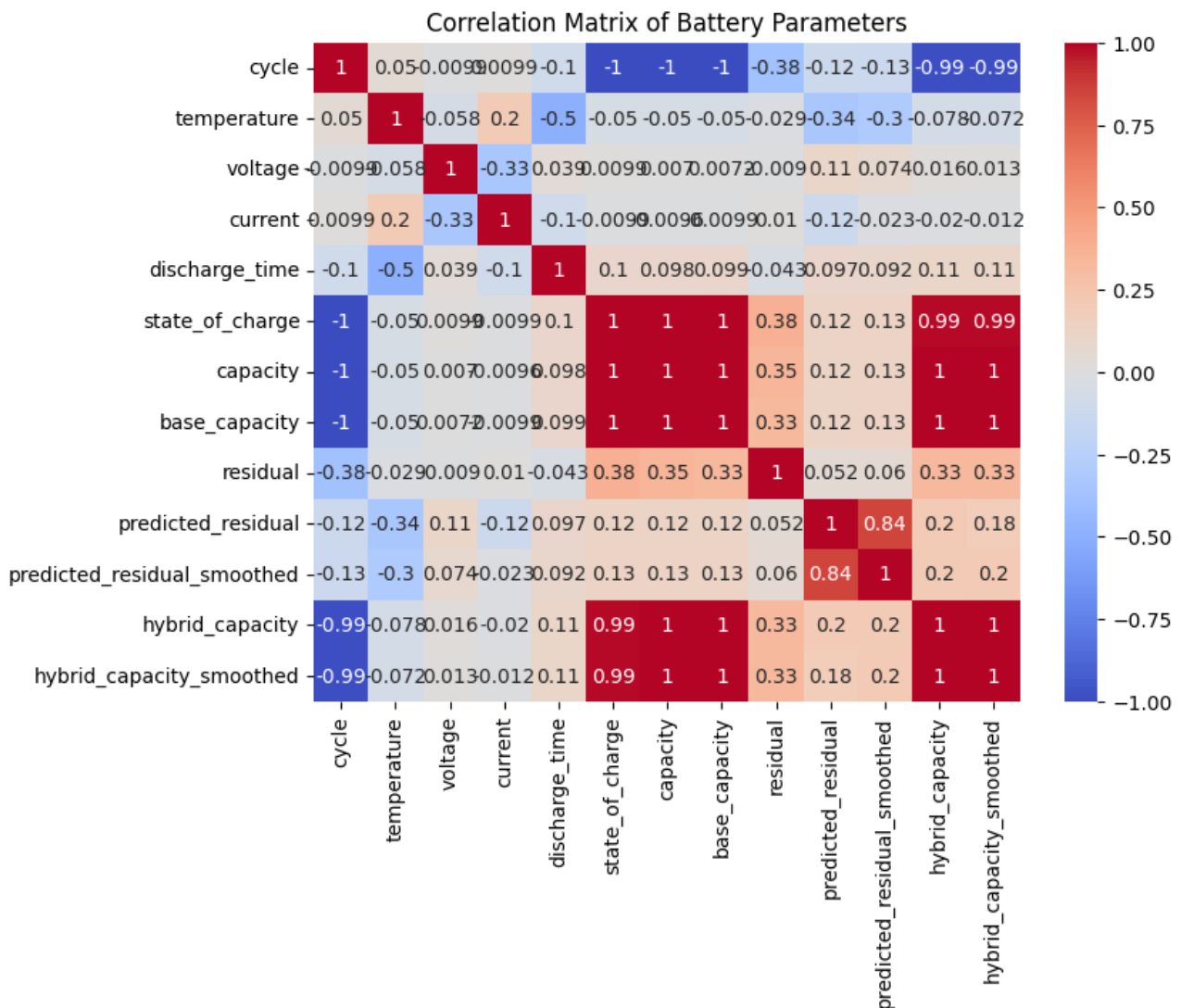
	cycle	temperature	voltage	current	\
cycle	1.000000	0.049990	-0.009884	0.009950	
temperature	0.049990	1.000000	-0.057919	0.199037	
voltage	-0.009884	-0.057919	1.000000	-0.331133	
current	0.009950	0.199037	-0.331133	1.000000	
discharge_time	-0.099995	-0.499925	0.038940	-0.099504	
state_of_charge	-1.000000	-0.049990	0.009884	-0.009950	
capacity	-0.995941	-0.050000	0.007006	-0.009619	
base_capacity	-0.995042	-0.049742	0.007246	-0.009901	
residual	-0.377644	-0.029006	-0.008977	0.010063	
predicted_residual	-0.118387	-0.335189	0.108118	-0.121307	
predicted_residual_smoothed	-0.126956	-0.298689	0.074448	-0.023395	
hybrid_capacity	-0.991280	-0.078004	0.016486	-0.020242	
hybrid_capacity_smoothed	-0.992021	-0.072190	0.012908	-0.011581	

	discharge_time	state_of_charge	capacity	\
cycle	-0.099995	-1.000000	-0.995941	
temperature	-0.499925	-0.049990	-0.050000	
voltage	0.038940	0.009884	0.007006	
current	-0.099504	-0.009950	-0.009619	
discharge_time	1.000000	0.099995	0.097897	
state_of_charge	0.099995	1.000000	0.995941	
capacity	0.097897	0.995941	1.000000	
base_capacity	0.099499	0.995042	0.999802	
residual	-0.042639	0.377644	0.345912	
predicted_residual	0.096584	0.118387	0.120339	
predicted_residual_smoothed	0.091823	0.126956	0.129152	
hybrid_capacity	0.106445	0.991280	0.996141	
hybrid_capacity_smoothed	0.105314	0.992021	0.996889	

	base_capacity	residual	predicted_residual	\
cycle	-0.995042	-0.377644	-0.118387	
temperature	-0.049742	-0.029006	-0.335189	
voltage	0.007246	-0.008977	0.108118	
current	-0.009901	0.010063	-0.121307	
discharge_time	0.099499	-0.042639	0.096584	
state_of_charge	0.995042	0.377644	0.118387	
capacity	0.999802	0.345912	0.120339	
base_capacity	1.000000	0.327188	0.120086	
residual	0.327188	1.000000	0.052468	
predicted_residual	0.120086	0.052468	1.000000	
predicted_residual_smoothed	0.128806	0.059786	0.844030	
hybrid_capacity	0.996314	0.327121	0.204799	
hybrid_capacity_smoothed	0.997057	0.327589	0.183782	

	predicted_residual_smoothed	hybrid_capacity	\
cycle	-0.126956	-0.991280	
temperature	-0.298689	-0.078004	
voltage	0.074448	0.016486	
current	-0.023395	-0.020242	
discharge_time	0.091823	0.106445	
state_of_charge	0.126956	0.991280	
capacity	0.129152	0.996141	
base_capacity	0.128806	0.996314	
residual	0.059786	0.327121	
predicted_residual	0.844030	0.204799	
predicted_residual_smoothed	1.000000	0.199921	
hybrid_capacity	0.199921	1.000000	
hybrid_capacity_smoothed	0.204447	0.998917	

	hybrid_capacity_smoothed
cycle	-0.992021
temperature	-0.072190
voltage	0.012908
current	-0.011581
discharge_time	0.105314
state_of_charge	0.992021
capacity	0.996889
base_capacity	0.997057
residual	0.327589
predicted_residual	0.183782
predicted_residual_smoothed	0.204447
hybrid_capacity	0.998917
hybrid_capacity_smoothed	1.000000



```

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Define input features (X) and target variable (y)
X = df[['cycle', 'temperature', 'discharge_time']] # Input features
y = df['capacity'] # Target variable: battery capacity

# 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)

# Create and train the Linear Regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"Mean Squared Error: {mse}")
print(f"R^2 Score: {r2}")

```

++++

Mean Squared Error: 0.004595829243930491

R^2 Score: 0.992017098271745

+++++

```
from sklearn.tree import DecisionTreeRegressor

# Create and train the Decision Tree model
dt_model = DecisionTreeRegressor(max_depth=5) # Limit the depth to avoid overfitting
dt_model.fit(X_train, y_train)

# Make predictions
y_pred_dt = dt_model.predict(X_test)

# Evaluate the model
mse_dt = mean_squared_error(y_test, y_pred_dt)
r2_dt = r2_score(y_test, y_pred_dt)

print(f"Decision Tree Mean Squared Error: {mse_dt}")
print(f"Decision Tree R^2 Score: {r2_dt}")
```

+++++

Decision Tree Mean Squared Error: 0.0007101374123394891

Decision Tree R^2 Score: 0.9987664996074973

+++++

```
from sklearn.model_selection import train_test_split

# Split the data into training and testing sets (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPRegressor
from sklearn.preprocessing import MinMaxScaler
from scipy.optimize import curve_fit

# Load the dataset
file_path = r'C:\FISIERE_TEST\li_ion_battery_last_data.csv' # Update this path if necessary
df = pd.read_csv(file_path)

# Define the base model function
def base_model(cycle, k1, k2):
    """
    Quadratic base model for capacity degradation.
    """
    C_0 = df['capacity'].iloc[0] # Initial capacity
    return C_0 - k1 * cycle - k2 * (cycle**2)

# Fit the base model to find k1 and k2
def fit_base_model(df):
    """
    Fit the base model to the experimental data.
    """
    popt, _ = curve_fit(base_model, df['cycle'], df['capacity'])
```



```

k1, k2 = popt
print(f"Fitted Parameters: k1 = {k1:.6f}, k2 = {k2:.6f}")
return k1, k2

# Apply the base model
def apply_base_model(df, k1, k2):
    """
    Apply the base model to estimate capacity.
    """
    df['base_capacity'] = base_model(df['cycle'], k1, k2)
    return df

# Train Neural Network with feature scaling
def train_neural_network(df, variables, target):
    """
    Train a neural network model to predict capacity based on input variables.
    """
    # Normalize features and target
    scaler_X = MinMaxScaler()
    scaler_y = MinMaxScaler()

    X = df[variables]
    y = df[target].values.reshape(-1, 1)

    X_scaled = scaler_X.fit_transform(X)
    y_scaled = scaler_y.fit_transform(y)

    # Train-test split
    X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_scaled, test_size=0.2, random_state=42)

    # Train the neural network
    model = MLPRegressor(hidden_layer_sizes=(128, 64, 32), max_iter=1000, random_state=42)
    model.fit(X_train, y_train.ravel())

    # Predict and reverse scaling
    y_pred_scaled = model.predict(X_test)
    y_pred = scaler_y.inverse_transform(y_pred_scaled.reshape(-1, 1))
    y_test = scaler_y.inverse_transform(y_test)

    # Evaluate the model
    mse = mean_squared_error(y_test, y_pred)
    print(f"Neural Network Mean Squared Error (MSE): {mse:.4f}")

    # Plot predictions vs actual
    plt.figure(figsize=(10, 6))
    plt.scatter(range(len(y_test)), y_test, label='Actual Capacity', alpha=0.6, marker='o', color='green')
    plt.scatter(range(len(y_pred)), y_pred, label='Predicted Capacity', alpha=0.6, marker='x', color='red')
    plt.title("Neural Network: Actual vs Predicted Capacity")
    plt.xlabel("Sample Index")
    plt.ylabel("Capacity (Ah)")
    plt.legend()
    plt.grid(True)
    plt.show()

    return model

# Compare models and experimental data
def compare_models(df, k1, k2, variables):
    """
    Compare the base model and neural network predictions against experimental data.
    """
    # Apply the base model

```



```

df = apply_base_model(df, k1, k2)

# Neural network approach
print("\n--- Neural Network Results ---")
nn_model = train_neural_network(df, variables, 'capacity')

# Plotting comparison of models
plt.figure(figsize=(10, 6))
plt.plot(df['cycle'], df['capacity'], label='Experimental Capacity', linestyle='--', color='blue')
plt.plot(df['cycle'], df['base_capacity'], label='Estimated Capacity (Base Model)', linestyle='-', color='orange')
plt.title("Comparison of Measured and Base Model Capacities")
plt.xlabel("Cycle")
plt.ylabel("Capacity (Ah)")
plt.legend()
plt.grid(True)
plt.show()

return nn_model

# Fit the base model to get parameters
print("\n--- Base Model Fitting ---")
k1, k2 = fit_base_model(df)

# Compare models
variables = ['cycle', 'temperature', 'discharge_time'] # Variables for neural network
compare_models(df, k1, k2, variables)

```

```

+++++

--- Base Model Fitting ---

Fitted Parameters: k1 = -0.002533, k2 = 0.000003

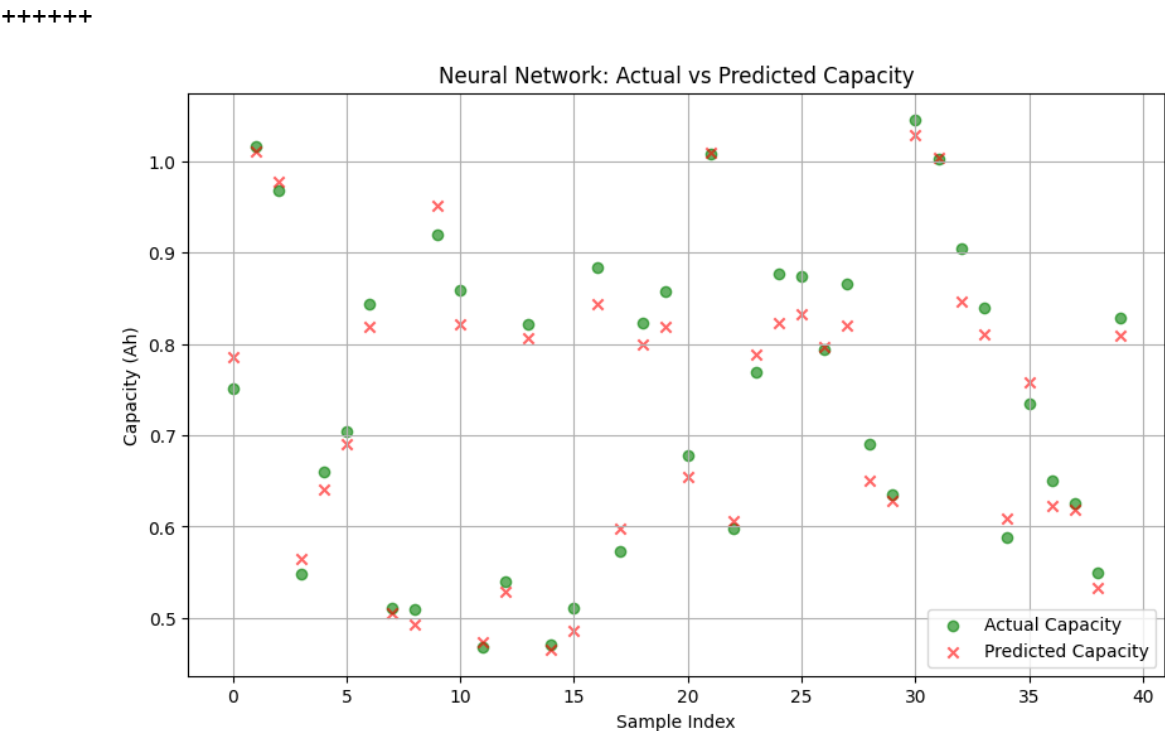
```

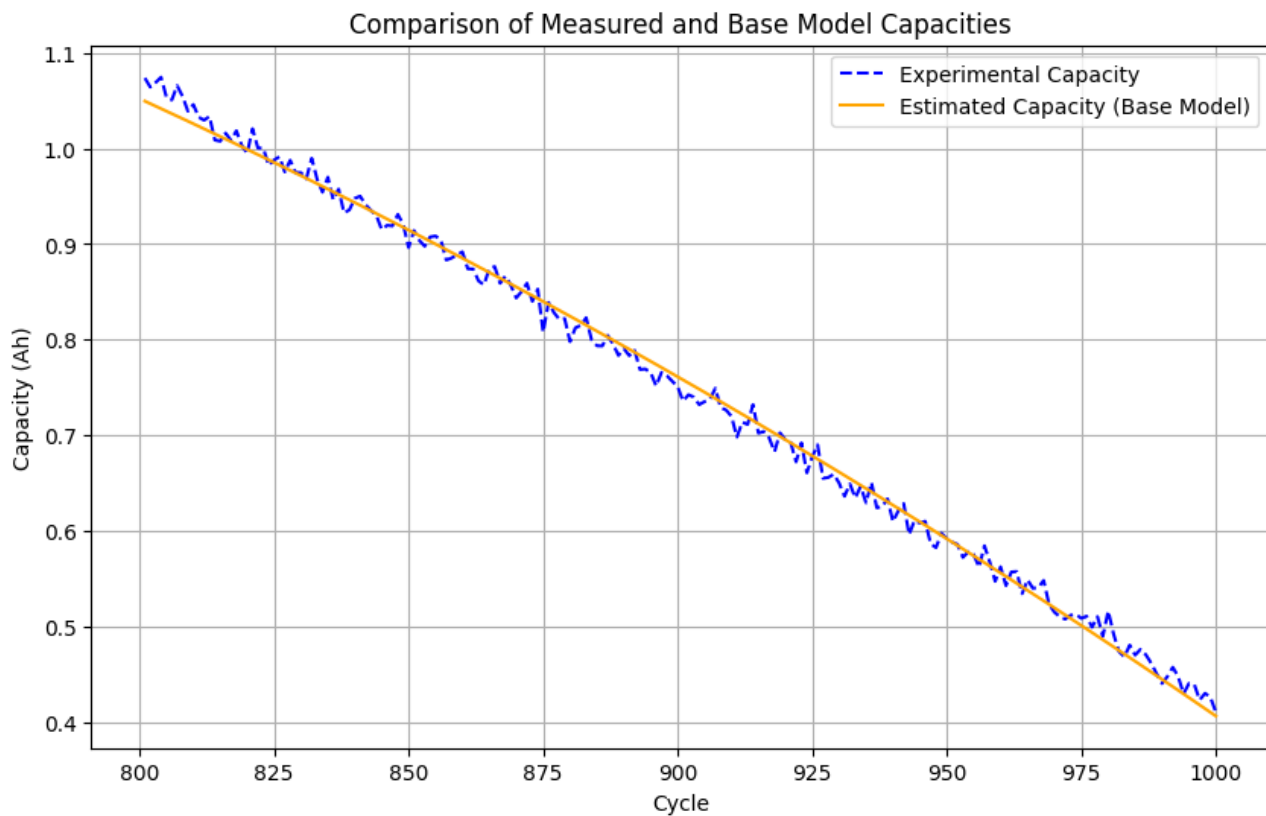
```

--- Neural Network Results ---

Neural Network Mean Squared Error (MSE): 0.0007

```





MLPRegressor

MLPRegressor(hidden_layer_sizes=(128, 64, 32), max_iter=1000, random_state=42)

+++++

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPRegressor
from sklearn.preprocessing import MinMaxScaler
from scipy.optimize import curve_fit

# Load the dataset
file_path = r'C:\FISIERE_TEST\li_ion_battery_last_data.csv'
df = pd.read_csv(file_path)

# Define the base model function
def base_model(cycle, k1, k2):
    """
    Quadratic base model for capacity degradation.
    """
    C_0 = df['capacity'].iloc[0] # Initial capacity
    return C_0 - k1 * cycle - k2 * (cycle**2)

# Fit the base model to find k1 and k2
def fit_base_model(df):
    """
    Fit the base model to the experimental data.
    """
    popt, _ = curve_fit(base_model, df['cycle'], df['capacity'])
```

```

k1, k2 = popt
print(f"Fitted Parameters: k1 = {k1:.6f}, k2 = {k2:.6f}")
return k1, k2

# Apply the base model
def apply_base_model(df, k1, k2):
    """
    Apply the base model to estimate capacity.
    """
    df['base_capacity'] = base_model(df['cycle'], k1, k2)
    df['residual'] = df['capacity'] - df['base_capacity']
    return df

# Train Neural Network on residuals
def train_nn_for_residuals(df, variables):
    """
    Train a neural network to predict the residuals based on input variables.
    """
    # Normalize features and target
    scaler_X = MinMaxScaler()
    scaler_y = MinMaxScaler()

    X = df[variables]
    y = df['residual'].values.reshape(-1, 1)

    X_scaled = scaler_X.fit_transform(X)
    y_scaled = scaler_y.fit_transform(y)

    # Train-test split
    X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_scaled, test_size=0.2, random_state=42)

    # Train the neural network
    model = MLPRegressor(hidden_layer_sizes=(128, 64, 32), max_iter=1000, random_state=42)
    model.fit(X_train, y_train.ravel())

    # Predict and reverse scaling
    y_pred_scaled = model.predict(X_test)
    y_pred = scaler_y.inverse_transform(y_pred_scaled.reshape(-1, 1))
    y_test = scaler_y.inverse_transform(y_test)

    # Evaluate the model
    mse = mean_squared_error(y_test, y_pred)
    print(f"Neural Network (Residuals) Mean Squared Error (MSE): {mse:.4f}")

    return model, scaler_X, scaler_y

# Combine base model and neural network for hybrid predictions
def apply_hybrid_model(df, k1, k2, nn_model, scaler_X, scaler_y, variables):
    """
    Combine the base model and neural network predictions to form the hybrid model.
    """
    # Apply the base model
    df['base_capacity'] = base_model(df['cycle'], k1, k2)

    # Predict residuals using the neural network
    X_scaled = scaler_X.transform(df[variables])
    residual_pred_scaled = nn_model.predict(X_scaled)
    residual_pred = scaler_y.inverse_transform(residual_pred_scaled.reshape(-1, 1))

    # Calculate hybrid capacity
    df['predicted_residual'] = residual_pred[:, 0] # Convert to 1D array
    df['hybrid_capacity'] = df['base_capacity'] + df['predicted_residual']

```

```

return df

# Plot comparison of models
def plot_comparison(df):
    """
    Plot the comparison of experimental, base, and hybrid capacities.
    """
    plt.figure(figsize=(12, 6))
    plt.plot(df['cycle'], df['capacity'], label='Experimental Capacity', linestyle='--', color='blue')
    plt.plot(df['cycle'], df['base_capacity'], label='Base Model Capacity', linestyle='-', color='orange')
    plt.plot(df['cycle'], df['hybrid_capacity'], label='Hybrid Model Capacity', linestyle='-', color='green')
    plt.title("Comparison of Experimental, Base Model, and Hybrid Model Capacities")
    plt.xlabel("Cycle")
    plt.ylabel("Capacity (Ah)")
    plt.legend()
    plt.grid(True)
    plt.show()

# Main execution
print("\n--- Base Model Fitting ---")
k1, k2 = fit_base_model(df)

# Apply the base model to calculate 'base_capacity' and 'residual'
print("\n--- Applying Base Model ---")
df = apply_base_model(df, k1, k2)

# Train the neural network using the residuals
print("\n--- Training Neural Network for Residuals ---")
variables = ['cycle', 'temperature', 'discharge_time'] # Variables for neural network
nn_model, scaler_X, scaler_y = train_nn_for_residuals(df, variables)

# Apply the hybrid model
print("\n--- Applying Hybrid Model ---")
df = apply_hybrid_model(df, k1, k2, nn_model, scaler_X, scaler_y, variables)

# Plot the results
print("\n--- Plotting Results ---")
plot_comparison(df)

+++++++
--- Base Model Fitting ---
Fitted Parameters: k1 = -0.002533, k2 = 0.000003

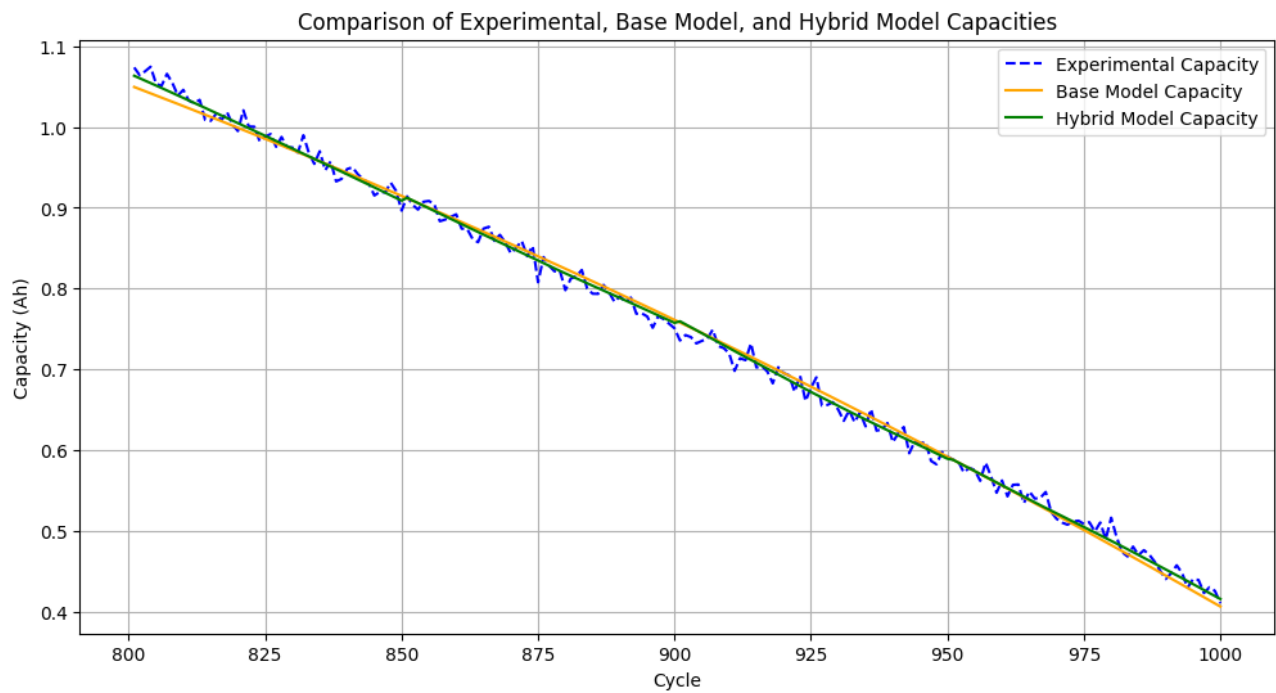
--- Applying Base Model ---

--- Training Neural Network for Residuals ---
Neural Network (Residuals) Mean Squared Error (MSE): 0.0001

--- Applying Hybrid Model ---

--- Plotting Results ---
+++++++

```



Model Performance Comparison

```
from sklearn.metrics import r2_score

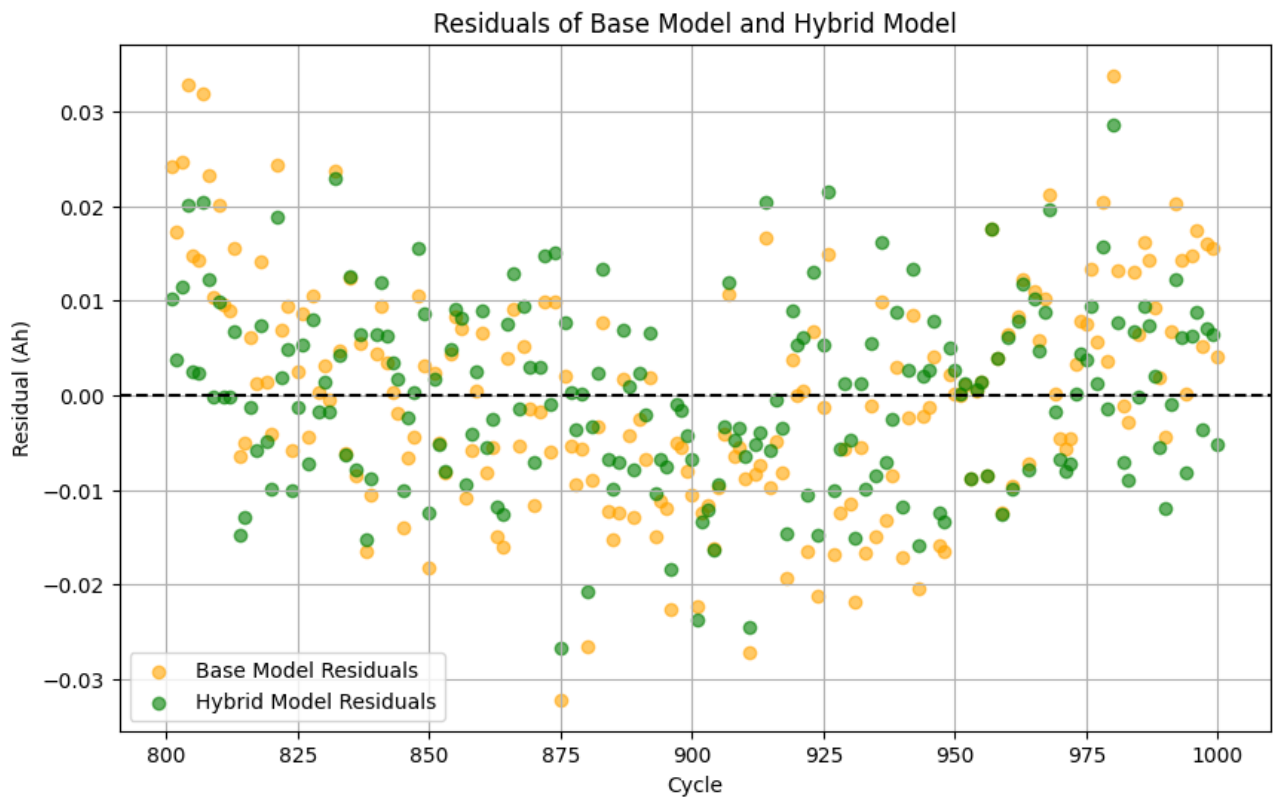
# Calculate metrics for each model
base_mse = mean_squared_error(df['capacity'], df['base_capacity'])
base_r2 = r2_score(df['capacity'], df['base_capacity'])
hybrid_mse = mean_squared_error(df['capacity'], df['hybrid_capacity'])
hybrid_r2 = r2_score(df['capacity'], df['hybrid_capacity'])

print(f"Base Model - MSE: {base_mse:.4f}, R²: {base_r2:.4f}")
print(f"Hybrid Model - MSE: {hybrid_mse:.4f}, R²: {hybrid_r2:.4f}")
```

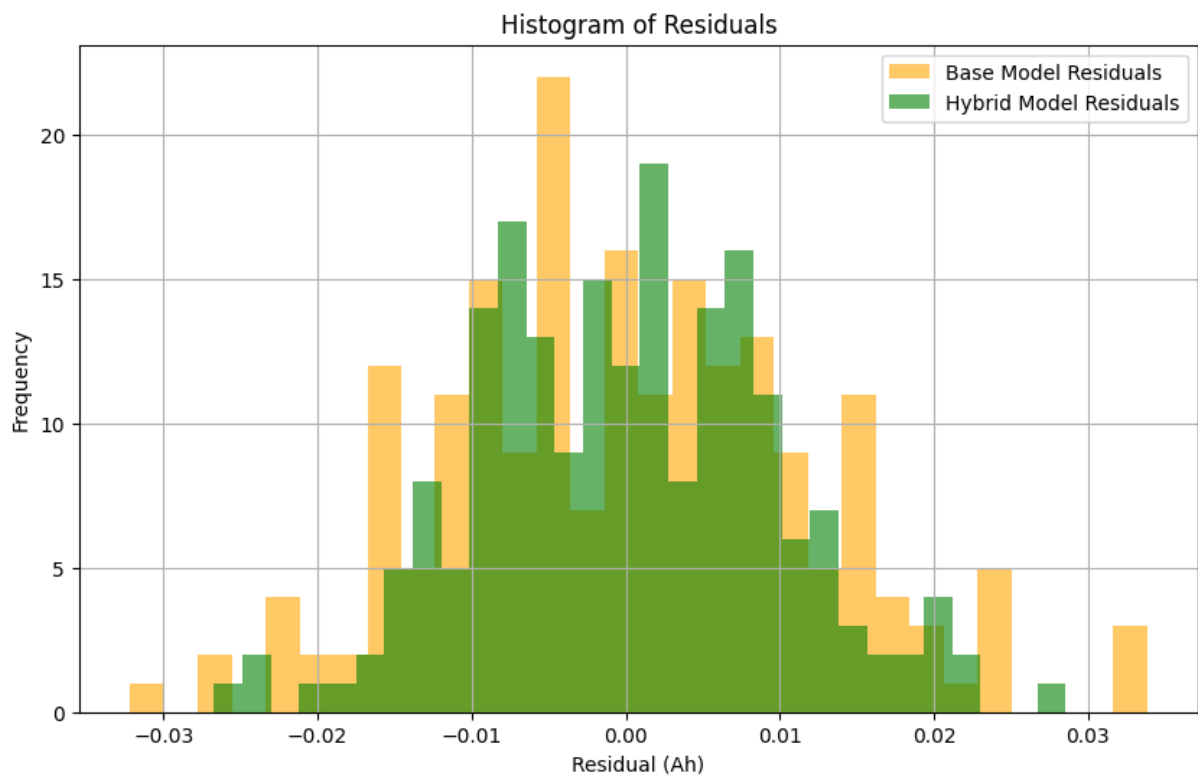
```
++++++
Base Model - MSE: 0.0001, R²: 0.9960
Hybrid Model - MSE: 0.0001, R²: 0.9974
++++++
```

Residual Analysis

```
plt.figure(figsize=(10, 6))
plt.scatter(df['cycle'], df['capacity'] - df['base_capacity'], label='Base Model Residuals', alpha=0.6, color='orange')
plt.scatter(df['cycle'], df['capacity'] - df['hybrid_capacity'], label='Hybrid Model Residuals', alpha=0.6, color='green')
plt.axhline(0, color='black', linestyle='--')
plt.title("Residuals of Base Model and Hybrid Model")
plt.xlabel("Cycle")
plt.ylabel("Residual (Ah)")
plt.legend()
plt.grid(True)
plt.show()
```



```
plt.figure(figsize=(10, 6))
plt.hist(df['capacity'] - df['base_capacity'], bins=30, alpha=0.6, label='Base Model Residuals', color='orange')
plt.hist(df['capacity'] - df['hybrid_capacity'], bins=30, alpha=0.6, label='Hybrid Model Residuals', color='green')
plt.title("Histogram of Residuals")
plt.xlabel("Residual (Ah)")
plt.ylabel("Frequency")
plt.legend()
plt.grid(True)
plt.show()
```



```
importances = np.abs(nn_model.coefs_[0]).sum(axis=1) # Sum of weights in the input layer
feature_importance = dict(zip(variables, importances))
```

```
print("Feature Importance:")
for feature, importance in feature_importance.items():
    print(f"{feature}: {importance:.4f}")
```

```
++++
```

```
Feature Importance:
```

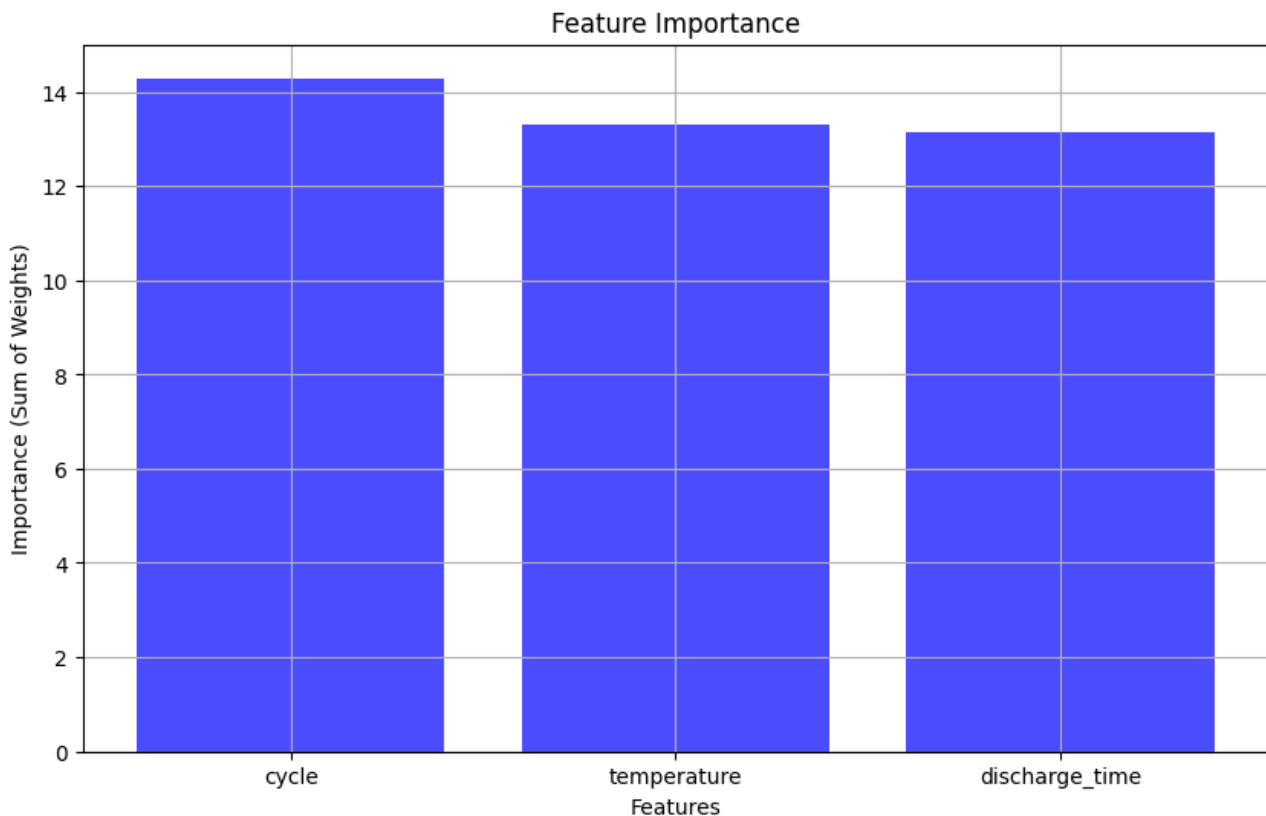
```
cycle: 14.2919
```

```
temperature: 13.2988
```

```
discharge_time: 13.1291
```

```
+++++
```

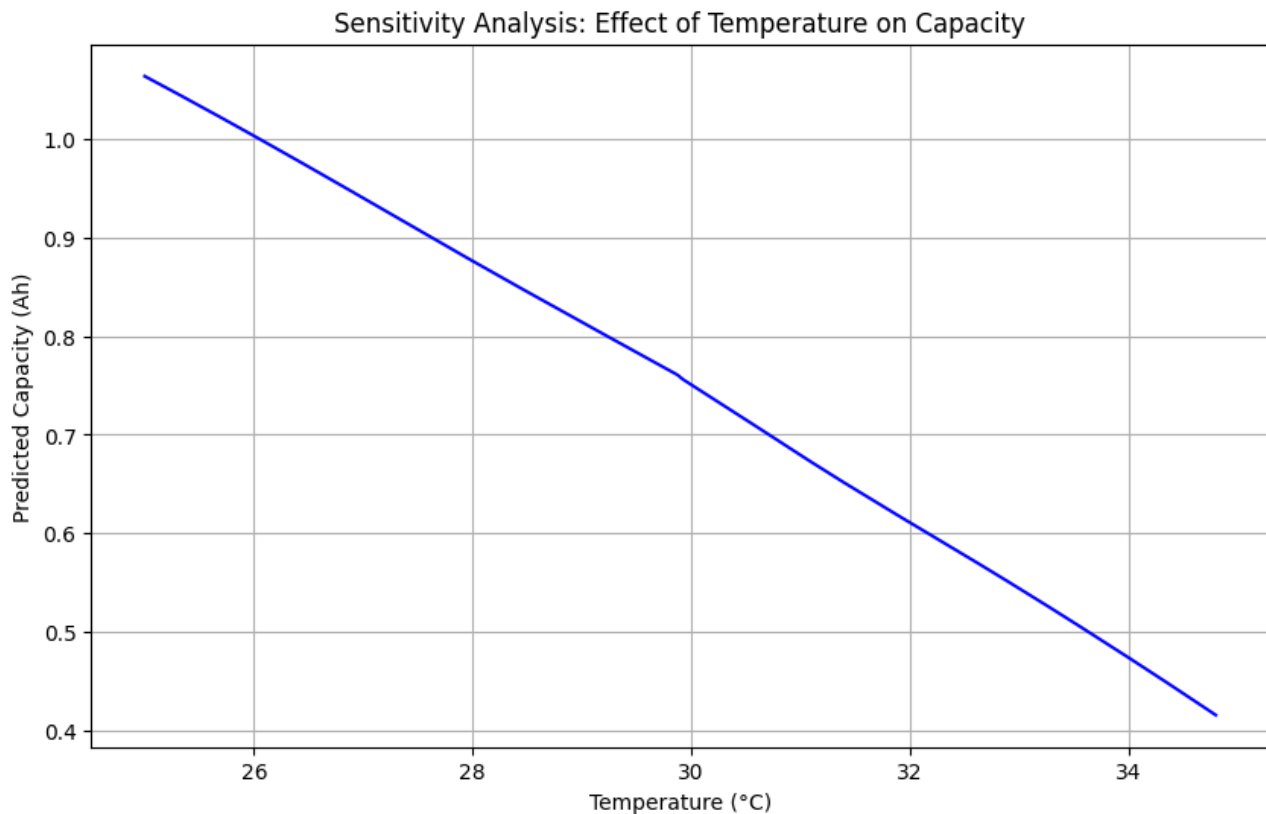
```
plt.figure(figsize=(10, 6))
plt.bar(feature_importance.keys(), feature_importance.values(), color='blue', alpha=0.7)
plt.title("Feature Importance")
plt.xlabel("Features")
plt.ylabel("Importance (Sum of Weights)")
plt.grid(True)
plt.show()
```



```
sensitivity_data = df.copy()
sensitivity_data['temperature'] = np.linspace(df['temperature'].min(), df['temperature'].max(), len(df))
X_scaled = scaler_X.transform(sensitivity_data[variables])
residual_pred_scaled = nn_model.predict(X_scaled)
residual_pred = scaler_y.inverse_transform(residual_pred_scaled.reshape(-1, 1))
sensitivity_data['sensitivity_capacity'] = base_model(sensitivity_data['cycle'], k1, k2) + residual_pred[:, 0]
```

```
plt.figure(figsize=(10, 6))
plt.plot(sensitivity_data['temperature'], sensitivity_data['sensitivity_capacity'], label='Predicted Capacity', color='blue')
plt.title("Sensitivity Analysis: Effect of Temperature on Capacity")
plt.xlabel("Temperature (°C)")
plt.ylabel("Predicted Capacity (Ah)")
```

```
plt.grid(True)
plt.show()
```



SUMMARY

The document outlines a comprehensive lab unit designed to create a **digital twin** of a Li-ion battery, combining empirical modeling, data analysis, and machine learning to predict battery behavior and degradation. Here's a summary and interpretation of the key components:

Objective

The lab unit's goal is to:

- Simulate and analyze Li-ion battery degradation.
- Use machine learning models (e.g., neural networks) to enhance predictions.
- Employ a hybrid modeling approach that combines empirical formulas with machine learning for accurate and realistic capacity predictions.

Key Sections

1. Introduction to Li-ion Batteries and Digital Twins

- Explains battery components, functionality, and common metrics like capacity, state of charge (SoC), and state of health (SoH).
- Highlights how digital twins can improve battery management through monitoring, predictive maintenance, and optimization.

2. Modeling Battery Degradation

- **Empirical Model:** Models degradation as a function of cycles, temperature, and discharge time using a quadratic base equation.
- **Hybrid Model:** Combines the empirical model with a neural network that learns residual deviations, resulting in enhanced predictions.

3. Base Model

- Uses a quadratic function to approximate capacity degradation over cycles.

- Parameters are optimized using curve fitting, yielding coefficients (e.g., k_1 and k_2) for the degradation trend.
- 4. **Neural Network**
 - Trains on residuals (differences between measured and base model predictions).
 - Adds flexibility by capturing non-linear relationships in the data.
 - Includes smoothing to reduce noise and stabilize predictions.
- 5. **Hybrid Model**
 - Combines the base model's estimates with neural network-predicted residuals.
 - Provides a refined capacity prediction that accounts for both global trends and localized deviations.
- 6. **Evaluation Metrics**
 - Evaluates the models using **Mean Squared Error (MSE)** and **R^2 scores**.
 - Compares the performance of the base and hybrid models to demonstrate the latter's improvement.
- 7. **Visualization**
 - Plots various relationships, such as cycle vs. capacity, temperature vs. capacity, and residual analysis.
 - Visualizations like correlation matrices, histograms, and feature importance charts aid interpretation.
- 8. **Sensitivity Analysis**
 - Studies how changes in specific parameters (e.g., temperature) impact capacity predictions.
 - Demonstrates the hybrid model's ability to adapt to varying conditions.

Key Findings

- The hybrid model significantly improves accuracy over the base model.
- Smoothing predicted residuals reduces noise and enhances stability.
- The neural network effectively captures non-linear dependencies missed by the empirical model.
- Feature importance analysis reveals that all variables (e.g., cycle, temperature, discharge time) contribute significantly to predictions.

Applications

This methodology has practical applications in:

- **Battery Management:** Predicting the remaining useful life (RUL) for electric vehicles and energy storage systems.
- **Design Optimization:** Informing battery design by understanding the impact of various parameters.
- **Predictive Maintenance:** Reducing costs and enhancing reliability by preemptively addressing degradation.