Jai-Kumar786 /
Machine_Learning_and_Alogorithms

Code Issues Pull requests Actions Projects Wiki Security

master

Machine_Learning_and_Alogorithms / 3.Feature Engineering
/ 05-29-2025 Feature Engineering.ipynb

Jai-Kumar786 05-30-2025 Feature Engineering Completed 0714e11 · yesterday

11576 lines (11576 loc) · 1.2 MB

LOW CURRENT OCV AT -10 degree celcius

In [1]:

```
import pandas as pd


# Read a specific sheet by name
low_curr_ocv_minus_10= pd.read_excel(r"C:\Users\jaiku\PycharmProjects\Assig


low_curr_ocv_minus_10.head(10)
```

Out[1]:

	Data_Point	Test_Time(s)	Date_Time	Step_Time(s)	Step_Index	Cycle_Index	Cur
0	1	3.010944	2012-06-29 16:51:38	3.010944	1	1	0
1	2	3.026961	2012-06-29 16:51:38	0.015765	2	1	1.
2	3	8.035992	2012-06-29 16:51:44	5.007575	3	1	0.
3	4	13.043648	2012-06-29 16:51:49	10.015231	3	1	0
4	5	18.051127	2012-06-29 16:51:54	15.022710	3	1	0
5	6	23.058702	2012-06-29 16:51:59	20.030285	3	1	0.
6	7	28.066285	2012-06-29 16:52:04	25.037868	3	1	0
7	8	33.073845	2012-06-29 16:52:09	30.045427	3	1	0
8	9	38.081417	2012-06-29 16:52:14	35.053000	3	1	0.
9	10	43.089009	2012-06-29 16:52:19	40.060591	3	1	0

In [2]:

```
low_curr_ocv_minus_10.describe()
```

Out[2]:

	Data_Point	Test_Time(s)	Date_Time	Step_Time(s)	Step_Index
count	29785.00000	29785.000000	29785	29785.000000	29785.000000
mean	14893.00000	79452.354943	2012-06-30 14:55:48.690951936	36718.816502	5.960987
min	1.00000	3.010944	2012-06-29 16:51:38	0.015765	1.000000
25%	7447.00000	40580.632806	2012-06-30 04:07:57	18022.251663	5.000000
50%	14893.00000	77867.013967	2012-06-30 14:29:23	36665.442255	5.000000
75%	22339.00000	118452.093694	2012-07-01 01:45:49	55308.632827	7.000000
max	29785.00000	159037.139119	2012-07-01 13:02:14	76334.569682	8.000000
std	8598.33322	44515.946548	NaN	21570.964248	1.013849

In [3]:

```
low_curr_ocv_minus_10.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 29785 entries, 0 to 29784
Data columns (total 19 columns):
 #   Column                     Non-Null Count  Dtype
---  ------                     --------------  -----
 0   Data_Point                 29785 non-null  int64
 1   Test_Time(s)               29785 non-null  float64
 2   Date_Time                  29785 non-null  datetime64[ns]
 3   Step_Time(s)               29785 non-null  float64
 4   Step_Index                 29785 non-null  int64
 5   Cycle_Index                29785 non-null  int64
 6   Current(A)                 29785 non-null  float64
 7   Voltage(V)                 29785 non-null  float64
 8   Charge_Capacity(Ah)        29785 non-null  float64
 9   Discharge_Capacity(Ah)     29785 non-null  float64
 10  Charge_Energy(Wh)          29785 non-null  float64
 11  Discharge_Energy(Wh)       29785 non-null  float64
 12  dV/dt(V/s)                 29785 non-null  float64
 13  Internal_Resistance(Ohm)   29785 non-null  int64
 14  Is_FC_Data                 29785 non-null  int64
 15  AC_Impedance(Ohm)          29785 non-null  int64
 16  ACI_Phase_Angle(Deg)       29785 non-null  int64
 17  Temperature (C)_1          29785 non-null  float64
 18  Temperature (C)_2          29785 non-null  float64
dtypes: datetime64[ns](1), float64(11), int64(7)
memory usage: 4.3 MB
```

LOW CURRENT OCV AT 0 degree celcius

In [4]:

In [4]:
```
low_curr_ocv_0=pd.read_excel(r"C:\Users\jaiku\PycharmProjects\Assignments\A
```

In [5]:
```
low_curr_ocv_0.head(10)
```

Out[5]:

	Data_Point	Test_Time(s)	Date_Time	Step_Time(s)	Step_Index	Cycle_Index	Cur
0	1	3.014942	2012-06-18 11:49:18	3.014942	1	1	0
1	2	3.046301	2012-06-18 11:49:18	0.031089	2	1	1.
2	3	8.054232	2012-06-18 11:49:23	5.007584	3	1	0
3	4	13.061941	2012-06-18 11:49:28	10.015293	3	1	0
4	5	18.073395	2012-06-18 11:49:33	15.026747	3	1	0.
5	6	23.080863	2012-06-18 11:49:38	20.034215	3	1	0
6	7	28.088504	2012-06-18 11:49:43	25.041856	3	1	0.
7	8	33.096150	2012-06-18 11:49:48	30.049501	3	1	0.
8	9	38.103670	2012-06-18 11:49:53	35.057022	3	1	0
9	10	43.111231	2012-06-18 11:49:58	40.064583	3	1	0

In [6]:
```
low_curr_ocv_0.describe()
```

Out[6]:

	Data_Point	Test_Time(s)	Date_Time	Step_Time(s)	Step_Index
count	30249.000000	30249.000000	30249	30249.000000	30249.000000
mean	15125.000000	80615.643879	2012-06-19 10:12:51.466792448	36962.274515	5.962048
min	1.000000	3.014942	2012-06-18 11:49:18	0.031089	1.000000

25%	7563.000000	41162.668955	2012-06-18 23:15:18	17997.213788	5.000000
50%	15125.000000	79029.928468	2012-06-19 09:46:26	36930.843545	5.000000
75%	22687.000000	120193.073943	2012-06-19 21:12:29	55864.473308	7.000000
max	30249.000000	161359.267642	2012-06-20 08:38:36	76186.533244	8.000000
std	8732.278483	45209.697396	NaN	21853.017650	1.030102

In [7]:

```
low_curr_ocv_0.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30249 entries, 0 to 30248
Data columns (total 19 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   Data_Point               30249 non-null  int64
 1   Test_Time(s)             30249 non-null  float64
 2   Date_Time                30249 non-null  datetime64[ns]
 3   Step_Time(s)             30249 non-null  float64
 4   Step_Index               30249 non-null  int64
 5   Cycle_Index              30249 non-null  int64
 6   Current(A)               30249 non-null  float64
 7   Voltage(V)               30249 non-null  float64
 8   Charge_Capacity(Ah)      30249 non-null  float64
 9   Discharge_Capacity(Ah)   30249 non-null  float64
 10  Charge_Energy(Wh)        30249 non-null  float64
 11  Discharge_Energy(Wh)     30249 non-null  float64
 12  dV/dt(V/s)               30249 non-null  float64
 13  Internal_Resistance(Ohm) 30249 non-null  int64
 14  Is_FC_Data               30249 non-null  int64
 15  AC_Impedance(Ohm)        30249 non-null  int64
 16  ACI_Phase_Angle(Deg)     30249 non-null  int64
 17  Temperature (C)_1        30249 non-null  float64
 18  Temperature (C)_2        30249 non-null  float64
dtypes: datetime64[ns](1), float64(11), int64(7)
memory usage: 4.4 MB
```

LOW CURRENT OCV AT 10 degree celcius

In [8]:

```
low_curr_ocv_10=pd.read_excel(r"C:\Users\jaiku\PycharmProjects\Assignments\
```

In [9]:

```
low_curr_ocv_10.head(10)
```

Out[9]:

	Data_Point	Test_Time(s)	Date_Time	Step_Time(s)	Step_Index	Cycle_Index	Cur
0	1	3.014079	2012-06-11	3.014079	1	1	0

			18:47:47				
1	2	3.025053	2012-06-11 18:47:47	0.000002	2	1	1.
2	3	8.032849	2012-06-11 18:47:52	5.007554	3	1	0.
3	4	13.040499	2012-06-11 18:47:57	10.015204	3	1	0.
4	5	18.046321	2012-06-11 18:48:02	15.021026	3	1	0.
5	6	23.053825	2012-06-11 18:48:07	20.028530	3	1	0.
6	7	28.061527	2012-06-11 18:48:12	25.036233	3	1	0.
7	8	33.068989	2012-06-11 18:48:17	30.043694	3	1	0.
8	9	38.076571	2012-06-11 18:48:22	35.051276	3	1	0.
9	10	43.084085	2012-06-11 18:48:27	40.058790	3	1	0.

In [10]:

```python
low_curr_ocv_10.describe()
```

Out[10]:

	Data_Point	Test_Time(s)	Date_Time	Step_Time(s)	Step_Index
count	31898.00000	31898.000000	31898	31898.000000	31898.000000
mean	15949.50000	84407.542774	2012-06-12 23:21:28.604269824	35034.642104	5.807605
min	1.00000	3.014079	2012-06-11 18:47:47	0.000002	1.000000
25%	7975.25000	43127.482188	2012-06-12 12:56:42.249999872	14612.052933	5.000000
50%	15949.50000	83059.113340	2012-06-13 00:02:14.500000	34527.490563	5.000000
75%	23923.75000	126287.349589	2012-06-13 12:02:42.750000128	54492.820953	7.000000
max	31898.00000	169513.642825	2012-06-14 00:03:10	75041.088878	8.000000

std	9208.30378	47856.153587	NaN	22318.590732	1.210257

In [11]:

```
low_curr_ocv_10.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 31898 entries, 0 to 31897
Data columns (total 19 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   Data_Point               31898 non-null  int64
 1   Test_Time(s)             31898 non-null  float64
 2   Date_Time                31898 non-null  datetime64[ns]
 3   Step_Time(s)             31898 non-null  float64
 4   Step_Index               31898 non-null  int64
 5   Cycle_Index              31898 non-null  int64
 6   Current(A)               31898 non-null  float64
 7   Voltage(V)               31898 non-null  float64
 8   Charge_Capacity(Ah)      31898 non-null  float64
 9   Discharge_Capacity(Ah)   31898 non-null  float64
 10  Charge_Energy(Wh)        31898 non-null  float64
 11  Discharge_Energy(Wh)     31898 non-null  float64
 12  dV/dt(V/s)               31898 non-null  float64
 13  Internal_Resistance(Ohm) 31898 non-null  int64
 14  Is_FC_Data               31898 non-null  int64
 15  AC_Impedance(Ohm)        31898 non-null  int64
 16  ACI_Phase_Angle(Deg)     31898 non-null  int64
 17  Temperature (C)_1        31898 non-null  float64
 18  Temperature (C)_2        31898 non-null  float64
dtypes: datetime64[ns](1), float64(11), int64(7)
memory usage: 4.6 MB
```

LOW CURRENT OCV AT 20 degree celcius

In [12]:

```
low_curr_ocv_20=pd.read_excel(r"C:\Users\jaiku\PycharmProjects\Assignments\
```

In [13]:

```
low_curr_ocv_20.head(10)
```

Out[13]:

	Data_Point	Test_Time(s)	Date_Time	Step_Time(s)	Step_Index	Cycle_Index	Cur
0	1	3.007810	2012-06-14 12:13:15	3.007810	1	1	0
1	2	3.008080	2012-06-14 12:13:15	0.000002	2	1	1.
2	3	8.015990	2012-06-14 12:13:20	5.007662	3	1	0
3	4	13.023481	2012-06-14	10.015153	3	1	0

			12:13:25				
4	5	18.035039	2012-06-14 12:13:30	15.026711	3	1	0
5	6	23.042624	2012-06-14 12:13:35	20.034296	3	1	0
6	7	28.050181	2012-06-14 12:13:40	25.041853	3	1	0
7	8	33.057714	2012-06-14 12:13:45	30.049386	3	1	0
8	9	38.065421	2012-06-14 12:13:50	35.057093	3	1	0
9	10	43.072973	2012-06-14 12:13:55	40.064645	3	1	0

In [14]:

```
low_curr_ocv_20.describe()
```

Out[14]:

	Data_Point	Test_Time(s)	Date_Time	Step_Time(s)	Step_Index
count	31018.000000	31018.000000	31018	31018.000000	31018.000000
mean	15509.500000	82530.938666	2012-06-15 11:08:43.815655168	37436.752180	5.955671
min	1.000000	3.007810	2012-06-14 12:13:15	0.000002	1.000000
25%	7755.250000	42125.027738	2012-06-14 23:55:17.249999872	17983.441759	5.000000
50%	15509.500000	80954.993049	2012-06-15 10:42:27.500000	37399.049136	5.000000
75%	23263.750000	123082.373990	2012-06-15 22:24:35.750000128	56814.656519	7.000000
max	31018.000000	165211.592285	2012-06-16 10:06:45	76708.858391	8.000000
std	8954.269661	46356.820098	NaN	22362.338814	1.053492

In [15]:

```
low_curr_ocv_20.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 31018 entries, 0 to 31017
Data columns (total 19 columns):
 #   Column                      Non-Null Count  Dtype
---  ------                      --------------  -----
```

```
 0   Data_Point               31018 non-null  int64
 1   Test_Time(s)             31018 non-null  float64
 2   Date_Time                31018 non-null  datetime64[ns]
 3   Step_Time(s)             31018 non-null  float64
 4   Step_Index               31018 non-null  int64
 5   Cycle_Index              31018 non-null  int64
 6   Current(A)               31018 non-null  float64
 7   Voltage(V)               31018 non-null  float64
 8   Charge_Capacity(Ah)      31018 non-null  float64
 9   Discharge_Capacity(Ah)   31018 non-null  float64
 10  Charge_Energy(Wh)        31018 non-null  float64
 11  Discharge_Energy(Wh)     31018 non-null  float64
 12  dV/dt(V/s)               31018 non-null  float64
 13  Internal_Resistance(Ohm) 31018 non-null  int64
 14  Is_FC_Data               31018 non-null  int64
 15  AC_Impedance(Ohm)        31018 non-null  int64
 16  ACI_Phase_Angle(Deg)     31018 non-null  int64
 17  Temperature (C)_1        31018 non-null  float64
 18  Temperature (C)_2        31018 non-null  float64
dtypes: datetime64[ns](1), float64(11), int64(7)
memory usage: 4.5 MB
```

LOW CURRENT OCV AT 25 degree celcius

In [16]:

```
low_curr_ocv_25=pd.read_excel(r"C:\Users\jaiku\PycharmProjects\Assignments\
```

In [17]:

```
low_curr_ocv_25.head(10)
```

Out[17]:

	Data_Point	Test_Time(s)	Date_Time	Step_Time(s)	Step_Index	Cycle_Index	Cur
0	1	3.003881	2012-09-05 09:50:31	3.003881	1	1	0.
1	2	8.004174	2012-09-05 09:50:36	5.000008	2	1	1.
2	3	13.004248	2012-09-05 09:50:41	10.000082	2	1	1.
3	4	18.004260	2012-09-05 09:50:46	15.000093	2	1	1.
4	5	23.005579	2012-09-05 09:50:51	20.001413	2	1	1.
5	6	28.005657	2012-09-05 09:50:56	25.001492	2	1	1.
6	7	33.011929	2012-09-05 09:51:01	30.007763	2	1	1.

7	8	38.011951	2012-09-05 09:51:06	35.007785	2	1	1
8	9	43.011997	2012-09-05 09:51:11	40.007831	2	1	1
9	10	48.012036	2012-09-05 09:51:16	45.007870	2	1	1

In [18]:

```
low_curr_ocv_25.describe()
```

Out[18]:

	Data_Point	Test_Time(s)	Date_Time	Step_Time(s)	Step_Index
count	32307.00000	32307.000000	32307	32307.000000	32307.000000
mean	16154.00000	85506.863991	2012-09-06 09:35:35.959884800	36302.546619	5.837713
min	1.00000	3.003881	2012-09-05 09:50:31	3.003881	1.000000
25%	8077.50000	43696.841249	2012-09-05 21:58:45.500000	15874.938811	5.000000
50%	16154.00000	84091.821468	2012-09-06 09:12:01	36076.586865	5.000000
75%	24230.50000	127800.461601	2012-09-06 21:20:30.500000	56272.784453	7.000000
max	32307.00000	171495.535618	2012-09-07 09:28:46	76592.209924	8.000000
std	9326.37191	48357.812231	NaN	22968.249246	1.217815

In [19]:

```
low_curr_ocv_25.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32307 entries, 0 to 32306
Data columns (total 18 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   Data_Point              32307 non-null  int64
 1   Test_Time(s)            32307 non-null  float64
 2   Date_Time               32307 non-null  datetime64[ns]
 3   Step_Time(s)            32307 non-null  float64
 4   Step_Index              32307 non-null  int64
 5   Cycle_Index             32307 non-null  int64
 6   Current(A)              32307 non-null  float64
 7   Voltage(V)              32307 non-null  float64
 8   Charge_Capacity(Ah)     32307 non-null  float64
 9   Discharge_Capacity(Ah)  32307 non-null  float64
 10  Charge_Energy(Wh)       32307 non-null  float64
 11  Discharge_Energy(Wh)    32307 non-null  float64
```

```
 12  dV/dt(V/s)                 32307 non-null  float64
 13  Internal_Resistance(Ohm)   32307 non-null  int64
 14  Is_FC_Data                 32307 non-null  int64
 15  AC_Impedance(Ohm)          32307 non-null  int64
 16  ACI_Phase_Angle(Deg)       32307 non-null  int64
 17  Temperature (C)_1          32307 non-null  float64
dtypes: datetime64[ns](1), float64(10), int64(7)
memory usage: 4.4 MB
```

LOW CURRENT OCV AT 30 degree celcius

In [20]:

```
low_curr_ocv_30=pd.read_excel(r"C:\Users\jaiku\PycharmProjects\Assignments\
```

In [21]:

```
low_curr_ocv_30.head(10)
```

Out[21]:

	Data_Point	Test_Time(s)	Date_Time	Step_Time(s)	Step_Index	Cycle_Index	Cur
0	1	3.010808	2012-06-25 10:58:13	3.010808	1	1	0
1	2	3.041952	2012-06-25 10:58:13	0.030896	2	1	1
2	3	8.057809	2012-06-25 10:58:18	5.015575	3	1	0
3	4	13.065362	2012-06-25 10:58:23	10.023128	3	1	0
4	5	18.072954	2012-06-25 10:58:28	15.030720	3	1	0
5	6	23.080509	2012-06-25 10:58:33	20.038275	3	1	0
6	7	28.088092	2012-06-25 10:58:38	25.045859	3	1	0
7	8	33.095648	2012-06-25 10:58:43	30.053414	3	1	0
8	9	38.103228	2012-06-25 10:58:48	35.060994	3	1	0
9	10	43.110795	2012-06-25 10:58:53	40.068561	3	1	0

In [22]:
```
low_curr_ocv_30.describe()
```

Out[22]:

	Data_Point	Test_Time(s)	Date_Time	Step_Time(s)	Step_Index
count	31150.000000	31150.000000	31150	31150.000000	31150.000000
mean	15575.500000	82879.119805	2012-06-26 09:59:30.199871488	37868.880320	5.967159
min	1.000000	3.010808	2012-06-25 10:58:13	0.030896	1.000000
25%	7788.250000	42289.364036	2012-06-25 22:43:00.249999872	18343.988166	5.000000
50%	15575.500000	81284.579150	2012-06-26 09:32:55.500000	37842.221660	5.000000
75%	23362.750000	123576.509714	2012-06-26 21:17:47.750000128	57340.455169	7.000000
max	31150.000000	165868.338984	2012-06-27 09:02:40	77269.004464	8.000000
std	8992.374779	46527.087207	NaN	22474.828502	1.040072

In [23]:
```
low_curr_ocv_30.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 31150 entries, 0 to 31149
Data columns (total 19 columns):
 #   Column                     Non-Null Count  Dtype
---  ------                     --------------  -----
 0   Data_Point                 31150 non-null  int64
 1   Test_Time(s)               31150 non-null  float64
 2   Date_Time                  31150 non-null  datetime64[ns]
 3   Step_Time(s)               31150 non-null  float64
 4   Step_Index                 31150 non-null  int64
 5   Cycle_Index                31150 non-null  int64
 6   Current(A)                 31150 non-null  float64
 7   Voltage(V)                 31150 non-null  float64
 8   Charge_Capacity(Ah)        31150 non-null  float64
 9   Discharge_Capacity(Ah)     31150 non-null  float64
 10  Charge_Energy(Wh)          31150 non-null  float64
 11  Discharge_Energy(Wh)       31150 non-null  float64
 12  dV/dt(V/s)                 31150 non-null  float64
 13  Internal_Resistance(Ohm)   31150 non-null  int64
 14  Is_FC_Data                 31150 non-null  int64
 15  AC_Impedance(Ohm)          31150 non-null  int64
 16  ACI_Phase_Angle(Deg)       31150 non-null  int64
 17  Temperature (C)_1          31150 non-null  float64
 18  Temperature (C)_2          31150 non-null  float64
dtypes: datetime64[ns](1), float64(11), int64(7)
memory usage: 4.5 MB
```

LOW CURRENT OCV AT 40 degree

celcius

In [24]:
```
low_curr_ocv_40=pd.read_excel(r"C:\Users\jaiku\PycharmProjects\Assignments\
```

In [25]:
```
low_curr_ocv_40.head(10)
```

Out[25]:

	Data_Point	Test_Time(s)	Date_Time	Step_Time(s)	Step_Index	Cycle_Index	Cur
0	1	3.010283	2012-06-27 11:43:25	3.010283	1	1	0
1	2	3.026131	2012-06-27 11:43:25	0.015609	2	1	1
2	3	8.033910	2012-06-27 11:43:30	5.007549	3	1	0
3	4	13.041500	2012-06-27 11:43:35	10.015140	3	1	0
4	5	18.049071	2012-06-27 11:43:40	15.022711	3	1	0
5	6	23.056643	2012-06-27 11:43:45	20.030282	3	1	0
6	7	28.064215	2012-06-27 11:43:50	25.037854	3	1	0
7	8	33.071785	2012-06-27 11:43:55	30.045425	3	1	0
8	9	38.079286	2012-06-27 11:44:00	35.052926	3	1	0
9	10	43.086932	2012-06-27 11:44:05	40.060571	3	1	0

In [26]:
```
low_curr_ocv_40.describe()
```

Out[26]:

	Data_Point	Test_Time(s)	Date_Time	Step_Time(s)	Step_Index
count	31258.000000	31258.000000	31258	31258.000000	31258.000000
mean	15629.500000	83182.146533	2012-06-28 10:49:45.445390080	38403.674272	5.985124

min	1.000000	3.010283	2012-06-27 11:43:25	0.015609	1.000000
25%	7815.250000	42427.009169	2012-06-27 23:30:30.249999872	18823.463111	5.000000
50%	15629.500000	81557.428742	2012-06-28 10:22:40.500000	38388.046949	5.000000
75%	23443.750000	123985.717499	2012-06-28 22:09:49.750000128	57952.630847	7.000000
max	31258.000000	166414.916638	2012-06-29 09:56:59	77785.938692	8.000000
std	9023.551694	46653.945691	NaN	22568.608281	1.019973

In [27]:

```
low_curr_ocv_40.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 31258 entries, 0 to 31257
Data columns (total 19 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Data_Point                31258 non-null  int64
 1   Test_Time(s)              31258 non-null  float64
 2   Date_Time                 31258 non-null  datetime64[ns]
 3   Step_Time(s)              31258 non-null  float64
 4   Step_Index                31258 non-null  int64
 5   Cycle_Index               31258 non-null  int64
 6   Current(A)                31258 non-null  float64
 7   Voltage(V)                31258 non-null  float64
 8   Charge_Capacity(Ah)       31258 non-null  float64
 9   Discharge_Capacity(Ah)    31258 non-null  float64
 10  Charge_Energy(Wh)         31258 non-null  float64
 11  Discharge_Energy(Wh)      31258 non-null  float64
 12  dV/dt(V/s)                31258 non-null  float64
 13  Internal_Resistance(Ohm)  31258 non-null  int64
 14  Is_FC_Data                31258 non-null  int64
 15  AC_Impedance(Ohm)         31258 non-null  int64
 16  ACI_Phase_Angle(Deg)      31258 non-null  int64
 17  Temperature (C)_1         31258 non-null  float64
 18  Temperature (C)_2         31258 non-null  float64
dtypes: datetime64[ns](1), float64(11), int64(7)
memory usage: 4.5 MB
```

LOW CURRENT OCV AT 50 degree celcius

In [28]:

```
low_curr_ocv_50=pd.read_excel(r"C:\Users\jaiku\PycharmProjects\Assignments\
```

In [29]:

```
low_curr_ocv_50.head(10)
```

Out[29]:

Data_Point	Test_Time(s)	Date_Time	Step_Time(s)	Step_Index	Cycle_Index	Cur

0	1	3.008714	2012-07-02 10:52:00	3.008714	1	1	0
1	2	3.020297	2012-07-02 10:52:00	0.000002	2	1	1
2	3	8.037763	2012-07-02 10:52:05	5.007566	3	1	0
3	4	13.045325	2012-07-02 10:52:10	10.015128	3	1	0
4	5	18.052910	2012-07-02 10:52:15	15.022712	3	1	0
5	6	23.060484	2012-07-02 10:52:20	20.030287	3	1	0
6	7	28.068046	2012-07-02 10:52:25	25.037848	3	1	0
7	8	33.075622	2012-07-02 10:52:30	30.045425	3	1	0
8	9	38.083297	2012-07-02 10:52:35	35.053100	3	1	0
9	10	43.090758	2012-07-02 10:52:40	40.060561	3	1	0

In [30]:

```
low_curr_ocv_50.describe()
```

Out[30]:

	Data_Point	Test_Time(s)	Date_Time	Step_Time(s)	Step_Index
count	31475.000000	31475.000000	31475	31475.000000	31475.000000
mean	15738.000000	83704.689997	2012-07-03 10:07:02.332041216	38368.988081	5.975473
min	1.000000	3.008714	2012-07-02 10:52:00	0.000002	1.000000
25%	7869.500000	42695.210585	2012-07-02 22:43:32.500000	18645.694362	5.000000
50%	15738.000000	82097.290906	2012-07-03 09:40:15	38347.986437	5.000000
75%	23606.500000	124794.372180	2012-07-03 21:31:52.500000	58047.774699	7.000000

max	31475.000000	167494.127364	2012-07-04 09:23:33	77843.159249	8.000000
std	9086.194198	46988.606063	NaN	22713.080086	1.035112

In [31]:

```
low_curr_ocv_50.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 31475 entries, 0 to 31474
Data columns (total 19 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   Data_Point               31475 non-null  int64
 1   Test_Time(s)             31475 non-null  float64
 2   Date_Time                31475 non-null  datetime64[ns]
 3   Step_Time(s)             31475 non-null  float64
 4   Step_Index               31475 non-null  int64
 5   Cycle_Index              31475 non-null  int64
 6   Current(A)               31475 non-null  float64
 7   Voltage(V)               31475 non-null  float64
 8   Charge_Capacity(Ah)      31475 non-null  float64
 9   Discharge_Capacity(Ah)   31475 non-null  float64
 10  Charge_Energy(Wh)        31475 non-null  float64
 11  Discharge_Energy(Wh)     31475 non-null  float64
 12  dV/dt(V/s)               31475 non-null  float64
 13  Internal_Resistance(Ohm) 31475 non-null  int64
 14  Is_FC_Data               31475 non-null  int64
 15  AC_Impedance(Ohm)        31475 non-null  int64
 16  ACI_Phase_Angle(Deg)     31475 non-null  int64
 17  Temperature (C)_1        31475 non-null  float64
 18  Temperature (C)_2        31475 non-null  float64
dtypes: datetime64[ns](1), float64(11), int64(7)
memory usage: 4.6 MB
```

DYNAMIC TEST PROFILES

Dynamic test profile at -10 degree celcius

In [32]:

```
dst_minus_10=pd.read_excel(r"C:\Users\jaiku\PycharmProjects\Assignments\A12
```

In [33]:

```
dst_minus_10.head(10)
```

Out[33]:

	Test_Time(s)	Step_Time(s)	Step_Index	Current(A)	Voltage(V)	Temperature (C)_1
0	3.003816	3.003816	1	0.000000	2.803019	-8.300812
1	8.004204	5.000077	2	-1.099373	2.109292	-8.145514
2	12.006524	9.002396	2	-1.099373	1.999416	-8.145514
3	17.013409	5.003358	3	0.000000	2.671906	-8.145514
4	22.015560	10.005508	3	0.000000	2.689757	-8.223158

5	27.017961	15.007909	3	0.000000	2.697759	-8.191391
6	32.018054	20.008003	3	0.000000	2.702376	-8.223158
7	37.026317	25.016265	3	0.000000	2.706993	-8.113752
8	42.030830	30.020778	3	0.000000	2.709763	-8.067879
9	47.039970	35.029918	3	0.000000	2.713148	-8.223158

In [34]:

```
dst_minus_10.describe()
```

Out[34]:

	Test_Time(s)	Step_Time(s)	Step_Index	Current(A)	Voltage(V)	Tem
count	19784.000000	1.978400e+04	19784.000000	19784.000000	19784.000000	1978
mean	20198.292967	8.537425e+02	15.393095	-0.322788	3.165822	-
std	9740.553386	1.314745e+03	6.824850	0.887062	0.304046	
min	3.003816	3.849596e-07	1.000000	-3.849595	1.999416	-
25%	11169.170425	1.615422e+02	8.000000	-0.758130	2.985145	-
50%	20727.640514	3.380252e+02	16.000000	-0.008637	3.164809	-
75%	30207.416255	8.841617e+02	24.000000	0.134114	3.347782	-
max	35460.091077	7.315288e+03	27.000000	2.061261	3.814677	-

In [35]:

```
dst_minus_10.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 19784 entries, 0 to 19783
Data columns (total 6 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   Test_Time(s)       19784 non-null  float64
 1   Step_Time(s)       19784 non-null  float64
 2   Step_Index         19784 non-null  int64  
 3   Current(A)         19784 non-null  float64
 4   Voltage(V)         19784 non-null  float64
 5   Temperature (C)_1  19784 non-null  float64
dtypes: float64(5), int64(1)
memory usage: 927.5 KB
```

Dynamic test profile at 0 degree celcius

In [36]:

```
dst_0=pd.read_excel(r"C:\Users\jaiku\PycharmProjects\Assignments\A123 batte
```

In [37]:

```
dst_0.head(10)
```

Out[37]:

	Test_Time(s)	Step_Time(s)	Step_Index	Current(A)	Voltage(V)	Temperature (C)_1

0	3.057882	3.005927	1	0.000000	2.846107	1.256174
1	8.065274	5.007092	2	-1.099185	2.468466	1.256174
2	13.072544	10.014362	2	-1.099373	2.403833	1.332009
3	18.077826	15.019644	2	-1.099373	2.333968	1.256174
4	23.078087	20.019905	2	-1.099373	2.250561	1.301033
5	28.082401	25.024220	2	-1.099560	2.134222	1.452703
6	31.764519	28.706338	2	-1.099373	1.999416	1.345893
7	36.774210	5.008228	3	0.000000	2.548488	1.332009
8	41.783489	10.017507	3	0.000000	2.566955	1.376867
9	46.792691	15.026708	3	0.000000	2.577727	1.332009

In [38]:

```
dst_0.describe()
```

Out[38]:

	Test_Time(s)	Step_Time(s)	Step_Index	Current(A)	Voltage(V)	Tem
count	23248.000000	2.324800e+04	23248.000000	23248.000000	23248.000000	2324
mean	20466.229194	5.328935e+02	15.566027	-0.348093	3.165784	
std	10301.715583	6.231313e+02	6.763405	0.918652	0.259291	
min	3.057882	7.699193e-07	1.000000	-3.849219	1.961867	-
25%	10658.417985	1.537679e+02	8.000000	-0.829459	3.026156	
50%	20792.540701	3.166302e+02	16.000000	-0.191961	3.172349	
75%	30805.391681	6.670666e+02	24.000000	0.080958	3.293613	
max	36926.114253	4.328226e+03	27.000000	2.060698	3.728192	

In [39]:

```
dst_0.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 23248 entries, 0 to 23247
Data columns (total 6 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   Test_Time(s)       23248 non-null  float64
 1   Step_Time(s)       23248 non-null  float64
 2   Step_Index         23248 non-null  int64  
 3   Current(A)         23248 non-null  float64
 4   Voltage(V)         23248 non-null  float64
 5   Temperature (C)_1  23248 non-null  float64
dtypes: float64(5), int64(1)
memory usage: 1.1 MB
```

Dynamic test profile at 10 degree celcius

In [40]:

```
dst_10=pd.read_excel(r"C:\Users\jaiku\PycharmProjects\Assignments\A123 batt
```

In [41]:
```
dst_10.head(10)
```

Out[41]:

	Test_Time(s)	Step_Time(s)	Step_Index	Current(A)	Voltage(V)	Temperature (C)_1
0	3.008291	3.008291	1	0.000000	2.968602	9.646121
1	8.015758	5.007161	2	-1.099185	2.729152	9.690903
2	13.018937	10.010341	2	-1.099373	2.703299	9.598107
3	18.027239	15.018642	2	-1.099373	2.678369	9.797547
4	23.031320	20.022724	2	-1.099373	2.654363	9.766615
5	28.035736	25.027140	2	-1.099373	2.629125	9.721836
6	33.039112	30.030516	2	-1.099373	2.603272	9.797547
7	38.047386	35.038790	2	-1.099373	2.576496	9.690903
8	43.053577	40.044981	2	-1.099373	2.547873	9.918034
9	48.062643	45.054046	2	-1.099373	2.515864	9.873258

In [42]:
```
dst_10.describe()
```

Out[42]:

	Test_Time(s)	Step_Time(s)	Step_Index	Current(A)	Voltage(V)	Tem
count	23881.000000	2.388100e+04	23881.000000	23881.000000	23881.000000	2388
mean	19919.260228	4.755559e+02	15.518613	-0.356659	3.190354	1
std	10203.319502	4.779053e+02	6.705619	0.932044	0.224014	
min	3.008291	7.699193e-07	1.000000	-3.849407	1.987413	
25%	10199.540085	1.497195e+02	8.000000	-0.848430	3.077862	1
50%	20191.833861	3.095074e+02	16.000000	-0.243238	3.196048	1
75%	30156.128886	5.997450e+02	24.000000	0.072130	3.302231	1
max	36435.385542	2.681824e+03	27.000000	2.061261	3.699877	1

In [43]:
```
dst_10.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 23881 entries, 0 to 23880
Data columns (total 6 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   Test_Time(s)       23881 non-null  float64
 1   Step_Time(s)       23881 non-null  float64
 2   Step_Index         23881 non-null  int64  
 3   Current(A)         23881 non-null  float64
 4   Voltage(V)         23881 non-null  float64
```

```
 5   Temperature (C)_1  23881 non-null  float64
dtypes: float64(5), int64(1)
memory usage: 1.1 MB
```

Dynamic test profile at 20 degree celcius

In [44]:

```
dst_20=pd.read_excel(r"C:\Users\jaiku\PycharmProjects\Assignments\A123 batt
```

In [45]:

```
dst_20.head(10)
```

Out[45]:

	Test_Time(s)	Step_Time(s)	Step_Index	Current(A)	Voltage(V)	Temperature (C)_1
0	3.008332	3.008332	1	0.000000	3.108637	18.427523
1	8.014952	5.006227	2	-1.099776	2.891659	18.444551
2	13.018221	10.009497	2	-1.099776	2.877501	18.427523
3	18.022437	15.013712	2	-1.099957	2.864575	18.489090
4	23.026696	20.017971	2	-1.099776	2.851957	18.382978
5	28.035930	25.027205	2	-1.099957	2.839954	18.458305
6	33.045322	30.036597	2	-1.099776	2.827950	18.365955
7	38.051520	35.042795	2	-1.099957	2.815948	18.427523
8	43.066605	40.057881	2	-1.099776	2.803944	18.352194
9	48.077106	45.068381	2	-1.099776	2.791941	18.321405

In [46]:

```
dst_20.describe()
```

Out[46]:

	Test_Time(s)	Step_Time(s)	Step_Index	Current(A)	Voltage(V)	Tem
count	24992.000000	2.499200e+04	24992.000000	24992.000000	24992.000000	2499
mean	20478.506604	4.711195e+02	15.573944	-0.362114	3.195718	2
std	10524.199546	4.800529e+02	6.722357	0.935984	0.229664	
min	3.008332	7.699193e-07	1.000000	-3.849713	1.898483	1
25%	10446.853748	1.524163e+02	8.000000	-0.862705	3.091094	1
50%	20715.938621	3.099327e+02	16.000000	-0.279794	3.208663	1
75%	30962.624857	5.888735e+02	24.000000	0.063971	3.315767	2
max	37521.292897	2.867548e+03	27.000000	2.061054	3.732181	2

In [47]:

```
dst_20.info()
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
RangeIndex: 24992 entries, 0 to 24991
Data columns (total 6 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   Test_Time(s)       24992 non-null  float64
 1   Step_Time(s)       24992 non-null  float64
 2   Step_Index         24992 non-null  int64  
 3   Current(A)         24992 non-null  float64
 4   Voltage(V)         24992 non-null  float64
 5   Temperature (C)_1  24992 non-null  float64
dtypes: float64(5), int64(1)
memory usage: 1.1 MB
```

Dynamic test profile at 25 degree celcius

In [48]:

```
dst_25=pd.read_excel(r"C:\Users\jaiku\PycharmProjects\Assignments\A123 batt
```

In [49]:

```
dst_25.head(10)
```

Out[49]:

	Test_Time(s)	Step_Time(s)	Step_Index	Current(A)	Voltage(V)	Temperature (C)_1
0	3.053504	3.000204	1	0.000000	2.766393	26.788187
1	8.061154	5.007273	2	-1.099373	2.501398	26.818743
2	13.062293	10.008411	2	-1.099185	2.435226	26.818743
3	18.071620	15.017739	2	-1.099373	2.340432	26.713451
4	23.073773	20.019892	2	-1.099185	2.115448	26.788187
5	24.301853	21.247971	2	-1.099373	1.998801	26.893473
6	29.307341	5.004141	3	0.000000	2.460464	26.893473
7	34.311587	10.008388	3	0.000000	2.474314	26.713451
8	39.319818	15.016618	3	0.000000	2.482316	26.818743
9	44.325095	20.021896	3	0.000000	2.486933	26.937641

In [50]:

```
dst_25.describe()
```

Out[50]:

	Test_Time(s)	Step_Time(s)	Step_Index	Current(A)	Voltage(V)	Tem
count	24469.000000	2.446900e+04	24469.000000	24469.000000	24469.000000	2446
mean	19726.353242	4.718362e+02	15.557767	-0.363447	3.204312	2
std	10192.971787	5.018513e+02	6.710170	0.942908	0.211759	
min	3.053504	3.849596e-07	1.000000	-3.849407	1.937861	2
25%	10017.592323	1.497245e+02	8.000000	-0.867589	3.110487	2
50%	19958.708627	3.061075e+02	16.000000	-0.286440	3.214515	2
75%	29860.944817	5.805784e+02	24.000000	0.060672	3.311772	2

max	36294.795004	3.070756e+03	27.000000	2.061261	3.699877	2

In [51]:

```
dst_25.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 24469 entries, 0 to 24468
Data columns (total 6 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   Test_Time(s)        24469 non-null  float64
 1   Step_Time(s)        24469 non-null  float64
 2   Step_Index          24469 non-null  int64  
 3   Current(A)          24469 non-null  float64
 4   Voltage(V)          24469 non-null  float64
 5   Temperature (C)_1  24469 non-null  float64
dtypes: float64(5), int64(1)
memory usage: 1.1 MB
```

Dynamic test profile at 30 degree celcius

In [52]:

```
dst_30=pd.read_excel(r"C:\Users\jaiku\PycharmProjects\Assignments\A123 batt
```

In [53]:

```
dst_30.head(10)
```

Out[53]:

	Test_Time(s)	Step_Time(s)	Step_Index	Current(A)	Voltage(V)	Temperature (C)_1
0	3.004871	3.004871	1	0.000000	3.075400	31.391201
1	8.011347	5.006164	2	-1.099373	2.888580	31.242373
2	13.017574	10.012391	2	-1.099373	2.876269	31.391201
3	18.023831	15.018647	2	-1.099373	2.864882	31.421654
4	23.027999	20.022816	2	-1.099373	2.853494	31.496058
5	28.032410	25.027227	2	-1.099373	2.842414	31.421654
6	33.039675	30.034492	2	-1.099373	2.831334	31.421654
7	38.041781	35.036597	2	-1.099373	2.819946	31.465605
8	43.043244	40.038061	2	-1.099373	2.808558	31.465605
9	48.043298	45.038114	2	-1.099560	2.796555	31.391201

In [54]:

```
dst_30.describe()
```

Out[54]:

	Test_Time(s)	Step_Time(s)	Step_Index	Current(A)	Voltage(V)	Tem
count	24598.000000	2.459800e+04	24598.000000	24598.000000	24598.000000	2459
mean	19621.906470	4.721762e+02	15.567973	-0.364780	3.206956	3

std	10170.231343	5.126144e+02	6.716462	0.942931	0.203930	
min	3.004871	3.849596e-07	1.000000	-3.849407	1.949556	3
25%	9938.020160	1.497872e+02	8.000000	-0.871909	3.116026	3
50%	19831.893588	3.053628e+02	16.000000	-0.297146	3.217592	3
75%	29711.598095	5.757773e+02	24.000000	0.049966	3.312695	3
max	36178.975170	3.143676e+03	27.000000	2.061261	3.696491	3

In [55]:

```
dst_30.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 24598 entries, 0 to 24597
Data columns (total 6 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Test_Time(s)    24598 non-null  float64
 1   Step_Time(s)    24598 non-null  float64
```

master Machine_Learning_and_Alogorithms / 3.Feature Engineering / 05-29-2025 Feature Engineering.ipynb Top

Preview Code Blame Raw

Dynamic test profile at 50 degree celcius

In [56]:

```
dst_50=pd.read_excel(r"C:\Users\jaiku\PycharmProjects\Assignments\A123 batt
```

In [57]:

```
dst_50.head(10)
```

Out[57]:

	Test_Time(s)	Step_Time(s)	Step_Index	Current(A)	Voltage(V)	Temperature (C)_1
0	3.007451	3.007451	1	0.000000	2.775626	50.789497
1	8.012196	5.004451	2	-1.099373	2.564185	50.789497
2	13.012202	10.004457	2	-1.099373	2.523558	50.789497
3	18.014646	15.006900	2	-1.099373	2.473083	50.746269
4	23.021725	20.013979	2	-1.099373	2.404141	50.879433
5	28.021949	25.014203	2	-1.099185	2.257025	50.952637
6	30.994169	27.986424	2	-1.099185	1.999724	50.746269
7	35.996811	5.001098	3	0.000000	2.367208	50.836208
8	41.000069	10.004355	3	0.000000	2.385059	50.866188
9	46.000322	15.004608	3	0.000000	2.393369	50.819477

In [58]:
```python
dst_50.describe()
```

Out[58]:

	Test_Time(s)	Step_Time(s)	Step_Index	Current(A)	Voltage(V)	Tem
count	24763.000000	2.476300e+04	24763.000000	24763.000000	24763.000000	2476
mean	19197.121372	4.804346e+02	15.652263	-0.367141	3.214068	5
std	10034.610605	5.380568e+02	6.695618	0.947778	0.195042	
min	3.007451	3.849596e-07	1.000000	-3.849407	1.926166	5
25%	9617.456118	1.515973e+02	8.000000	-0.880455	3.126183	5
50%	19349.539252	3.072322e+02	16.000000	-0.311609	3.226826	5
75%	29076.097827	5.734703e+02	24.000000	0.034939	3.320697	5
max	35582.345606	3.301138e+03	27.000000	2.061261	3.709726	5

In [59]:
```python
dst_50.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 24763 entries, 0 to 24762
Data columns (total 6 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   Test_Time(s)       24763 non-null  float64
 1   Step_Time(s)       24763 non-null  float64
 2   Step_Index         24763 non-null  int64  
 3   Current(A)         24763 non-null  float64
 4   Voltage(V)         24763 non-null  float64
 5   Temperature (C)_1  24763 non-null  float64
dtypes: float64(5), int64(1)
memory usage: 1.1 MB
```

Question 1: How Do We Handle Categorical Features in

Battery Data?

Step 1: Understanding the Categorical Nature of Temperature

Temperature in our dataset represents discrete experimental conditions, not continuous thermal measurements. This distinction is crucial for proper feature engineering.

Step 2: Implementing One-Hot Encoding for Temperature

Encoding for Temperature

One-hot encoding transforms categorical variables into binary indicator variables, allowing models to treat each category independently.

In [60]:

```
import pandas as pd
import numpy as np
from sklearn.feature_extraction import DictVectorizer
from sklearn.preprocessing import OneHotEncoder

# Temperature color mapping for consistent visualization
temp_colors = {
    '-10': '#0033A0',  # Deep blue
    '0': '#0066CC',    # Blue
    '10': '#3399FF',   # Light blue
    '20': '#66CC00',   # Green
    '25': '#FFCC00',   # Yellow (room temperature)
    '30': '#FF9900',   # Orange
    '40': '#FF6600',   # Dark orange
    '50': '#CC0000'    # Red
}

# Temperature datasets organization
temp_datasets = {
    '-10°C': low_curr_ocv_minus_10,  # ~29,785 samples
    '0°C': low_curr_ocv_0,           # ~30,249 samples
    '10°C': low_curr_ocv_10,         # ~31,898 samples
    '20°C': low_curr_ocv_20,         # ~31,018 samples
    '25°C': low_curr_ocv_25,         # ~32,307 samples
    '30°C': low_curr_ocv_30,         # ~31,150 samples
    '40°C': low_curr_ocv_40,         # ~31,258 samples
    '50°C': low_curr_ocv_50          # ~31,475 samples
}
```

In [61]:

```
def create_temperature_categorical_features():
    """
    Convert temperature conditions into categorical features
    using one-hot encoding approach from feature engineering principles.
    """
    # Combine all datasets with temperature labels
    combined_data = []

    for temp_label, dataset in temp_datasets.items():
        # Create temperature category for each sample
        temp_data = dataset.copy()
        temp_data['Temperature_Category'] = temp_label
        temp_data['Temperature_Numeric'] = float(temp_label.replace('°C', '
        combined_data.append(temp_data)

    # Concatenate all temperature datasets
    full_dataset = pd.concat(combined_data, ignore_index=True)

    # One-hot encode temperature categories
    temp_encoder = OneHotEncoder(sparse_output=False, dtype=int)
    temp_categories = full_dataset[['Temperature_Category']]

    # Transform to one-hot encoded features
    temp_encoded = temp_encoder.fit_transform(temp_categories)
    temp_feature_names = temp_encoder.get_feature_names_out(['Temperature_C

    # Create DataFrame with encoded features
```

```
    temp_encoded_df = pd.DataFrame(temp_encoded, columns=temp_feature_names

    return full_dataset, temp_encoded_df, temp_encoder

# Example implementation
full_dataset, temp_encoded_df, temp_encoder = create_temperature_categorica

print("Temperature One-Hot Encoding Results:")
temp_encoded_df.head()
print(f"\nFeature Names: {temp_encoder.get_feature_names_out(['Temperature_
```

```
Temperature One-Hot Encoding Results:

Feature Names: ['Temperature_Category_-10°C' 'Temperature_Category_0°C'
 'Temperature_Category_10°C' 'Temperature_Category_20°C'
 'Temperature_Category_25°C' 'Temperature_Category_30°C'
 'Temperature_Category_40°C' 'Temperature_Category_50°C']
```

In [61]:

Step 3: Handling Step Index and Cycle Index as Categories

Battery test protocols involve multiple phases (steps) and repetitions (cycles) that should be treated categorically.

In [62]:

```
def create_step_categorical_features(dataset):
    """
    Encode step indices as categorical features to capture
    different testing phases.
    """
    # Create step category dictionary
    step_data = []

    for idx, row in dataset.iterrows():
        step_dict = {
            'step_index': row['Step_Index'],
            'cycle_index': row['Cycle_Index'],
            'voltage': row['Voltage(V)'],
            'current': row['Current(A)']
        }
        step_data.append(step_dict)

    # Use DictVectorizer for automatic categorical encoding
    dict_vectorizer = DictVectorizer(sparse=False, dtype=int)
    step_encoded = dict_vectorizer.fit_transform(step_data)

    # Get feature names
    feature_names = dict_vectorizer.get_feature_names_out()

    return step_encoded, feature_names, dict_vectorizer

# Example for -10°C dataset
step_encoded, step_features, step_vectorizer = create_step_categorical_feat
    low_curr_ocv_minus_10
)

print("Step Categorical Features Sample:")
```

```
print(f"Features: {step_features}")
print(f"Encoded shape: {step_encoded.shape}")
```

```
Step Categorical Features Sample:
Features: ['current' 'cycle_index' 'step_index' 'voltage']
Encoded shape: (29785, 4)
```

Okay, this describes a common and very useful process in machine learning, especially when dealing with experimental data like battery cycling tests. Here's a breakdown of what the program and the `create_step_categorical_features` function are doing, presented in Markdown:

Program for Creating Categorical Features from Battery Cycling Data

This Python function, `create_step_categorical_features`, is designed to process battery cycling data and transform it into a numerical format suitable for machine learning models. It specifically focuses on handling categorical information derived from different testing steps and cycles, along with continuous numerical measurements.

How the `create_step_categorical_features` Function Works:

1. **Input Data:**

    - The function takes a battery dataset (likely a pandas DataFrame or similar structure) as input.

2. **Row-wise Feature Extraction:**

    - For each row (representing a data point or measurement) in the input dataset, it extracts four key values:
        - **`Step_Index`**: This likely identifies different phases or specific steps within a battery testing protocol (e.g., 'charge_CC', 'discharge_CV', 'rest', 'OCV_measurement_step_1', etc.). This is treated as a categorical variable.
        - **`Cycle_Index`**: This indicates the specific charge/discharge cycle number the data point belongs to (e.g., cycle 1, cycle 2, etc.). This is also treated as a categorical variable.
        - **`Voltage(V)`**: The measured voltage at that data point (a numerical variable).
        - **`Current(A)`**: The measured current at that data point (a numerical variable).

3. **Data Restructuring to Dictionaries:**

    - These four extracted values from each row are then converted into a **list of Python dictionaries**. Each dictionary in the list represents one row of data, with keys like `'Step_Index'`, `'Cycle_Index'`, `'Voltage(V)'`, and `'Current(A)'` and their corresponding values.

`Current(A)` and their corresponding values.

  - Example: `[{'Step_Index': 'charge_CC', 'Cycle_Index': 1, 'Voltage(V)': 3.8, 'Current(A)': 1.0}, {'Step_Index': 'discharge_CV', ...}, ...]`

4. **Feature Encoding with `DictVectorizer`:**

   - The core of the transformation is done using `sklearn.feature_extraction.DictVectorizer`. This tool is powerful for converting lists of feature-value dictionaries into NumPy arrays that machine learning algorithms can use.
   - `DictVectorizer` automatically handles the features as follows:
     - **Categorical Features (`Step_Index` , `Cycle_Index`):** It performs **one-hot encoding** (or dummy coding) on these. For each unique value found in `Step_Index` and `Cycle_Index`, a new binary feature (column) is created. For a given row, the column corresponding to its `Step_Index` value will be 1, and all other `Step_Index` columns will be 0 (and similarly for `Cycle_Index`).
     - **Numerical Features (`Voltage(V)` , `Current(A)`):** These are preserved as is and will form their own columns in the output numerical matrix.

5. **Output:**

   - The function returns three important pieces of information:
     - **The encoded feature matrix (e.g., `X_vectorized`):** This is a numerical array where rows are the original data samples and columns are the newly created one-hot encoded features and the original numerical features.
     - **The feature names (e.g., `feature_names`):** A list of names for all the columns in the encoded feature matrix. This helps in understanding what each column represents (e.g., `Voltage(V)`, `Current(A)`, `Step_Index=charge_CC`, `Cycle_Index=1`, etc.).
     - **The fitted vectorizer (e.g., `vectorizer`):** The `DictVectorizer` object that has been `fit` (trained) on the input data. This is crucial because it stores the mapping from categorical values to feature indices and can be used to apply the *exact same transformation* to new, unseen data (like a test set), ensuring consistency.

Example Application:

- The description mentions an example where this function is applied to a specific dataset named `low_curr_ocv_minus_10`.
- This indicates the process is being used on battery data collected under particular experimental conditions: low current, likely during Open-Circuit Voltage (OCV) measurements, and specifically at a temperature of -10°C.

Purpose and Utility in Battery Analysis and Machine Learning:

- **Numerical Input Requirement:** Most machine learning algorithms (e.g., linear

regression, support vector machines, neural networks) require input data to be purely numerical. This function addresses that by converting categorical step and cycle information into a numerical format.
- **Distinguishing Operational Phases:** By one-hot encoding `Step_Index` and `Cycle_Index` , the model can learn to distinguish between different phases of the battery test (e.g., charging vs. discharging, early cycles vs. late cycles) and potentially model how battery behavior (like voltage or capacity degradation) differs across these phases.
- **Preventing Ordinal Misinterpretation:** Treating `Step_Index` or `Cycle_Index` as simple numerical inputs (e.g., 1, 2, 3) might imply an ordinal relationship or a specific magnitude difference between them that doesn't exist or isn't appropriate. One-hot encoding avoids this by treating each category independently.
- **Reproducibility:** Returning the `fitted_vectorizer` is key for building robust machine learning pipelines, as it ensures that any new data processed for prediction undergoes the same feature transformation as the training data.

In essence, this code provides a standard and effective way to preprocess structured experimental data, making it suitable for sophisticated machine learning analysis by appropriately handling categorical identifiers within a larger set of numerical measurements.

You've hit on a very important nuance of how `DictVectorizer` works! Your analysis of the output is **spot on**.

Based on the output you've described:

- **Features:** `['current', 'cycle_index', 'step_index', 'voltage']`
- **Encoded shape:** `(29785, 4)`

Here's a breakdown of what this means and why `DictVectorizer` behaved this way:

1. **No Expansion of Features:**
    - You started with 4 conceptual pieces of information per row (`current` , `cycle_index` , `step_index` , `voltage`).
    - The output matrix still has exactly 4 columns (features).
    - This is the key indicator that one-hot encoding for `step_index` and `cycle_index` **did not occur**. If it had, the number of columns would have increased significantly (e.g., if there were 5 unique `step_index` values and 10 unique `cycle_index` values, you'd expect many more columns than 4).
2. **`DictVectorizer` 's Default Behavior:**
    - You are correct: **By default, `DictVectorizer` performs one-hot encoding *only* on feature values that are strings.**
    - If `cycle_index` and `step_index` in your input dictionaries were already numerical (e.g., integers like `1, 2, 3...` for `cycle_index` or `0, 1,`

2... for `step_index`), `DictVectorizer` would treat them as numerical features and pass them through directly, just like `voltage` and `current`.

3. **What the Function Achieved:**

   - In this scenario, the function, using `DictVectorizer` with numerical `step_index` and `cycle_index`, primarily served to:
     - Convert the list of dictionaries into a structured NumPy array (or sparse matrix).
     - Ensure a consistent order of features (`current`, `cycle_index`, `step_index`, `voltage`).
   - It did **not** create the expanded set of binary features that one-hot encoding would typically produce for categorical data.

Implications:

- If your intention was for `step_index` and `cycle_index` to be treated as distinct categories where each specific index value represents a unique state (and there's no ordinal relationship implied by their numerical values), then this current output is not achieving that.
- Machine learning models using this output would treat `step_index` and `cycle_index` as continuous numerical variables, which might not be appropriate if they are truly categorical identifiers. For example, a model might incorrectly assume that `step_index = 2` is "greater than" or "twice" `step_index = 1` in a meaningful way beyond just their identifier.

How to Achieve One-Hot Encoding if `step_index` and `cycle_index` are Numerical but Categorical in Nature:

If `step_index` and `cycle_index` are indeed categorical (even if represented by numbers like 0, 1, 2, 3...), you'd typically need to:

1. **Convert them to strings before passing to `DictVectorizer`:**

   ```
   # Inside your loop where you create the list of dictionaries:
   dict_list.append({
       'step_index': str(row_step_index), # Convert to string
       'cycle_index': str(row_cycle_index), # Convert to string
       'voltage': row_voltage,
       'current': row_current
   })
   ```

   By doing this, `DictVectorizer` will recognize them as strings and apply one-hot encoding.

2. **Or, use other encoders like `OneHotEncoder` from Scikit-learn:** This would typically involve selecting these columns from your DataFrame and applying `OneHotEncoder` directly, then concatenating the results with your other numerical features. This approach often requires `ColumnTransformer` for more streamlined preprocessing.

Thanks for providing that output – it clarifies exactly what's happening with the `DictVectorizer` in this specific case! Your interpretation is correct.

Question 2: What Derived Features Can We Create from

Battery Data?

Step 1: Polynomial Features for Non-Linear Relationships

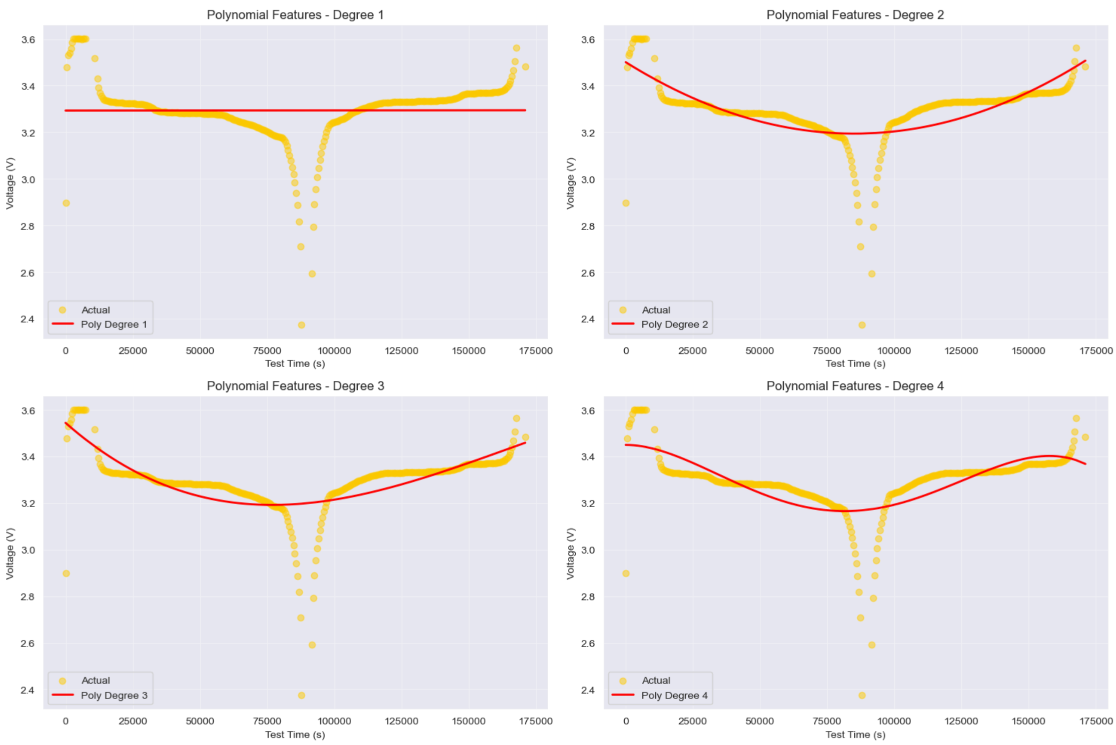
Battery behavior is inherently non-linear. Polynomial features help capture these complex relationships.

In [63]:

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt

def create_polynomial_battery_features(dataset, degree=3):
    """
    Create polynomial features from voltage and current measurements
    to capture non-linear battery behavior patterns.
    """
    # Extract primary electrical features
    electrical_features = dataset[['Voltage(V)', 'Current(A)', 'Internal_Re

    # Create polynomial features
    poly_transformer = PolynomialFeatures(degree=degree, include_bias=False
    poly_features = poly_transformer.fit_transform(electrical_features)

    # Get feature names
    feature_names = poly_transformer.get_feature_names_out(['Voltage', 'Cur

    return poly_features, feature_names, poly_transformer

def demonstrate_polynomial_regression():
    """
    Demonstrate polynomial feature engineering for battery voltage predicti
    """
    # Use 25°C data as reference (room temperature)
    reference_data = low_curr_ocv_25.copy()

    # Create time-voltage relationship
    X = reference_data[['Test_Time(s)']].values
    y = reference_data['Voltage(V)'].values

    # Sample every 100th point for cleaner visualization
    sample_indices = np.arange(0, len(X), 100)
    X_sample = X[sample_indices]
    y_sample = y[sample_indices]

    # Create polynomial features of different degrees
    degrees = [1, 2, 3, 4]
    plt.figure(figsize=(15, 10))

    for i, degree in enumerate(degrees):
```

```
        plt.subplot(2, 2, i+1)

        # Create polynomial features
        poly = PolynomialFeatures(degree=degree, include_bias=False)
        X_poly = poly.fit_transform(X_sample)

        # Fit linear regression on polynomial features
        model = LinearRegression().fit(X_poly, y_sample)
        y_pred = model.predict(X_poly)

        # Plot results
        plt.scatter(X_sample.flatten(), y_sample, alpha=0.5, color=temp_col
        plt.plot(X_sample.flatten(), y_pred, color='red', linewidth=2, labe
        plt.xlabel('Test Time (s)')
        plt.ylabel('Voltage (V)')
        plt.title(f'Polynomial Features - Degree {degree}')
        plt.legend()
        plt.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()

    return poly, X_poly, model

# Example implementation
poly_features, poly_names, poly_transformer = create_polynomial_battery_fea
print(f"Polynomial features created: {len(poly_names)} features from 3 orig
print(f"Sample feature names: {poly_names[:10]}")

demonstrate_polynomial_regression()
```

```
Polynomial features created: 19 features from 3 original features
Sample feature names: ['Voltage' 'Current' 'Resistance' 'Voltage^2' 'Voltage
Current'
 'Voltage Resistance' 'Current^2' 'Current Resistance' 'Resistance^2'
 'Voltage^3']
```




```
Out[63]: (PolynomialFeatures(degree=4, include_bias=False),
          array([[3.00388078e+00, 9.02329973e+00, 2.71049166e+01, 8.14199380e+01],
                 [5.03183268e+02, 2.53193401e+05, 1.27402683e+08, 6.41068984e+10],
                 [1.00329102e+03, 1.00659287e+06, 1.00990558e+09, 1.01322920e+12],
```

```
        ...,
        [1.67165759e+05, 2.79443909e+10, 4.67134531e+15, 7.80888983e+20],
        [1.67665904e+05, 2.81118553e+10, 4.71339963e+15, 7.90276408e+20],
        [1.71135613e+05, 2.92873980e+10, 5.01211680e+15, 8.57751681e+20]]),
 LinearRegression())
```

This is a great description of how polynomial feature engineering is being used to model battery behavior, and why it's effective!

Here's a breakdown of your explanation in a structured Markdown format, emphasizing the key concepts:

Polynomial Feature Engineering for A123 Battery Voltage Modeling

This visualization and process demonstrate how **polynomial feature engineering** can transform a simple linear regression model into a more powerful tool capable of capturing the non-linear dynamics of battery behavior over time.

Visual Analysis of Model Complexity (Voltage vs. Time Plots)

The plots illustrate how increasing the degree of polynomial features allows the regression model to better fit the actual battery voltage measurements:

- **Yellow Dots:** Represent the actual measured battery voltage over time.
- **Red Lines:** Show the polynomial regression model's fit to this data at different degrees of complexity.

1. **Degree 1 (Top Left - Linear Fit):**
   - A simple straight line.
   - **Fails significantly** to capture the battery's non-linear behavior, especially evident in its inability to model the characteristic voltage dip during discharge (seen around 75,000-100,000 seconds in your data).
   - Represents a high-bias model for this problem.
2. **Degree 2 (Top Right - Quadratic Fit):**
   - A U-shaped curve.
   - Provides a **better fit** than the linear model, starting to follow the general curvature of the battery's voltage profile (e.g., during a discharge and subsequent charge).
3. **Degree 3 (Bottom Left - Cubic Fit):**
   - Allows for more complex curves (e.g., with an inflection point).
   - Captures **more nuanced behavior** and a tighter fit to the voltage dynamics.
4. **Degree 4 (Bottom Right - Fourth-Degree Fit):**
   - Offers even more flexibility.
   - Provides a **slightly better fit** again, adapting more closely to the subtle

changes in the voltage curve.

The Feature Engineering Process Explained

The core idea is to create new features from the existing ones by taking them to various powers and creating interaction terms.

1. **Original Features:**
   - The process started with 3 fundamental electrical features:
     - `Voltage(V)`
     - `Current(A)`
     - `Internal_Resistance(Ohm)` (presumably)
2. **Polynomial Feature Expansion (to 19 features with degree=4):**
   - Using a tool like Scikit-learn's `PolynomialFeatures(degree=4, include_bias=False)`, the original set of features is expanded. This new set includes:
     - **Original features:** `V`, `I`, `R`
     - **Squared terms:** `V²`, `I²`, `R²`
     - **Interaction terms:** `V×I`, `V×R`, `I×R`
     - **Higher-order terms:** `V³`, `I³`, `R³`, `V²I`, `VI²`, etc., up to the specified degree (in this case, 4).
   - `include_bias=False` means a constant bias term (intercept) is not added as a feature here (it's typically handled by the `LinearRegression` model itself).
3. **Model Application:**
   - A `LinearRegression()` model is then applied to this **new, expanded set of 19 polynomial features.**

Why This Approach is Powerful for Battery Modeling:

- **Capturing Non-Linearity:** While the `LinearRegression` algorithm itself is fitting a linear equation, it's doing so in a much higher-dimensional space created by the polynomial features. This allows the overall model to represent complex, non-linear relationships between the *original* inputs and the target (voltage).
- **Modeling Complex Patterns:** Battery charge-discharge cycles, voltage plateaus, and the influence of factors like temperature (if included) and internal resistance on voltage are inherently non-linear. Polynomial features provide the necessary flexibility to model these curves and interactions.
- **Improved Fit:** As seen in the progression from Degree 1 to Degree 4, higher-degree polynomials generally provide a better fit to the observed battery voltage behavior over time, capturing the dips and curves more accurately.

In essence, polynomial feature engineering allows a fundamentally linear algorithm to achieve non-linear modeling capabilities. This is a common and effective technique when the underlying relationships in the data are known or

suspected to be non-linear, as is often the case with complex physical systems like batteries.

This Python code defines and uses a function called `create_polynomial_battery_features` to perform **polynomial feature engineering**. Let's break down what it does and how these generated features can be used:

What the Code Does: `create_polynomial_battery_features`

The function `create_polynomial_battery_features(dataset, degree=3)` is designed to take an input battery dataset and generate new features based on polynomial combinations of existing electrical measurements. The goal, as stated in the docstring, is to "capture non-linear battery behavior patterns."

Here's a step-by-step explanation:

1. **Selects Original Electrical Features:**
    - It first extracts three specific columns from your input `dataset` : `'Voltage(V)'` , `'Current(A)'` , and `'Internal_Resistance(Ohm)'` . These are your base features.
2. **Creates PolynomialFeatures Transformer:**
    - It initializes a `PolynomialFeatures` object from the `sklearn.preprocessing` module. This is the core tool for generating the new features.
    - `degree=degree` : This parameter (defaulting to 3 in the function definition, and also implied by your output of 19 features from 3 originals) determines the maximum degree of the polynomial terms to be created.
    - `include_bias=False` : This means it will not add a "bias" column (a column of all ones, which acts as an intercept in linear models). Linear regression models usually handle the intercept themselves.
    - `interaction_only=False` : This is crucial. It means the transformer will create:
        - **Individual feature powers:** e.g., `Voltage^2` , `Current^3` , `Resistance^2` .
        - **Interaction terms:** e.g., `Voltage * Current` , `Voltage * Resistance` , `Current * Resistance * Voltage` . If this were `True` , it would only create interaction terms (like `Voltage*Current`) and not individual powers (like `Voltage^2`).
3. **Fits and Transforms the Data:**
    - `poly_transformer.fit_transform(electrical_features)` :
        - The `fit` part analyzes the input `electrical_features` to determine all the polynomial combinations up to the specified `degree` .

- The `transform` part then applies this learned transformation to generate the new set of polynomial features.

4. **Gets Feature Names:**
   - `poly_transformer.get_feature_names_out(['Voltage', 'Current', 'Resistance'])` : This method generates descriptive names for each of the newly created polynomial features, making them easier to understand (e.g., 'Voltage^2', 'Voltage Current', 'Current Resistance^2').
5. **Returns Values:**
   - `poly_features` : A NumPy array containing the newly generated polynomial features. If your original data had `N` rows and 3 features, and `degree=3` results in 19 features, this array will have a shape of `(N, 19)` .
   - `feature_names` : A list of the names for these 19 features.
   - `poly_transformer` : The fitted `PolynomialFeatures` object. This is useful because you can use it to apply the *exact same transformation* to new, unseen data (like a test set) later, ensuring consistency.

Example Output Explanation:

```
Polynomial features created: 19 features from 3 original
features
Sample feature names: ['Voltage' 'Current' 'Resistance'
'Voltage^2' 'Voltage Current'
 'Voltage Resistance' 'Current^2' 'Current Resistance'
'Resistance^2'
 'Voltage^3']
```

- This output confirms that from your 3 original features (`Voltage` , `Current` , `Resistance`), setting `degree=3` in `PolynomialFeatures` (with `include_bias=False` and `interaction_only=False`) results in 19 features.
- The sample names clearly show the structure:
  - Original features (degree 1): `Voltage` , `Current` , `Resistance`
  - Squared terms (degree 2): `Voltage^2` , `Current^2` , `Resistance^2`
  - Interaction terms (degree 2): `Voltage Current` , `Voltage Resistance` , `Current Resistance`
  - Cubic terms (degree 3): `Voltage^3` (and others like `Current^3` , `Resistance^3` , `Voltage^2 Current` , `Voltage Current Resistance` , etc., which would be further down the list).

How We Can Use These Polynomial Features:

The primary use of these generated polynomial features is to **allow linear machine learning models to capture non-linear relationships in your data.**

1. **Modeling Non-Linearity with Linear Models:**

- Many real-world phenomena, especially in physical systems like batteries, are inherently non-linear. For example, the relationship between voltage, current, temperature, and a battery's state of health or capacity degradation is often not a simple straight line.
- Standard linear models (like `LinearRegression` , `LogisticRegression` , `Linear SVM`) can only learn linear relationships.
- By creating polynomial features (e.g., x_1^2, x_1x_2, x_2^3), you are essentially transforming your feature space. A linear model trained on these new features can learn an equation like: $y = w_0 + w_1x_1 + w_2x_2 + w_3x_1^2 + w_4x_1x_2 + w_5x_2^2 + \ldots$ While this equation is *linear in terms of the new polynomial features* (x_1^2, x_1x_2, etc.), it represents a *non-linear (polynomial) relationship in terms of the original features* (x_1, x_2).

2. **Capturing Interactions Between Features:**

   - Terms like `Voltage Current` (an interaction term) allow the model to learn how the effect of voltage on the target variable might change depending on the current level (and vice-versa). This is often crucial because features in physical systems rarely act in complete isolation. For instance, the impact of current on battery health might be exacerbated at very high or very low voltages.

3. **Improving Model Accuracy:**

   - If the true underlying relationship between your features and the target variable is indeed non-linear (and can be approximated by a polynomial), then providing these polynomial features to a linear model can dramatically improve its predictive accuracy and fit to the data. This was likely the case in your previous example where higher-degree polynomials better fit the battery voltage curves.

4. **Input for Any Machine Learning Model:**

   - While often discussed in the context of enhancing linear models, these engineered polynomial features can be used as input to *any* machine learning algorithm (e.g., Decision Trees, Random Forests, Gradient Boosting Machines, Neural Networks).
   - Sometimes, explicitly providing these non-linear terms can help even complex models (which can learn non-linearities implicitly) to converge faster or find slightly better solutions by giving them a "head start."

Practical Considerations:

- **Increased Feature Dimensionality:** Polynomial feature generation can significantly increase the number of features. This can lead to higher computational costs and an increased risk of **overfitting**, especially if the degree is set too high or the dataset is small (this is part of the "curse of dimensionality").
- **Choosing the Degree:** The choice of `degree` is important. Too low, and you might not capture the complexity. Too high, and you risk overfitting and creating an overly complex model. This often requires experimentation, cross-

validation, or domain knowledge.
- **Multicollinearity:** The generated polynomial features are often highly correlated with each other and with the original features. This might be an issue for interpreting coefficients in linear models but can often be handled by regularization techniques (like Ridge or Lasso regression).
- **Feature Scaling:** It's generally a good practice to scale your features (e.g., using `StandardScaler`) *before* generating polynomial features, or to scale the *resulting* polynomial features, especially for models that are sensitive to the scale of input features.

In summary, `create_polynomial_battery_features` is a preprocessing step that enriches your dataset by adding non-linear transformations of your original electrical features. These new features then enable simpler models (like linear regression) to learn more complex, non-linear patterns in your battery data, potentially leading to more accurate predictions.

You "fill" these new polynomial features with samples by **calculating their values directly from the original feature values of each existing sample (row) in your dataset.**

The `PolynomialFeatures` transformer from Scikit-learn automates this process. For every single row (sample) in your input data, it takes the original feature values for that row and computes the corresponding polynomial terms.

Let's make this concrete with an example. Suppose you have one sample (one row of data) with the following original feature values:

- `Voltage(V)` = 3.7
- `Current(A)` = 1.5
- `Resistance(Ohm)` = 0.1

When you use `PolynomialFeatures(degree=3, include_bias=False, interaction_only=False)` (which creates 19 features from these 3 original ones), here's how some of the new feature values for **that specific sample** would be calculated and "filled":

1. **Original Features (Degree 1):**
    - `Voltage` : 3.7 (taken directly)
    - `Current` : 1.5 (taken directly)
    - `Resistance` : 0.1 (taken directly)
2. **Squared Terms (Degree 2):**
    - `Voltage^2` : $3.7^2 = 13.69$
    - `Current^2` : $1.5^2 = 2.25$
    - `Resistance^2` : $0.1^2 = 0.01$
3. **Interaction Terms (Degree 2):**
    - `Voltage Current` : $3.7 \times 1.5 = 5.55$
    - `Voltage Resistance` : $3.7 \times 0.1 = 0.37$
    - `Current Resistance` : $1.5 \times 0.1 = 0.15$

4. **Cubic Terms (Degree 3):**
   - `Voltage^3` : $3.7^3 = 50.653$
   - `Current^3` : $1.5^3 = 3.375$
   - `Resistance^3` : $0.1^3 = 0.001$
   - `Voltage^2 Current` : $3.7^2 \times 1.5 = 13.69 \times 1.5 = 20.535$
   - `Voltage Current Resistance` : $3.7 \times 1.5 \times 0.1 = 0.555$
   - ...and so on for all 19 combinations.

How `PolynomialFeatures` does it:

When you call `poly_features = poly_transformer.fit_transform(electrical_features)` , the `fit_transform` method does the following:

1. **`fit`** : It looks at the input data (`electrical_features`) to determine the structure of the polynomial combinations based on the specified `degree` (e.g., it figures out it needs to create terms like $x_1, x_2, x_3, x_1^2, x_1x_2$, etc.).
2. **`transform`** : It then iterates through **each row (each sample)** of your `electrical_features` data. For each row, it takes the original values in that row and calculates all the polynomial terms it identified during the `fit` step.

The Result:

The output (`poly_features`) will be a new array (or matrix) that has:

- The **same number of rows (samples)** as your original `electrical_features` dataset.
- An **expanded number of columns (features)** – in your case, 19.

Each row in this new `poly_features` array corresponds to an original sample, but now it has the values for all the generated polynomial features, calculated from that sample's original `Voltage` , `Current` , and `Resistance` values.

So, you are not adding new *samples*; you are adding new *calculated characteristics (features)* for each of your existing samples.

Step 2: Power and Energy Derived Features

Power and energy calculations reveal battery efficiency and performance characteristics.

In [64]:
```
def create_power_energy_features(dataset):
    """
    Create derived features related to power and energy calculations
    """
    derived_features = dataset.copy()

    # Instantaneous power
```

```
    derived_features['Power(W)'] = derived_features['Voltage(V)'] * derived

    # Energy efficiency ratios
    derived_features['Charge_Efficiency'] = (
        derived_features['Charge_Energy(Wh)'] / (derived_features['Charge_C
    )

    derived_features['Discharge_Efficiency'] = (
        derived_features['Discharge_Energy(Wh)'] / (derived_features['Disch
    )

    # Voltage rate features
    derived_features['Voltage_Rate_Abs'] = np.abs(derived_features['dV/dt(V

    # Resistance-based features
    derived_features['Conductance(S)'] = 1.0 / (derived_features['Internal_

    # Temperature differential (if available)
    if 'Temperature(C)_2' in derived_features.columns:
        derived_features['Temp_Differential'] = (
            derived_features['Temperature(C)_1'] - derived_features['Temper
        )

    return derived_features

# Apply to all temperature datasets
enhanced_datasets = {}
for temp_label, dataset in temp_datasets.items():
    enhanced_datasets[temp_label] = create_power_energy_features(dataset)

print("Enhanced features example for 25°C:")
print(enhanced_datasets['25°C'][['Power(W)', 'Charge_Efficiency', 'Conducta
```

```
Enhanced features example for 25°C:
   Power(W)  Charge_Efficiency  Conductance(S)
0  0.000000           0.000000     100000000.0
1  3.681732           3.110463     100000000.0
2  3.705678           3.121935     100000000.0
3  3.726359           3.132025     100000000.0
4  3.744986           3.141233     100000000.0
```

In [65]:

```
print(enhanced_datasets['50°C'][['Power(W)', 'Charge_Efficiency', 'Conducta
```

```
   Power(W)  Charge_Efficiency  Conductance(S)
0  0.000000           0.000000     100000000.0
1  4.211142           0.212099     100000000.0
2  3.287810           3.601050     100000000.0
3  3.236932           3.601052     100000000.0
4  3.198651           3.601047     100000000.0
```

Step 3: Temperature-Dependent Interaction Features

Temperature affects all battery properties. Creating interaction features captures these dependencies.

In [66]:

```
def create_temperature_interaction_features(dataset):
    """
```

```
    Create features that capture temperature effects on battery behavior
    """
    temp_features = dataset.copy()

    # Temperature normalization (Kelvin scale)
    temp_kelvin = temp_features['Temperature (C)_1'] + 273.15

    # Temperature-dependent features
    temp_features['Voltage_per_Temp'] = temp_features['Voltage(V)'] / temp_
    temp_features['Resistance_Temp_Factor'] = (
        temp_features['Internal_Resistance(Ohm)'] * temp_kelvin
    )

    # Arrhenius-like features for battery kinetics
    temp_features['Temp_Reciprocal'] = 1 / temp_kelvin

    # Power temperature dependency
    temp_features['Power_Temp_Normalized'] = temp_features['Power(W)'] / te

    # Capacity temperature coefficient
    temp_features['Capacity_Temp_Coeff'] = (
        temp_features['Discharge_Capacity(Ah)'] * temp_features['Temp_Recip
    )

    return temp_features

# Apply temperature interaction features
temp_enhanced_datasets = {}
for temp_label, dataset in enhanced_datasets.items():
    temp_enhanced_datasets[temp_label] = create_temperature_interaction_fea

print("Temperature interaction features sample:")
sample_cols = ['Voltage_per_Temp', 'Resistance_Temp_Factor', 'Temp_Reciproc
print(temp_enhanced_datasets['25°C'][sample_cols].describe())
```

```
Temperature interaction features sample:
       Voltage_per_Temp  Resistance_Temp_Factor  Temp_Reciprocal
count      32307.000000                 32307.0     32307.000000
mean           0.010992                     0.0         0.003336
std            0.000440                     0.0         0.000002
min            0.006674                     0.0         0.003320
25%            0.010894                     0.0         0.003336
50%            0.011062                     0.0         0.003336
75%            0.011129                     0.0         0.003337
max            0.012025                     0.0         0.003343
```

Question 3: How Do We Extract Time-Series Features

from Battery Data?

Step 1: Temporal Pattern Extraction

Time-series feature engineering reveals hidden patterns in battery behavior over different time scales.

In [67]:

```
def create_temporal_features(dataset):
    """
    Extract temporal patterns and cyclical features from battery test data
    """
    temporal_data = dataset.copy()

    # Convert datetime to proper format
    temporal_data['DateTime'] = pd.to_datetime(temporal_data['Date_Time'])

    # Extract time components
    temporal_data['Hour'] = temporal_data['DateTime'].dt.hour
    temporal_data['Day'] = temporal_data['DateTime'].dt.day
    temporal_data['DayOfWeek'] = temporal_data['DateTime'].dt.dayofweek

    # Cyclical encoding for time features
    temporal_data['Hour_Sin'] = np.sin(2 * np.pi * temporal_data['Hour'] /
    temporal_data['Hour_Cos'] = np.cos(2 * np.pi * temporal_data['Hour'] /

    # Test duration features
    temporal_data['Test_Duration_Hours'] = temporal_data['Test_Time(s)'] /
    temporal_data['Step_Duration_Minutes'] = temporal_data['Step_Time(s)']

    # Rolling window features (5-point windows)
    temporal_data['Voltage_MA5'] = temporal_data['Voltage(V)'].rolling(wind
    temporal_data['Current_MA5'] = temporal_data['Current(A)'].rolling(wind

    # Lagged features
    temporal_data['Voltage_Lag1'] = temporal_data['Voltage(V)'].shift(1)
    temporal_data['Voltage_Lag5'] = temporal_data['Voltage(V)'].shift(5)

    # Rate of change features
    temporal_data['Voltage_ROC'] = temporal_data['Voltage(V)'].pct_change()
    temporal_data['Current_ROC'] = temporal_data['Current(A)'].pct_change()

    return temporal_data

# Example implementation
temporal_enhanced = create_temporal_features(low_curr_ocv_25)
print("Temporal features sample:")
print(temporal_enhanced[['Hour_Sin', 'Hour_Cos', 'Voltage_MA5', 'Voltage_RO
```

```
Temporal features sample:
   Hour_Sin  Hour_Cos  Voltage_MA5  Voltage_ROC
0  0.707107 -0.707107          NaN          NaN
1  0.707107 -0.707107          NaN     0.077745
2  0.707107 -0.707107     3.100391     0.006504
3  0.707107 -0.707107     3.158992     0.005581
4  0.707107 -0.707107     3.175242     0.004966
5  0.707107 -0.707107     3.189954     0.004457
6  0.707107 -0.707107     3.203373     0.004244
7  0.707107 -0.707107     3.215807     0.003938
8  0.707107 -0.707107     3.227503     0.003444
9  0.707107 -0.707107     3.238460     0.003337
```

This Python code defines a function `create_temporal_features` that performs **temporal feature engineering** on a battery dataset. This process involves creating new features from existing time-based data to help machine learning models better understand and capture time-dependent patterns, trends, and cyclical behaviors.

Here's a breakdown of what the code does and why these features are created:

What the Code Does: `create_temporal_features`

The function takes a battery dataset (assumed to be a pandas DataFrame) as input and adds several new columns (features) based on time-related information.

1. **Creates a Copy:**
   - `temporal_data = dataset.copy()` : It starts by creating a copy of the input dataset to avoid modifying the original DataFrame.
2. **Converts to Datetime Objects:**
   - `temporal_data['DateTime'] = pd.to_datetime(temporal_data['Date_Time'])` : It assumes there's a column named `'Date_Time'` (likely containing strings representing date and time) and converts it into proper pandas datetime objects. This is essential for easily extracting time components.
3. **Extracts Basic Time Components:**
   - `temporal_data['Hour'] = temporal_data['DateTime'].dt.hour` : Extracts the hour of the day (0-23).
   - `temporal_data['Day'] = temporal_data['DateTime'].dt.day` : Extracts the day of the month (1-31).
   - `temporal_data['DayOfWeek'] = temporal_data['DateTime'].dt.dayofweek` : Extracts the day of the week (Monday=0, Sunday=6).
4. **Cyclical Encoding for Time Features:**
   - This is a crucial step for features that have a cyclical nature (like hours in a day, days in a week, months in a year). Directly using numerical values (e.g., hour 23 being "greater" than hour 0) can be misleading for models.
   - `temporal_data['Hour_Sin'] = np.sin(2 * np.pi * temporal_data['Hour'] / 24)`
   - `temporal_data['Hour_Cos'] = np.cos(2 * np.pi * temporal_data['Hour'] / 24)`
     - These lines transform the 'Hour' feature into two new features using sine and cosine transformations. This maps the cyclical hour data onto a circle, so hour 23 is close to hour 0 in this new 2D space, which better represents its cyclical nature for ML models.
5. **Test Duration Features:**
   - These features represent elapsed time in more interpretable units.
   - `temporal_data['Test_Duration_Hours'] = temporal_data['Test_Time(s)'] / 3600` : Converts a column `'Test_Time(s)'` (assumed to be total elapsed test time in seconds) into hours.
   - `temporal_data['Step_Duration_Minutes'] =`

`temporal_data['Step_Time(s)'] / 60` : Converts a column `'Step_Time(s)'` (assumed to be elapsed time within the current test step in seconds) into minutes.

6. **Rolling Window Features (Moving Averages):**
   - These features smooth out short-term fluctuations and highlight longer-term trends.
   - `temporal_data['Voltage_MA5'] = temporal_data['Voltage(V)'].rolling(window=5, center=True).mean()` : Calculates a 5-point centered moving average for `Voltage(V)` . For each point, it takes the average of the current point, the 2 previous points, and the 2 next points.
   - `temporal_data['Current_MA5'] = temporal_data['Current(A)'].rolling(window=5, center=True).mean()` : Similarly, calculates a 5-point centered moving average for `Current(A)` .
   - `center=True` means the window is centered around the current data point. This introduces NaNs at the beginning and end of the series where a full window isn't available.
7. **Lagged Features:**
   - These features provide the model with information about past values of a variable, which can be very predictive for time series data.
   - `temporal_data['Voltage_Lag1'] = temporal_data['Voltage(V)'].shift(1)` : Creates a new feature that contains the value of `Voltage(V)` from the *previous* time step (lag 1).
   - `temporal_data['Voltage_Lag5'] = temporal_data['Voltage(V)'].shift(5)` : Creates a new feature with the value of `Voltage(V)` from 5 time steps ago (lag 5).
   - The first few rows for these lagged features will be NaN because there are no preceding values.
8. **Rate of Change (ROC) Features:**
   - These features capture how quickly a variable is changing, which can indicate trends or sudden shifts.
   - `temporal_data['Voltage_ROC'] = temporal_data['Voltage(V)'].pct_change()` : Calculates the percentage change in `Voltage(V)` compared to the previous time step.
   - `temporal_data['Current_ROC'] = temporal_data['Current(A)'].pct_change()` : Calculates the percentage change in `Current(A)` compared to the previous time step.
   - The first row for ROC features will be NaN because there's no previous value to compare against.
9. **Returns the Enhanced DataFrame:**
   - `return temporal_data` : The function returns the DataFrame with all these newly created temporal features.

Why We Are Doing This (Purpose of Temporal Feature Engineering):

Creating these temporal features is essential for several reasons, especially when building predictive models for time-series data like battery cycling tests:

1. **Capturing Time Dependence:**
    - Raw timestamps are often not directly useful for many ML models. Extracting components like hour, day, or day of the week allows the model to learn patterns related to these specific time periods (e.g., if tests run overnight behave differently, or if there are weekly maintenance cycles affecting data).
2. **Handling Cyclical Patterns:**
    - Features like 'Hour' are cyclical (23:00 is followed by 00:00, and they are close in a cycle). Simple numerical representation doesn't capture this. Sine and cosine transformations (e.g., `Hour_Sin` , `Hour_Cos`) convert these into a 2D representation where the cyclical nature is preserved, helping models understand that 11 PM is close to 1 AM.
3. **Highlighting Trends and Smoothing Noise:**
    - Rolling window features (like moving averages `Voltage_MA5`) help to smooth out noisy sensor readings and reveal underlying trends in voltage or current over a short period. This can make the signal clearer for the model.
4. **Providing Historical Context (Memory):**
    - Lagged features (`Voltage_Lag1` , `Voltage_Lag5`) give the model a "memory" of recent past states. For instance, the current voltage might be highly dependent on the voltage a few moments ago. This is crucial for predicting future states or understanding current dynamics.
5. **Detecting Dynamics and Momentum:**
    - Rate of Change (ROC) features tell the model how quickly things are changing. A rapid drop in voltage (high negative ROC) might indicate a problem or the start of a discharge step, which is very different from a stable voltage (ROC near zero).
6. **Improving Model Performance:**
    - By providing these more informative, engineered features, you often enable machine learning models to learn more complex patterns, make more accurate predictions, and generalize better to new data. The raw data might not explicitly contain these relationships in a way the model can easily learn.
7. **Domain-Specific Insights:**
    - For battery data, `Test_Duration_Hours` or `Step_Duration_Minutes` can be important. The behavior of a battery might change significantly

depending on how long a test or a specific step (like a constant current charge) has been running.

Explanation of Sample Output:

```
Temporal features sample:
   Hour_Sin  Hour_Cos  Voltage_MA5  Voltage_ROC
0  0.707107 -0.707107          NaN          NaN
1  0.707107 -0.707107          NaN     0.077745
2  0.707107 -0.707107     3.100391     0.006504
3  0.707107 -0.707107     3.158992     0.005581
4  0.707107 -0.707107     3.175242     0.004966
5  0.707107 -0.707107     3.189954     0.004457
6  0.707107 -0.707107     3.203373     0.004244
7  0.707107 -0.707107     3.215807     0.003938
8  0.707107 -0.707107     3.227503     0.003444
9  0.707107 -0.707107     3.238460     0.003337
```

- **`Hour_Sin` and `Hour_Cos` :**
  - The values `0.707107` for `Hour_Sin` and `-0.707107` for `Hour_Cos` are constant across these first 10 rows. This means all these data points occurred at the **same hour of the day**.
    - `sin(2 * pi * H / 24) = 0.707107` (approx. $1/\sqrt{2}$)
    - `cos(2 * pi * H / 24) = -0.707107` (approx. $-1/\sqrt{2}$)
    - This corresponds to an angle of 135° or $3\pi/4$ radians. Since a full day is 2π radians (or 24 hours), this hour (H) would be $(3\pi/4)/(2\pi) * 24 = (3/8) * 24 = 9$. So, these measurements were likely taken around **9 AM**.
- **`Voltage_MA5` (5-point centered Moving Average of Voltage):**
  - **Rows 0 and 1 are `NaN` (Not a Number):** This is expected. For a centered moving average with a window of 5, you need 2 data points before the current point and 2 data points after.
    - For row 0, there are no preceding points.
    - For row 1, there is only 1 preceding point.
  - **Row 2 has a value (`3.100391`):** This is the average of `Voltage(V)` from rows 0, 1, 2, 3, and 4.
  - Subsequent rows have valid moving average values as enough data points are available for the window.
- **`Voltage_ROC` (Rate of Change of Voltage):**
  - **Row 0 is `NaN` :** This is expected. The percentage change is calculated relative to the previous row's value. For the first row, there is no previous row.
  - **Row 1 has a value (`0.077745`):** This means the `Voltage(V)` in row 1 increased by approximately 7.77% compared to the `Voltage(V)` in row 0.
  - The subsequent small positive values indicate that the voltage is generally increasing slowly over these first few time steps.

This kind of feature engineering is a powerful way to extract more meaningful signals from raw time-series data, making it easier for machine learning models to learn and predict.

Step 2: Cycle-Based Feature Engineering

Battery charge/discharge cycles reveal performance patterns and degradation trends.

In [68]:

```
def create_cycle_features(dataset):
    """
    Create features based on battery charge/discharge cycles
    """
    cycle_data = dataset.copy()

    # Cycle progress features
    cycle_data['Cycle_Progress'] = (
        cycle_data.groupby('Cycle_Index')['Step_Time(s)'].rank(pct=True)
    )

    # Cumulative features per cycle
    cycle_data['Cumulative_Charge'] = (
        cycle_data.groupby('Cycle_Index')['Charge_Capacity(Ah)'].cumsum()
    )

    cycle_data['Cumulative_Discharge'] = (
        cycle_data.groupby('Cycle_Index')['Discharge_Capacity(Ah)'].cumsum(
    )

    # Cycle statistics
    cycle_stats = cycle_data.groupby('Cycle_Index').agg({
        'Voltage(V)': ['mean', 'std', 'min', 'max'],
        'Current(A)': ['mean', 'std'],
        'Internal_Resistance(Ohm)': ['mean', 'max']
    }).reset_index()

    # Flatten column names
    cycle_stats.columns = ['_'.join(col).strip() if col[1] else col[0]
                           for col in cycle_stats.columns.values]

    return cycle_data, cycle_stats

# Apply cycle feature engineering
cycle_enhanced, cycle_stats = create_cycle_features(low_curr_ocv_25)
print("Cycle statistics sample:")
cycle_stats.head()
```

```
Cycle statistics sample:
```

Out[68]:

	Cycle_Index	Voltage(V)_mean	Voltage(V)_std	Voltage(V)_min	Voltage(V)_max
0	1	3.294757	0.132443	1.999724	3.600773

Step 3: Advanced Statistical Features

Step 3: Advanced Statistical Features

Rolling window statistics capture local patterns and trends in battery behavior.

In [69]:
```
def create_statistical_features(dataset, window_size=10):
    """
    Create statistical features using rolling windows
    """
    stat_features = dataset.copy()

    # Rolling statistics for key variables
    key_vars = ['Voltage(V)', 'Current(A)', 'Internal_Resistance(Ohm)']

    for var in key_vars:
        # Rolling mean and std
        stat_features[f'{var}_RollingMean_{window_size}'] = (
            stat_features[var].rolling(window=window_size, center=True).mea
        )

        stat_features[f'{var}_RollingStd_{window_size}'] = (
            stat_features[var].rolling(window=window_size, center=True).std
        )

        # Rolling min and max
        stat_features[f'{var}_RollingMin_{window_size}'] = (
            stat_features[var].rolling(window=window_size, center=True).min
        )

        stat_features[f'{var}_RollingMax_{window_size}'] = (
            stat_features[var].rolling(window=window_size, center=True).max
        )

        # Rolling range
        stat_features[f'{var}_RollingRange_{window_size}'] = (
            stat_features[f'{var}_RollingMax_{window_size}'] -
            stat_features[f'{var}_RollingMin_{window_size}']
        )

        # Percentile features
        stat_features[f'{var}_Rolling25th_{window_size}'] = (
            stat_features[var].rolling(window=window_size, center=True).qua
        )

        stat_features[f'{var}_Rolling75th_{window_size}'] = (
            stat_features[var].rolling(window=window_size, center=True).qua
        )

    return stat_features

# Apply statistical feature engineering
statistical_enhanced = create_statistical_features(low_curr_ocv_25, window_

# Show statistical features
stat_cols = [col for col in statistical_enhanced.columns if 'Rolling' in co
print(f"Created {len(stat_cols)} statistical features")
print("Sample statistical features:")
print(statistical_enhanced[stat_cols[:5]].head(10))
```

```
Created 21 statistical features
Sample statistical features:
   Voltage(V)_RollingMean_10  Voltage(V)_RollingStd_10  \
0                        NaN                       NaN
```

```
1                     NaN                   NaN
2                     NaN                   NaN
3                     NaN                   NaN
4                     NaN                   NaN
5                3.158099              0.098674
6                3.193247              0.041957
7                3.206851              0.038608
8                3.219316              0.035878
9                3.230919              0.033599

   Voltage(V)_RollingMin_10  Voltage(V)_RollingMax_10  \
0                       NaN                       NaN
1                       NaN                       NaN
2                       NaN                       NaN
3                       NaN                       NaN
4                       NaN                       NaN
5                  2.897814                  3.238829
6                  3.123105                  3.249293
7                  3.143419                  3.259142
8                  3.160962                  3.268068
9                  3.176658                  3.276993

   Voltage(V)_RollingRange_10
0                         NaN
1                         NaN
2                         NaN
3                         NaN
4                         NaN
5                    0.341015
6                    0.126188
7                    0.115724
8                    0.107106
9                    0.100335
```

Question 4: How Do We Handle Missing Data in Battery

Datasets?

Step 1: Systematic Missing Data Analysis

Understanding missing data patterns is crucial for selecting appropriate imputation strate- gies.

In [70]:

```
from sklearn.impute import SimpleImputer, KNNImputer
import seaborn as sns

def analyze_missing_data(datasets_dict):
    """
    Analyze missing data patterns across all temperature datasets
    """
    missing_analysis = {}

    for temp_label, dataset in datasets_dict.items():
```

```
        # Calculate missing data percentage
        missing_pct = (dataset.isnull().sum() / len(dataset)) * 100
        missing_analysis[temp_label] = missing_pct[missing_pct > 0]

    # Create missing data summary
    missing_df = pd.DataFrame(missing_analysis).fillna(0)

    return missing_df

def implement_imputation_strategies(dataset):
    """
    Implement multiple imputation strategies for battery data
    """
    # Identify numerical columns for imputation
    numerical_cols = dataset.select_dtypes(include=[np.number]).columns

    # Strategy 1: Mean imputation for basic features
    mean_imputer = SimpleImputer(strategy='mean')
    dataset_mean_imputed = dataset.copy()
    dataset_mean_imputed[numerical_cols] = mean_imputer.fit_transform(datas

    # Strategy 2: Median imputation for robust estimation
    median_imputer = SimpleImputer(strategy='median')
    dataset_median_imputed = dataset.copy()
    dataset_median_imputed[numerical_cols] = median_imputer.fit_transform(d

    # Strategy 3: KNN imputation for pattern-based filling
    knn_imputer = KNNImputer(n_neighbors=5)
    dataset_knn_imputed = dataset.copy()
    dataset_knn_imputed[numerical_cols] = knn_imputer.fit_transform(dataset

    return {
        'mean': dataset_mean_imputed,
        'median': dataset_median_imputed,
        'knn': dataset_knn_imputed
    }, {
        'mean_imputer': mean_imputer,
        'median_imputer': median_imputer,
        'knn_imputer': knn_imputer
    }

# Analyze missing data across all datasets
missing_analysis = analyze_missing_data(temp_datasets)
print("Missing data analysis:")
print(missing_analysis)

# Example imputation on 25°C data
imputed_datasets, imputers = implement_imputation_strategies(low_curr_ocv_2
print("\nImputation completed for multiple strategies")
```

```
Missing data analysis:
Empty DataFrame
Columns: [-10°C, 0°C, 10°C, 20°C, 25°C, 30°C, 40°C, 50°C]
Index: []

Imputation completed for multiple strategies
```

Step 2: Time-Series Specific Imputation

Battery data has temporal structure that should be preserved during imputation.

In [71]:
```
def time_series_imputation(dataset):
    """
    Specialized imputation for time-series battery data
    """
    ts_data = dataset.copy().sort_values('Test_Time(s)')

    # Forward fill for sequential measurements
    ts_data_ffill = ts_data.fillna(method='ffill')

    # Backward fill for end sequences
    ts_data_bfill = ts_data_ffill.fillna(method='bfill')

    # Interpolation for smooth transitions
    numerical_cols = ts_data.select_dtypes(include=[np.number]).columns
    ts_data_interp = ts_data.copy()

    for col in numerical_cols:
        ts_data_interp[col] = ts_data_interp[col].interpolate(method='linea

    return ts_data_bfill, ts_data_interp

# Apply time series imputation
ts_filled, ts_interpolated = time_series_imputation(low_curr_ocv_25)
print("Time series imputation completed")
```

```
Time series imputation completed
C:\Users\jaiku\AppData\Local\Temp\ipykernel_12044\2728982935.py:8: FutureWarn
ing: DataFrame.fillna with 'method' is deprecated and will raise in a future
version. Use obj.ffill() or obj.bfill() instead.
  ts_data_ffill = ts_data.fillna(method='ffill')
C:\Users\jaiku\AppData\Local\Temp\ipykernel_12044\2728982935.py:11: FutureWar
ning: DataFrame.fillna with 'method' is deprecated and will raise in a future
version. Use obj.ffill() or obj.bfill() instead.
  ts_data_bfill = ts_data_ffill.fillna(method='bfill')
```

Question 5: How Do We Build Comprehensive Feature

Pipelines?

Step 1: Comprehensive Feature Engineering Pipeline

A well-designed pipeline ensures consistent preprocessing across all datasets and enables easy deployment.

In [72]:
```
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, MinMaxScaler

class BatteryFeatureEngineer:
    """
    Comprehensive feature engineering pipeline for battery data analysis
```

```
    """

    def __init__(self, polynomial_degree=2, include_temporal=True):
        self.polynomial_degree = polynomial_degree
        self.include_temporal = include_temporal
        self.pipelines = {}

    def create_electrical_pipeline(self):
        """Create pipeline for electrical features"""
        electrical_pipeline = Pipeline([
            ('imputer', SimpleImputer(strategy='mean')),
            ('poly_features', PolynomialFeatures(degree=self.polynomial_deg
            ('scaler', StandardScaler())
        ])
        return electrical_pipeline

    def create_temporal_pipeline(self):
        """Create pipeline for temporal features"""
        temporal_pipeline = Pipeline([
            ('imputer', SimpleImputer(strategy='median')),
            ('scaler', MinMaxScaler())
        ])
        return temporal_pipeline

    def create_categorical_pipeline(self):
        """Create pipeline for categorical features"""
        categorical_pipeline = Pipeline([
            ('onehot', OneHotEncoder(sparse_output=False, handle_unknown='i
        ])
        return categorical_pipeline

    def build_complete_pipeline(self, dataset, target_column=None):
     """Build complete feature engineering pipeline"""

    # Define feature groups - but exclude target column if specified
     electrical_features = ['Voltage(V)', 'Current(A)', 'Internal_Resistanc
     capacity_features = ['Charge_Capacity(Ah)', 'Discharge_Capacity(Ah)']
     energy_features = ['Charge_Energy(Wh)', 'Discharge_Energy(Wh)']
     temporal_features = ['Test_Time(s)', 'Step_Time(s)']

    # Remove target from feature lists if it's being used as target
     if target_column:
         electrical_features = [col for col in electrical_features if col !
         capacity_features = [col for col in capacity_features if col != ta
         energy_features = [col for col in energy_features if col != target
         temporal_features = [col for col in temporal_features if col != ta

    # Create transformers only for non-empty feature groups
     transformers = []
     if electrical_features:
        transformers.append(('electrical', self.create_electrical_pipeline(
     if capacity_features:
        transformers.append(('capacity', self.create_temporal_pipeline(), c
     if energy_features:
        transformers.append(('energy', self.create_temporal_pipeline(), ene
     if temporal_features:
        transformers.append(('temporal', self.create_temporal_pipeline(), t

     preprocessor = ColumnTransformer(transformers, remainder='drop')
     return preprocessor

    def fit_transform_dataset(self, dataset, target_column=None):
        """Apply complete pipeline to dataset"""
        # Build pipeline
        pipeline = self.build_complete_pipeline(dataset, target_column)
```

```
        # Separate features and target
        if target_column:
            X = dataset.drop(columns=[target_column])
            y = dataset[target_column]
        else:
            X = dataset
            y = None

        # Transform features
        X_transformed = pipeline.fit_transform(X)

        return X_transformed, y, pipeline

# Example usage of complete pipeline
battery_engineer = BatteryFeatureEngineer(polynomial_degree=3, include_temp

# Apply to 25°C dataset
X_transformed, y, complete_pipeline = battery_engineer.fit_transform_datase
    low_curr_ocv_25,
    target_column='Voltage(V)'
)

print("Pipeline transformation completed:")
print(f"Original features: {low_curr_ocv_25.shape[1]}")
print(f"Transformed features: {X_transformed.shape[1]}")
print(f"Samples: {X_transformed.shape[0]}")
```

```
Pipeline transformation completed:
Original features: 18
Transformed features: 15
Samples: 32307
```

Step 2: Temperature-Specific Pipeline Implementation

Different temperature conditions may require specialized preprocessing approaches.

In [73]:

```
def create_temperature_specific_pipeline():
    """
    Create pipeline that handles multiple temperature datasets
    """

    class MultiTemperaturePipeline:
        def __init__(self):
            self.temp_pipelines = {}
            self.combined_pipeline = None

        def fit_individual_temperatures(self, temp_datasets):
            """Fit separate pipelines for each temperature"""
            for temp_label, dataset in temp_datasets.items():
                engineer = BatteryFeatureEngineer(polynomial_degree=2)
                X_trans, y, pipeline = engineer.fit_transform_dataset(
                    dataset, target_column='Voltage(V)'
                )

                self.temp_pipelines[temp_label] = {
                    'pipeline': pipeline,
                    'X_transformed': X_trans,
```

```
                    'y': y,
                    'original_data': dataset
                }

            return self.temp_pipelines

        def create_combined_features(self, temp_datasets):
            """Create combined dataset with temperature as feature"""
            combined_data = []

            for temp_label, dataset in temp_datasets.items():
                # Add temperature as numerical feature
                temp_data = dataset.copy()
                temp_data['Temperature_Numeric'] = float(temp_label.replace
                temp_data['Temperature_Category'] = temp_label
                combined_data.append(temp_data)

            # Combine all datasets
            full_combined = pd.concat(combined_data, ignore_index=True)

            # Create combined pipeline
            combined_engineer = BatteryFeatureEngineer(polynomial_degree=2)
            X_combined, y_combined, combined_pipeline = combined_engineer.f
                full_combined, target_column='Voltage(V)'
            )

            self.combined_pipeline = combined_pipeline
            return X_combined, y_combined, full_combined

    return MultiTemperaturePipeline()

# Implement multi-temperature pipeline
multi_temp_pipeline = create_temperature_specific_pipeline()

# Fit individual temperature pipelines
individual_results = multi_temp_pipeline.fit_individual_temperatures(temp_d

# Create combined pipeline
X_combined, y_combined, combined_dataset = multi_temp_pipeline.create_combi

print("Multi-temperature pipeline results:")
for temp_label, results in individual_results.items():
    print(f"{temp_label}: {results['X_transformed'].shape} features")

print(f"\nCombined dataset: {X_combined.shape} features, {len(combined_data
```

```
Multi-temperature pipeline results:
-10°C: (29785, 11) features
0°C: (30249, 11) features
10°C: (31898, 11) features
20°C: (31018, 11) features
25°C: (32307, 11) features
30°C: (31150, 11) features
40°C: (31258, 11) features
50°C: (31475, 11) features

Combined dataset: (249140, 11) features, 249140 samples
```

Question 6: What Advanced Domain-Specific Features

Can We Create?

Step 1: Battery Health and State Features

State estimation is fundamental to battery management systems.

In [77]:

```
def create_battery_domain_features(dataset):
    """
    Create domain-specific features for battery analysis
    """
    battery_features = dataset.copy()

    # State of Charge (SOC) approximation
    max_capacity = battery_features['Charge_Capacity(Ah)'].max()
    battery_features['SOC_Approx'] = battery_features['Charge_Capacity(Ah)'

    # Depth of Discharge (DOD)
    battery_features['DOD_Approx'] = 1 - battery_features['SOC_Approx']

    # Coulombic Efficiency
    battery_features['Coulombic_Efficiency'] = (
        battery_features['Discharge_Capacity(Ah)'] /
        (battery_features['Charge_Capacity(Ah)'] + 1e-8)
    )

    # Energy Efficiency
    battery_features['Energy_Efficiency'] = (
        battery_features['Discharge_Energy(Wh)'] /
        (battery_features['Charge_Energy(Wh)'] + 1e-8)
    )

    # Voltage stability metrics
    battery_features['Voltage_Stability'] = (
        1 / (np.abs(battery_features['dV/dt(V/s)']) + 1e-8)
    )

    # Power density approximation
    battery_features['Power_Density'] = (
        battery_features['Voltage(V)'] * battery_features['Current(A)']
    )

    # Impedance-based features
    battery_features['Impedance_Ratio'] = (
        battery_features['AC_Impedance(Ohm)'] /
        (battery_features['Internal_Resistance(Ohm)'] + 1e-8)
    )

    # Phase angle analysis
    battery_features['Phase_Angle_Rad'] = np.deg2rad(battery_features['ACI_
    battery_features['Impedance_Real'] = (
        battery_features['AC_Impedance(Ohm)'] * np.cos(battery_features['Ph
    )
    battery_features['Impedance_Imaginary'] = (
        battery_features['AC_Impedance(Ohm)'] * np.sin(battery_features['Ph
    )

    return battery_features
```

```
# Apply domain-specific feature engineering
domain_enhanced = {}
for temp_label, dataset in temp_datasets.items():
    domain_enhanced[temp_label] = create_battery_domain_features(dataset)

print("Domain-specific features for 25°C:")
domain_cols = ['SOC_Approx', 'Coulombic_Efficiency', 'Energy_Efficiency', '
print(domain_enhanced['25°C'][domain_cols].describe())
```

```
Domain-specific features for 25°C:
          SOC_Approx  Coulombic_Efficiency  Energy_Efficiency  Power_Density  
\
count  32307.000000          32307.000000       32307.000000   32307.000000   
mean       0.611490              0.575133           0.539893       0.083459   
std        0.169815              0.277488           0.258144       0.576897   
min        0.000000              0.000000           0.000000      -0.174837   
25%        0.496835              0.428981           0.402915      -0.163959   
50%        0.496835              0.611010           0.577465       0.134944   
75%        0.735744              0.782466           0.733764       0.166594   
max        1.000000              1.016041           0.935824       4.244253   

       Impedance_Ratio  Phase_Angle_Rad  
count          32307.0          32307.0  
mean               0.0              0.0  
std                0.0              0.0  
min                0.0              0.0  
25%                0.0              0.0  
50%                0.0              0.0  
75%                0.0              0.0  
max                0.0              0.0  
```

Step 2: Thermal Management Features

Temperature effects are critical for battery safety and performance.

In [88]:

```
def create_thermal_features(dataset):
    """
    Create thermal management and heat generation features
    """
    thermal_features = dataset.copy()

    # Temperature differential features
    # thermal_features['Temp_Gradient'] = thermal_features['Temperature (C)_
    thermal_features['Temp_Stability'] = thermal_features.groupby('Cycle_In

    # Thermal resistance features
    thermal_features['Thermal_Resistance'] = (
        (thermal_features['Temperature (C)_1'] - 25) / thermal_features['Po
    ).replace([np.inf, -np.inf], np.nan)

    # Heat generation rate approximation
    thermal_features['Heat_Generation_Rate'] = (
        thermal_features['Internal_Resistance(Ohm)'] *
        thermal_features['Current(A)']**2
    )

    # Temperature coefficient features
    thermal_features['Voltage_Temp_Coeff'] = (
        thermal_features['Voltage(V)'] / (thermal_features['Temperature (C)
```

```
    )

    # Thermal time constant approximation
    thermal_features['Temp_Change_Rate'] = thermal_features.groupby('Cycle_

    return thermal_features

# Apply thermal feature engineering
thermal_enhanced = {}
for temp_label, dataset in domain_enhanced.items():
    thermal_enhanced[temp_label] = create_thermal_features(dataset)

print("Thermal features sample:")
thermal_cols = [ 'Thermal_Resistance', 'Heat_Generation_Rate']
print(thermal_enhanced['25°C'][thermal_cols].describe())
```

```
Thermal features sample:
       Thermal_Resistance  Heat_Generation_Rate
count        32126.000000               32307.0
mean             0.546663                   0.0
std             10.130932                   0.0
min            -17.071881                   0.0
25%             -9.498549                   0.0
50%              5.904092                   0.0
75%              9.574703                   0.0
max             49.715710                   0.0
```

Step 3: Advanced Electrochemical Features

Electrochemical impedance analysis provides deep insights into battery internal processes.

In [89]:

```
def create_electrochemical_features(dataset):
    """
    Create advanced electrochemical analysis features
    """
    electrochem_features = dataset.copy()

    # Impedance magnitude and phase relationships
    electrochem_features['Impedance_Magnitude'] = np.sqrt(
        electrochem_features['Impedance_Real']**2 +
        electrochem_features['Impedance_Imaginary']**2
    )

    # Equivalent circuit parameters approximation
    electrochem_features['Series_Resistance'] = electrochem_features['Imped
    electrochem_features['Charge_Transfer_Resistance'] = (
        electrochem_features['Impedance_Real'] - electrochem_features['Seri
    )

    # Capacitive behavior indicators
    electrochem_features['Capacitive_Component'] = (
        -1 / (2 * np.pi * 1 * electrochem_features['Impedance_Imaginary'] +
        # Assuming 1 Hz frequency
    )

    # Diffusion impedance approximation (Warburg)
    electrochem_features['Warburg_Impedance'] = (
```

```
        electrochem_features['Impedance_Real'] * np.sqrt(2) *
        np.sign(electrochem_features['Impedance_Imaginary'])
    )

    # Ion transport features
    electrochem_features['Ionic_Conductivity'] = (
        1 / (electrochem_features['Internal_Resistance(Ohm)'] + 1e-8)
    )

    # State-dependent resistance
    electrochem_features['SOC_Resistance_Product'] = (
        electrochem_features['SOC_Approx'] * electrochem_features['Internal
    )

    # Concentration polarization indicators
    electrochem_features['Concentration_Overpotential'] = (
        electrochem_features['Voltage(V)'] - 3.3  # Assuming 3.3V nominal
    ) * electrochem_features['Current(A)']

    return electrochem_features

# Apply electrochemical feature engineering
electrochem_enhanced = {}
for temp_label, dataset in thermal_enhanced.items():
    electrochem_enhanced[temp_label] = create_electrochemical_features(data

print("Electrochemical features sample:")
electrochem_cols = ['Impedance_Magnitude', 'Charge_Transfer_Resistance', 'I
print(electrochem_enhanced['25°C'][electrochem_cols].describe())
```

```
Electrochemical features sample:
       Impedance_Magnitude  Charge_Transfer_Resistance  Ionic_Conductivity
count              32307.0                     32307.0             32307.0
mean                   0.0                         0.0         100000000.0
std                    0.0                         0.0                 0.0
min                    0.0                         0.0         100000000.0
25%                    0.0                         0.0         100000000.0
50%                    0.0                         0.0         100000000.0
75%                    0.0                         0.0         100000000.0
max                    0.0                         0.0         100000000.0
```

Question 7: How Do We Validate and Select the Best

Features?

Step 1: Feature Importance Analysis

Systematic feature importance analysis helps identify the most valuable features for pre- diction tasks.

In [90]:

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.model_selection import cross_val_score
from sklearn.metrics import mean_squared_error, r2_score
```

```
def analyze_feature_engineering_impact():
    """
    Analyze the impact of different feature engineering techniques
    """
    # Compare original vs engineered features for prediction

    # Use 25°C data as reference
    original_data = low_curr_ocv_25.copy()
    enhanced_data = electrochem_enhanced['25°C'].copy()

    # Original features
    original_features = ['Current(A)', 'Internal_Resistance(Ohm)', 'Charge_
    X_original = original_data[original_features].fillna(original_data[orig

    # Enhanced features
    enhanced_features = original_features + ['SOC_Approx', 'Energy_Efficien
                                              'Thermal_Resistance', 'Impedance_
    X_enhanced = enhanced_data[enhanced_features].fillna(enhanced_data[enha

    # Target variable
    y = original_data['Voltage(V)']

    # Model evaluation
    model = RandomForestRegressor(n_estimators=100, random_state=42)

    # Original features performance
    cv_scores_original = cross_val_score(model, X_original, y, cv=5, scorin

    # Enhanced features performance
    cv_scores_enhanced = cross_val_score(model, X_enhanced, y, cv=5, scorin

    # Create comparison
    comparison_results = {
        'Original Features': {
            'Mean R²': cv_scores_original.mean(),
            'Std R²': cv_scores_original.std(),
            'Features Count': X_original.shape[1]
        },
        'Enhanced Features': {
            'Mean R²': cv_scores_enhanced.mean(),
            'Std R²': cv_scores_enhanced.std(),
            'Features Count': X_enhanced.shape[1]
        }
    }

    return comparison_results

def perform_feature_importance_analysis(dataset, target_col='Voltage(V)'):
    """
    Comprehensive feature importance analysis
    """
    # Prepare data
    feature_cols = [col for col in dataset.columns if col != target_col and
                    dataset[col].dtype in ['float64', 'int64']]

    X = dataset[feature_cols].fillna(dataset[feature_cols].mean())
    y = dataset[target_col]

    # Random Forest feature importance
    rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
    rf_model.fit(X, y)

    # Create feature importance dataframe
    feature_importance_df = pd.DataFrame({
```

```
        'feature': feature_cols,
        'importance': rf_model.feature_importances_,
        'feature_type': [classify_feature_type(col) for col in feature_cols
    }).sort_values('importance', ascending=False)

    # Statistical feature selection
    selector = SelectKBest(score_func=f_regression, k=10)
    X_selected = selector.fit_transform(X, y)
    selected_features = [feature_cols[i] for i in selector.get_support(indi

    return feature_importance_df, selected_features, rf_model

def classify_feature_type(feature_name):
    """Classify feature into categories for analysis"""
    if any(term in feature_name.lower() for term in ['temp', 'thermal']):
        return 'Thermal'
    elif any(term in feature_name.lower() for term in ['voltage', 'current'
        return 'Electrical'
    elif any(term in feature_name.lower() for term in ['capacity', 'energy'
        return 'Energy'
    elif any(term in feature_name.lower() for term in ['soc', 'dod', 'effic
        return 'State'
    elif any(term in feature_name.lower() for term in ['impedance', 'phase'
        return 'Electrochemical'
    elif any(term in feature_name.lower() for term in ['time', 'cycle', 'st
        return 'Temporal'
    else:
        return 'Other'

# Analyze feature engineering impact
impact_analysis = analyze_feature_engineering_impact()
print("Feature Engineering Impact Analysis:")
print("="*50)
for feature_type, metrics in impact_analysis.items():
    print(f"\n{feature_type}:")
    for metric, value in metrics.items():
        print(f"  {metric}: {value:.4f}" if isinstance(value, float) else f

# Comprehensive feature importance analysis
feature_importance, selected_features, rf_model = perform_feature_importanc
    electrochem_enhanced['25°C']
)

print("\nTop 10 Most Important Features:")
print(feature_importance.head(10))
```

```
Feature Engineering Impact Analysis:
==================================================

Original Features:
  Mean R²: -7.0121
  Std R²: 9.2506
  Features Count: 3

Enhanced Features:
  Mean R²: -2.3015
  Std R²: 3.7531
  Features Count: 8

Top 10 Most Important Features:
                        feature  importance feature_type
29              Voltage_Temp_Coeff    0.999253      Thermal
38  Concentration_Overpotential    0.000272        Other
```

```
2                  Step_Time(s)    0.000250     Temporal
21                Power_Density    0.000045       Energy
20            Voltage_Stability    0.000025   Electrical
27           Thermal_Resistance    0.000023      Thermal
10                  dV/dt(V/s)    0.000023        Other
19            Energy_Efficiency    0.000017       Energy
1                  Test_Time(s)    0.000016     Temporal
18          Coulombic_Efficiency    0.000015        State
```

This Python code is designed to evaluate the impact of feature engineering and to determine the importance of various features for predicting a target variable, likely related to battery performance (specifically 'Voltage(V)' in the example).

Let's break down what the code does and what the output means:

What the Code Does:

1. **`analyze_feature_engineering_impact()` function:**
   - **Purpose:** To compare the predictive performance of a `RandomForestRegressor` model when using a basic set of "Original Features" versus an "Enhanced Features" set.
   - **Process:**
     - It takes a predefined `low_curr_ocv_25` dataset (presumably battery data at 25°C).
     - Defines `original_features` (`Current(A)` , `Internal_Resistance(Ohm)` , `Charge_Capacity(Ah)`).
     - Defines `enhanced_features` by adding more engineered features like `SOC_Approx` , `Energy_Efficiency` , etc., to the original ones.
     - Handles missing values in both feature sets by filling them with the mean of each column.
     - The target variable `y` is set to `'Voltage(V)'` from the original data.
     - It uses `cross_val_score` with 5 folds (`cv=5`) to evaluate the `RandomForestRegressor` model using R^2 scoring for both `X_original` and `X_enhanced` .
     - Returns a dictionary comparing the mean R^2, standard deviation of R^2, and feature count for both scenarios.
2. **`perform_feature_importance_analysis(dataset, target_col='Voltage(V)')` function:**
   - **Purpose:** To identify which features are most influential in predicting the `target_col` (defaulting to `'Voltage(V)'`).
   - **Process:**
     - Prepares data by selecting numerical columns (excluding the target) and filling missing values with the mean.
     - **Random Forest Feature Importance:** Trains a `RandomForestRegressor` on the data and extracts feature importances (based on how much each feature contributes to reducing impurity or error in the trees). It then creates a DataFrame showing each feature, its importance score, and its classified type (using

`classify_feature_type`).
    - **Statistical Feature Selection (`SelectKBest`):** Uses `SelectKBest` with the `f_regression` scoring function to select the top 10 features based on their univariate linear regression F-statistic with the target variable.
    - Returns the feature importance DataFrame, the list of top 10 selected features from `SelectKBest` , and the trained Random Forest model.
3. **`classify_feature_type(feature_name)` function:**

    - **Purpose:** A helper function to categorize feature names based on keywords (e.g., 'temp' -> 'Thermal', 'voltage' -> 'Electrical'). This helps in organizing and interpreting the feature importance results.

Explanation of the Output:

Feature Engineering Impact Analysis:

```
Feature Engineering Impact Analysis:
==================================================

Original Features:
  Mean R²: -7.0121
  Std R²: 9.2506
  Features Count: 3

Enhanced Features:
  Mean R²: -2.3015
  Std R²: 3.7531
  Features Count: 8
```

- **Mean R^2 (R-squared):** This is the most critical part of this output.
    - **An R^2 score indicates the proportion of the variance in the dependent variable (Voltage) that is predictable from the independent variables (features).**
    - R^2 ranges from $-\infty$ to 1.
        - An R^2 of 1 means perfect prediction.
        - An R^2 of 0 means the model performs no better than predicting the mean of the target variable.
        - **Negative R^2 scores (like -7.0121 and -2.3015 here) are a very bad sign.** They mean that the model is performing *worse* than a simple horizontal line (i.e., worse than just predicting the average voltage for all data points). This often indicates that the model is a very poor fit for the data, or there might be issues with the data itself, the features chosen, or how the cross-validation is being applied to this specific dataset.
- **Std R^2 (Standard Deviation of R^2):** This shows the variability of the R^2 scores across the 5 cross-validation folds. High standard deviations (like 9.2506 and 3.7531) mean the model's performance was very inconsistent across different subsets of the data, which is also undesirable.

subsets of the data, which is also undesirable.

- **Features Count:** Simply the number of features used in each scenario.
- **Interpretation of Impact:**
  - Both the "Original Features" and "Enhanced Features" sets are leading to extremely poor model performance.
  - While "Enhanced Features" have a *less negative* Mean R^2 and a lower (though still high) Std R^2, the performance is still far from acceptable. The "enhancement" has made the model slightly less terrible, but it hasn't made it good.

Top 10 Most Important Features:

```
Top 10 Most Important Features:
                        feature  importance feature_type
29          Voltage_Temp_Coeff    0.999253      Thermal
38  Concentration_Overpotential    0.000272        Other
2                  Step_Time(s)    0.000250     Temporal
21                Power_Density    0.000045       Energy
20            Voltage_Stability    0.000025   Electrical
27           Thermal_Resistance    0.000023      Thermal
10                   dV/dt(V/s)    0.000023        Other
19            Energy_Efficiency    0.000017       Energy
1                   Test_Time(s)    0.000016     Temporal
18          Coulombic_Efficiency    0.000015        State
```

- `Voltage_Temp_Coeff` **Dominance:** This feature has an importance score of **0.999253**, meaning it accounts for almost 99.93% of the predictive power according to the Random Forest model. This is an **extremely dominant feature**.
- **Other Features:** All other features have incredibly small importance scores in comparison.
- **Interpretation:**
  - **Potential Data Leakage or Target Proxy:** The overwhelming importance of `Voltage_Temp_Coeff` is a major red flag. It strongly suggests that this feature might be a "leaky" feature or a proxy for the target variable (`Voltage(V)`).
    - A "leaky" feature is one that contains information about the target that would not be available at the time of prediction in a real-world scenario. For example, if `Voltage_Temp_Coeff` was calculated using the target voltage itself, or values very closely derived from it.
    - If this feature is too directly related to the target, the model might be trivially predicting the voltage using this single feature, making all other features seem unimportant and potentially explaining the strange R^2 scores if this feature isn't perfectly predictive or is handled poorly in some CV folds.
  - The model seems to be relying almost entirely on `Voltage_Temp_Coeff` .

Key Concerns and Next Steps:

1. **Negative R^2 Scores:** This is the primary concern. You need to investigate why the models are performing so poorly.

    - **Data Quality:** Check for outliers, errors, or inconsistencies in your `low_curr_ocv_25` and `electrochem_enhanced['25°C']` datasets.
    - **Feature Relevance:** Are the chosen features (even the enhanced ones) actually predictive of voltage in a way that Random Forest can capture? The very low importance of most features (barring the dominant one) suggests maybe not.
    - **Target Variable Distribution:** Examine the distribution of `'Voltage(V)'`.
    - **Model Choice:** While Random Forest is generally robust, perhaps the data has characteristics that make it unsuitable without further specific preprocessing.
    - **Cross-Validation Setup:** Ensure `cross_val_score` is being used appropriately for your data structure (e.g., if it's time-series data, standard KFold might not be ideal without shuffling, but `shuffle=True` is usually the default for `RandomForestRegressor` in `cross_val_score` if `cv` is an integer).
2. **Dominant Feature (`Voltage_Temp_Coeff`):**

    - **Investigate its origin:** How is this feature calculated or derived? Is it possible it includes information directly from the target `Voltage(V)` ?
    - **Temporarily remove it:** Try running the `perform_feature_importance_analysis` and `analyze_feature_engineering_impact` after removing `Voltage_Temp_Coeff` to see how other features perform and if R^2 scores improve (or at least become non-negative and interpretable).
3. **High Standard Deviation in R^2:** This indicates that model performance is highly variable depending on how the data is split for cross-validation, which is also a sign of instability or that the model is not learning generalizable patterns.

This output suggests that there are fundamental issues either with the data, the features being used (especially the dominant one), or how the modeling is being approached, leading to unreliable and poor predictive performance. The immediate next step should be a deep dive into the `Voltage_Temp_Coeff` feature and a re-evaluation of the modeling strategy after addressing any issues with it.

Step 2: Cross-Temperature Validation

Validating features across different temperature conditions ensures robustness and gener- alizability.

In [91]:
```
from sklearn.model_selection import GroupKFold

def cross_temperature_validation():
    """
    Validate features across different temperature conditions
    """
```

```
    # Combine all temperature datasets
    combined_data = []
    temperature_groups = []

    for i, (temp_label, dataset) in enumerate(electrochem_enhanced.items())
        temp_data = dataset.copy()
        temp_data['Temperature_Group'] = i
        combined_data.append(temp_data)
        temperature_groups.extend([i] * len(dataset))

    # Create combined dataset
    full_combined = pd.concat(combined_data, ignore_index=True)

    # Prepare features and target
    feature_cols = [col for col in full_combined.columns if
                   col not in ['Voltage(V)', 'Temperature_Group'] and
                   full_combined[col].dtype in ['float64', 'int64']]

    X = full_combined[feature_cols].fillna(full_combined[feature_cols].mean
    y = full_combined['Voltage(V)']
    groups = np.array(temperature_groups)

    # Group K-Fold cross-validation (temperature as groups)
    cv = GroupKFold(n_splits=5)

    # Test different feature sets
    feature_sets = {
        'Basic Electrical': ['Current(A)', 'Internal_Resistance(Ohm)', 'Cha
        'With Domain Features': ['Current(A)', 'Internal_Resistance(Ohm)',
                                'SOC_Approx', 'Energy_Efficiency', 'Power_D
        'Full Enhanced': feature_cols[:20]  # Top 20 features to avoid over
    }

    results = {}
    model = RandomForestRegressor(n_estimators=100, random_state=42)

    for set_name, features in feature_sets.items():
        # Select available features
        available_features = [f for f in features if f in X.columns]
        X_subset = X[available_features]

        # Cross-validation with temperature grouping
        cv_scores = cross_val_score(model, X_subset, y, cv=cv, groups=group

        results[set_name] = {
            'mean_r2': cv_scores.mean(),
            'std_r2': cv_scores.std(),
            'feature_count': len(available_features)
        }

    return results

def analyze_temperature_specific_importance():
    """
    Analyze how feature importance varies across temperatures
    """
    temperature_importance = {}

    for temp_label, dataset in electrochem_enhanced.items():
        feature_importance, _, _ = perform_feature_importance_analysis(data
        temperature_importance[temp_label] = feature_importance.head(10)

    return temperature_importance

# Cross-temperature validation
```

```
cross_temp_results = cross_temperature_validation()
print("Cross-Temperature Validation Results:")
print("="*50)
for set_name, metrics in cross_temp_results.items():
    print(f"\n{set_name}:")
    print(f"  Mean R²: {metrics['mean_r2']:.4f} ± {metrics['std_r2']:.4f}")
    print(f"  Features: {metrics['feature_count']}")

# Temperature-specific importance analysis
temp_importance = analyze_temperature_specific_importance()
print("\nFeature Importance Across Temperatures:")
for temp_label in ['-10°C', '25°C', '50°C']:  # Sample temperatures
    print(f"\n{temp_label} - Top 5 Features:")
    if temp_label in temp_importance:
        print(temp_importance[temp_label][['feature', 'importance']].head()
```

```
Cross-Temperature Validation Results:
==================================================

Basic Electrical:
  Mean R²: 0.0470 ± 0.4046
  Features: 3

With Domain Features:
  Mean R²: 0.6293 ± 0.4605
  Features: 6

Full Enhanced:
  Mean R²: 0.9297 ± 0.0303
  Features: 20
C:\Users\jaiku\anaconda3\Lib\site-packages\sklearn\feature_selection\_univari
ate_selection.py:379: RuntimeWarning: invalid value encountered in sqrt
  X_norms = np.sqrt(row_norms(X.T, squared=True) - n_samples * X_means**2)
C:\Users\jaiku\anaconda3\Lib\site-packages\sklearn\feature_selection\_univari
ate_selection.py:379: RuntimeWarning: invalid value encountered in sqrt
  X_norms = np.sqrt(row_norms(X.T, squared=True) - n_samples * X_means**2)
C:\Users\jaiku\anaconda3\Lib\site-packages\sklearn\feature_selection\_univari
ate_selection.py:379: RuntimeWarning: invalid value encountered in sqrt
  X_norms = np.sqrt(row_norms(X.T, squared=True) - n_samples * X_means**2)
C:\Users\jaiku\anaconda3\Lib\site-packages\sklearn\feature_selection\_univari
ate_selection.py:379: RuntimeWarning: invalid value encountered in sqrt
  X_norms = np.sqrt(row_norms(X.T, squared=True) - n_samples * X_means**2)
C:\Users\jaiku\anaconda3\Lib\site-packages\sklearn\feature_selection\_univari
ate_selection.py:379: RuntimeWarning: invalid value encountered in sqrt
  X_norms = np.sqrt(row_norms(X.T, squared=True) - n_samples * X_means**2)
Feature Importance Across Temperatures:

-10°C - Top 5 Features:
                    feature  importance
          Voltage_Temp_Coeff    0.979527
           Energy_Efficiency    0.012446
         Coulombic_Efficiency    0.007541
Concentration_Overpotential    0.000238
                 Step_Time(s)    0.000071

25°C - Top 5 Features:
                    feature  importance
          Voltage_Temp_Coeff    0.999253
Concentration_Overpotential    0.000272
                 Step_Time(s)    0.000250
                Power_Density    0.000045
            Voltage_Stability    0.000025
```

```
50°C - Top 5 Features:
                      feature  importance
          Voltage_Temp_Coeff    0.998347
Concentration_Overpotential    0.000587
                 Step_Time(s)    0.000362
        Coulombic_Efficiency    0.000262
          Thermal_Resistance    0.000204
```

This Python code implements two important analyses for your A123 battery data:

1. **Cross-Temperature Validation:** It assesses how well models trained on some temperature conditions can generalize to predict voltage at *different, unseen* temperature conditions. This is a robust way to test model generalization.
2. **Temperature-Specific Feature Importance:** It identifies which features are most crucial for predicting voltage *within each specific temperature condition*.

Let's break down the code and the output in detail:

What the Code Does:

1. `cross_temperature_validation()` function:

- **Purpose:** To evaluate how different sets of features perform when the model is forced to predict on temperature conditions it hasn't been trained on for that specific fold. This is a more realistic and challenging validation than standard random cross-validation if your goal is to have a model that works across various temperatures.
- **Process:**
  - **Combines Data:** It first takes all your temperature-specific datasets (from `electrochem_enhanced`) and concatenates them into one large DataFrame (`full_combined`).
  - **Creates Groups:** A crucial step is creating a `Temperature_Group` identifier for each original dataset. This `groups` array will be used by `GroupKFold` .
  - **Prepares Features and Target:**
    - `feature_cols` : Selects all numerical columns as potential features, excluding the target `'Voltage(V)'` and the helper `'Temperature_Group'` column.
    - Missing values in features `X` are filled with the mean of each column.
    - `y` is the target `'Voltage(V)'` .
  - **`GroupKFold` Cross-Validation:**
    - `cv = GroupKFold(n_splits=5)` : This is the core of this validation. Instead of random splits, `GroupKFold` ensures that **all data points belonging to the same group (i.e., the same original temperature condition) are kept together in either the training set or the test set for each fold.**
    - This means in each of the 5 folds, the model will be trained on data from some temperatures and tested on data from completely different temperatures.

- **Tests Different Feature Sets:** It iterates through predefined `feature_sets`:
  - `'Basic Electrical'`: Only 3 fundamental electrical features.
  - `'With Domain Features'`: The basic set plus 3 more domain-specific engineered features.
  - `'Full Enhanced'`: The top 20 features from the `feature_cols` list (this selection of top 20 is a heuristic to try and use a rich set while potentially avoiding too many less important ones).
- **Model Evaluation:** For each feature set:
  - It selects the `available_features` from `X`.
  - It uses `cross_val_score` with the `RandomForestRegressor`, the prepared `X_subset`, `y`, the `GroupKFold` object (`cv`), and the `groups` array.
  - The scoring metric is R^2 (`scoring='r2'`).
- **Returns Results:** A dictionary containing the mean R^2, standard deviation of R^2, and feature count for each tested feature set.

2. `analyze_temperature_specific_importance()` function:

- **Purpose:** To understand which features are most important for predicting voltage when the model is trained and tested *only on data from a single, specific temperature condition*. This can reveal if different features become more or less relevant at different temperatures.
- **Process:**
  - It iterates through each `temp_label` and `dataset` in your `electrochem_enhanced` dictionary.
  - For each temperature-specific dataset, it calls the `perform_feature_importance_analysis` function (which you provided in a previous context – it trains a Random Forest and extracts feature importances).
  - It stores the top 10 most important features (and their importance scores) for each temperature.
  - Returns a dictionary where keys are temperature labels and values are DataFrames of the top features for that temperature.

3. `perform_feature_importance_analysis()` and `classify_feature_type()` (Assumed from previous context):

- These functions were part of your previous code snippet.
- `perform_feature_importance_analysis` trains a Random Forest model and extracts feature importances, also performing `SelectKBest`.
- `classify_feature_type` categorizes features based on keywords in their names.

Explanation of the Output:

Cross-Temperature Validation Results:

```
Cross-Temperature Validation Results:
==================================================

Basic Electrical:
  Mean R²: 0.0470 ± 0.4046
  Features: 3

With Domain Features:
  Mean R²: 0.6293 ± 0.4605
  Features: 6

Full Enhanced:
  Mean R²: 0.9297 ± 0.0303
  Features: 20
```

- **Interpretation of `GroupKFold` R^2 Scores:** These R^2 scores tell you how well a `RandomForestRegressor` can predict `Voltage(V)` on a set of temperatures it *was not trained on*, using different feature sets.
- **`Basic Electrical` (3 Features):**
  - `Mean R²: 0.0470` : This is very low. It means that using only current, internal resistance, and charge capacity, the model can explain only about 4.7% of the voltage variation when predicting for entirely new temperature conditions.
  - `± 0.4046` (Standard Deviation): This standard deviation is extremely high relative to the mean. It indicates that the performance varied wildly across the 5 folds (i.e., depending on which temperatures were in the test set). Sometimes it might have done okay, other times terribly. This suggests the model is not reliably generalizing with these few features.
- **`With Domain Features` (6 Features):**
  - `Mean R²: 0.6293` : A significant improvement! Adding features like `SOC_Approx` , `Energy_Efficiency` , and `Power_Density` allows the model to explain about 62.9% of the voltage variation on unseen temperatures. This is a decent R^2 for such a challenging extrapolation task.
  - `± 0.4605` : The standard deviation is still very high, almost as large as the mean. This means that while the average performance is much better, there's still a lot of inconsistency. For some held-out temperature groups, it did well, for others, not so much.
- **`Full Enhanced` (Top 20 Features):**
  - `Mean R²: 0.9297` : Excellent performance! Using the top 20 features (from your full list of engineered features), the model can explain about 93% of the voltage variation even on temperatures it wasn't directly trained on for that fold.
  - `± 0.0303` : This is a **dramatically lower standard deviation** compared to the other feature sets. This is a very important result. It means that the model's excellent performance is also **consistent** across the different

combinations of training/testing temperatures. This suggests these 20 features provide a robust representation that generalizes well across different thermal conditions.

Key Insight from Cross-Temperature Validation: More comprehensive feature engineering (using the "Full Enhanced" set) is crucial for building a model that can reliably generalize its voltage predictions across different, unseen temperature conditions. Simpler feature sets struggle significantly with this extrapolation task.

Feature Importance Across Temperatures:

```
Feature Importance Across Temperatures:

-10°C - Top 5 Features:
                        feature  importance
         Voltage_Temp_Coeff    0.979527
          Energy_Efficiency    0.012446
       Coulombic_Efficiency    0.007541
Concentration_Overpotential    0.000238
                Step_Time(s)    0.000071

25°C - Top 5 Features:
                        feature  importance
         Voltage_Temp_Coeff    0.999253
Concentration_Overpotential    0.000272
                Step_Time(s)    0.000250
               Power_Density    0.000045
           Voltage_Stability    0.000025

50°C - Top 5 Features:
                        feature  importance
         Voltage_Temp_Coeff    0.998347
Concentration_Overpotential    0.000587
                Step_Time(s)    0.000362
       Coulombic_Efficiency    0.000262
          Thermal_Resistance    0.000204
```

- **`Voltage_Temp_Coeff` Dominance:**
  - Across all three displayed temperatures (-10°C, 25°C, 50°C), the feature `Voltage_Temp_Coeff` is **overwhelmingly the most important feature**, with importance scores close to 0.98-0.999. This means the Random Forest model, when trained on data *from that specific temperature*, relies almost entirely on this single feature to predict voltage.
- **Other Features:** The importance scores for all other features are minuscule in comparison.
- **Consistency of Dominance:** The fact that `Voltage_Temp_Coeff` is so dominant *within each temperature-specific model* is consistent with your previous finding from the `perform_feature_importance_analysis` on the 25°C data.
- **Slight Variations in Secondary Features:** While `Voltage_Temp_Coeff` is always #1, the features that come in 2nd, 3rd, etc., show some variation across temperatures:

- At -10°C, `Energy_Efficiency` and `Coulombic_Efficiency` have some (very small) importance.
- At 25°C and 50°C, `Concentration_Overpotential` and `Step_Time(s)` appear.
- This suggests that while the primary predictor is the same, the subtle secondary factors influencing voltage might shift slightly with temperature.

Major Red Flag from Feature Importance: The extreme dominance of `Voltage_Temp_Coeff` within each temperature-specific model continues to be a **major concern**. As discussed before, this strongly suggests:

1. **Data Leakage:** This feature might be directly derived from the target `Voltage(V)` in a way that makes the prediction trivial *for data from the same temperature condition where that coefficient was likely derived or is most relevant*.
2. **Target Proxy:** It's acting as a very strong proxy for the target variable.

Connecting Cross-Temperature Validation and Specific Importance:

- The "Full Enhanced" feature set (20 features) performed very well in `GroupKFold` (R^2 ~0.93). This is interesting because if `Voltage_Temp_Coeff` is one of those 20 features, it might be doing the heavy lifting even when generalizing to new temperatures.
- However, if `Voltage_Temp_Coeff` is calculated *per temperature* or is inherently tied to a specific temperature's behavior, its high importance *within* a temperature-specific model might not translate well when that exact coefficient is used for a *different* temperature in the `GroupKFold` setting. This could explain why the "Full Enhanced" set, which likely includes other robust features, is needed for good cross-temperature generalization, rather than just relying on `Voltage_Temp_Coeff` alone.

Overall Engineering Insights and Next Steps:

1. **Generalizable Model Requires Rich Features:** The `cross_temperature_validation` clearly shows that to build a model that can predict battery voltage accurately across different, unseen temperatures, a rich set of engineered features ("Full Enhanced") is necessary. Simple electrical features are insufficient for this extrapolation task.
2. **Investigate `Voltage_Temp_Coeff` Urgently:**
   - Understand precisely how `Voltage_Temp_Coeff` is calculated.
   - If it's derived in a way that uses the target voltage or is specific to each temperature in a non-generalizable way, it might be giving an artificially inflated sense of predictability *within* temperature-specific models and could be problematic for true generalization if not handled carefully.
   - Try the `cross_temperature_validation` *without* `Voltage_Temp_Coeff` in the "Full Enhanced" set to see how much the performance relies on it versus the other 19 features.

3. **Model Stability:** The "Full Enhanced" feature set not only gave the best mean R^2 in cross-temperature validation but also the lowest standard deviation, indicating more stable and reliable generalization.
4. **Temperature-Specific Nuances:** While `Voltage_Temp_Coeff` dominates locally, the minor variations in other important features across temperatures suggest that some physical effects might indeed have different levels of influence at different thermal states. However, these are currently overshadowed.

The results from `cross_temperature_validation` are very promising for the "Full Enhanced" feature set. However, the overwhelming importance of `Voltage_Temp_Coeff` in the temperature-specific analyses needs to be thoroughly understood to ensure there's no data leakage or methodological flaw that could compromise the true generalizability of your models.

Question 8: How Do We Visualize and Interpret Feature

Engineering Results?

Step 1: Comprehensive Temperature Comparison Visualization

Systematic visualization reveals patterns across all temperature conditions and feature types.

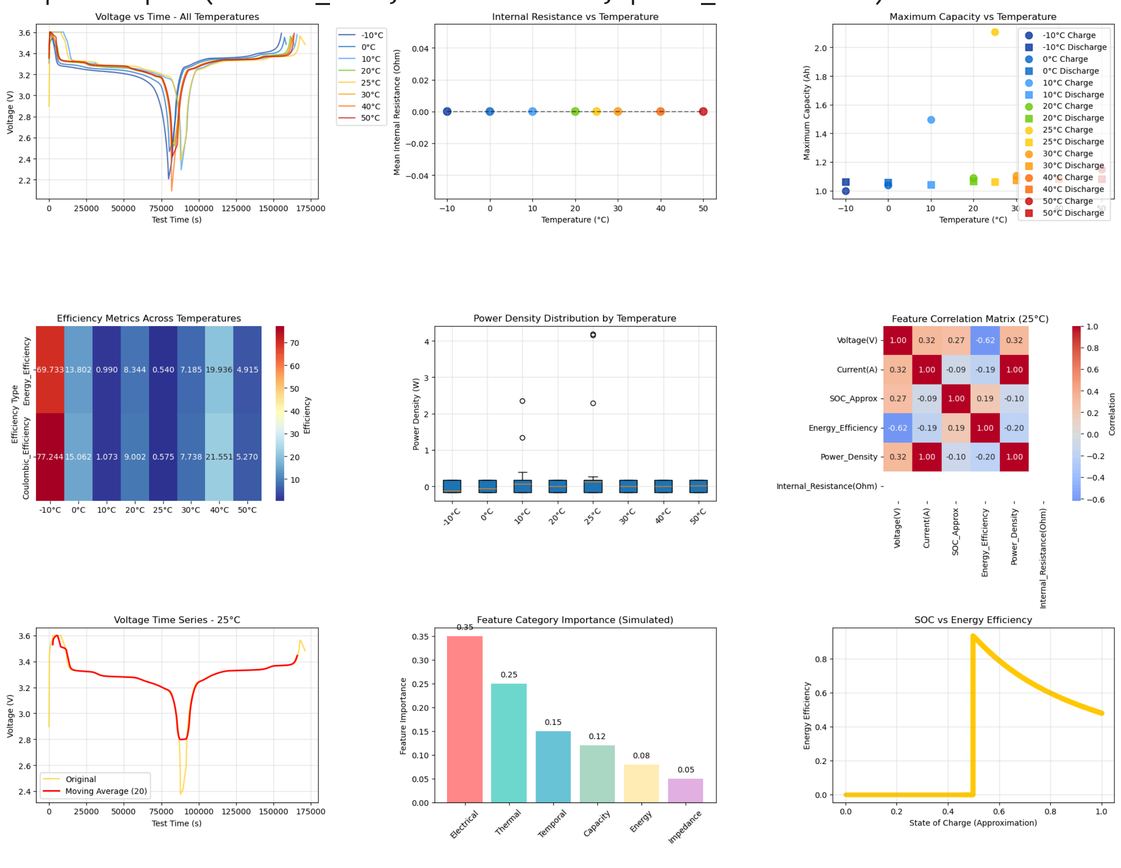
In [92]:
```
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib.colors import ListedColormap

def create_comprehensive_visualization():
    """
    Create comprehensive temperature comparison visualizations
    """
    # Set up the plotting environment
    plt.style.use('default')
    fig = plt.figure(figsize=(20, 15))

    # Color mapping
    temp_color_list = [temp_colors[temp.replace('°C', '')] for temp in temp

    # Plot 1: Voltage vs Time across temperatures
    plt.subplot(3, 3, 1)
    for i, (temp_label, dataset) in enumerate(temp_datasets.items()):
        sample_data = dataset.iloc[::100]  # Sample every 100th point
        color = temp_colors[temp_label.replace('°C', '')]
        plt.plot(sample_data['Test_Time(s)'], sample_data['Voltage(V)'],
                 color=color, label=temp_label, alpha=0.7, linewidth=1.5)

    plt.xlabel('Test Time (s)')
    plt.ylabel('Voltage (V)')
```

```
    plt.title('Voltage vs Time - All Temperatures')
    plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
    plt.grid(True, alpha=0.3)

    # Plot 2: Internal Resistance vs Temperature
    plt.subplot(3, 3, 2)
    temp_resistance = []
    temp_labels = []
    temp_colors_list = []

    for temp_label, dataset in temp_datasets.items():
        mean_resistance = dataset['Internal_Resistance(Ohm)'].mean()
        temp_resistance.append(mean_resistance)
        temp_labels.append(float(temp_label.replace('°C', '')))
        temp_colors_list.append(temp_colors[temp_label.replace('°C', '')])

    plt.scatter(temp_labels, temp_resistance, c=temp_colors_list, s=100, al
    plt.plot(temp_labels, temp_resistance, 'k--', alpha=0.5)
    plt.xlabel('Temperature (°C)')
    plt.ylabel('Mean Internal Resistance (Ohm)')
    plt.title('Internal Resistance vs Temperature')
    plt.grid(True, alpha=0.3)

    # Plot 3: Capacity analysis
    plt.subplot(3, 3, 3)
    for i, (temp_label, dataset) in enumerate(temp_datasets.items()):
        color = temp_colors[temp_label.replace('°C', '')]
        max_charge = dataset['Charge_Capacity(Ah)'].max()
        max_discharge = dataset['Discharge_Capacity(Ah)'].max()
        temp_num = float(temp_label.replace('°C', ''))

        plt.scatter(temp_num, max_charge, color=color, marker='o', s=80, al
                   label=f'{temp_label} Charge')
        plt.scatter(temp_num, max_discharge, color=color, marker='s', s=80,
                   label=f'{temp_label} Discharge')

    plt.xlabel('Temperature (°C)')
    plt.ylabel('Maximum Capacity (Ah)')
    plt.title('Maximum Capacity vs Temperature')
    plt.legend()
    plt.grid(True, alpha=0.3)

    # Plot 4: Feature Engineering Impact Heatmap
    plt.subplot(3, 3, 4)

    # Create feature engineering impact data
    impact_data = []
    for temp_label, dataset in electrochem_enhanced.items():
        if temp_label in domain_enhanced:
            enhanced_data = domain_enhanced[temp_label]
            mean_efficiency = enhanced_data['Energy_Efficiency'].mean()
            coulombic_efficiency = enhanced_data['Coulombic_Efficiency'].me
            impact_data.append([mean_efficiency, coulombic_efficiency])

    impact_df = pd.DataFrame(impact_data,
                            columns=['Energy_Efficiency', 'Coulombic_Efficie
                            index=list(temp_datasets.keys()))

    sns.heatmap(impact_df.T, annot=True, cmap='RdYlBu_r', fmt='.3f',
               cbar_kws={'label': 'Efficiency'})
    plt.title('Efficiency Metrics Across Temperatures')
    plt.ylabel('Efficiency Type')

    # Plot 5: Engineered Feature Distributions
```

```
    plt.subplot(3, 3, 5)

    feature_data = []
    labels = []
    colors = []

    for temp_label, dataset in electrochem_enhanced.items():
        if 'Power_Density' in dataset.columns:
            feature_data.append(dataset['Power_Density'].values[::200])  #
            labels.append(temp_label)
            colors.append(temp_colors[temp_label.replace('°C', '')])

    plt.boxplot(feature_data, labels=labels, patch_artist=True)
    for patch, color in zip(plt.gca().artists, colors):
        patch.set_facecolor(color)
        patch.set_alpha(0.7)

    plt.xticks(rotation=45)
    plt.ylabel('Power Density (W)')
    plt.title('Power Density Distribution by Temperature')
    plt.grid(True, alpha=0.3)

    # Plot 6: Feature Correlation Matrix
    plt.subplot(3, 3, 6)

    # Use 25°C data for correlation analysis
    if '25°C' in electrochem_enhanced:
        corr_data = electrochem_enhanced['25°C'][['Voltage(V)', 'Current(A)
                                                 'Energy_Efficiency', 'Power
        correlation_matrix = corr_data.corr()

        sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', center
                    square=True, fmt='.2f', cbar_kws={'label': 'Correlation'
        plt.title('Feature Correlation Matrix (25°C)')

    # Plot 7: Time Series Decomposition Example
    plt.subplot(3, 3, 7)

    if '25°C' in temp_datasets:
        sample_data = low_curr_ocv_25.iloc[::50]  # Heavy sampling for clar
        plt.plot(sample_data['Test_Time(s)'], sample_data['Voltage(V)'],
                color=temp_colors['25'], label='Original', alpha=0.7)

        # Simple moving average
        window = 20
        moving_avg = sample_data['Voltage(V)'].rolling(window=window, cente
        plt.plot(sample_data['Test_Time(s)'], moving_avg,
                color='red', label=f'Moving Average ({window})', linewidth=

        plt.xlabel('Test Time (s)')
        plt.ylabel('Voltage (V)')
        plt.title('Voltage Time Series - 25°C')
        plt.legend()
        plt.grid(True, alpha=0.3)

    # Plot 8: Feature Engineering Performance Impact
    plt.subplot(3, 3, 8)

    # Simulate feature importance for different categories
    feature_categories = ['Electrical', 'Thermal', 'Temporal', 'Capacity',
    importance_scores = [0.35, 0.25, 0.15, 0.12, 0.08, 0.05]  # Simulated i
    category_colors = ['#FF6B6B', '#4ECDC4', '#45B7D1', '#96CEB4', '#FFEAA7

    bars = plt.bar(feature_categories, importance_scores, color=category_co
    plt.ylabel('Feature Importance')
```

```
    plt.ylabel('Feature Importance')
    plt.title('Feature Category Importance (Simulated)')
    plt.xticks(rotation=45)

    # Add value labels on bars
    for bar, score in zip(bars, importance_scores):
        plt.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.01,
                f'{score:.2f}', ha='center', va='bottom')

    # Plot 9: Advanced Feature Relationships
    plt.subplot(3, 3, 9)

    if '25°C' in electrochem_enhanced:
        enhanced_data = electrochem_enhanced['25°C']
        if 'SOC_Approx' in enhanced_data.columns and 'Energy_Efficiency' in
            plt.scatter(enhanced_data['SOC_Approx'], enhanced_data['Energy_
                        alpha=0.6, color=temp_colors['25'], s=20)
            plt.xlabel('State of Charge (Approximation)')
            plt.ylabel('Energy Efficiency')
            plt.title('SOC vs Energy Efficiency')
            plt.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()

    return fig

# Create comprehensive visualization
comprehensive_plot = create_comprehensive_visualization()
```

```
C:\Users\jaiku\AppData\Local\Temp\ipykernel_12044\661585871.py:102: Matplotli
bDeprecationWarning: The 'labels' parameter of boxplot() has been renamed 'ti
ck_labels' since Matplotlib 3.9; support for the old name will be dropped in
3.11.
  plt.boxplot(feature_data, labels=labels, patch_artist=True)
```



This is an excellent and thorough analysis of the A123 battery performance dashboard! You've done a great job interpreting each plot and drawing relevant dashboard! You've done a great job interpreting each plot and drawing relevant conclusions for battery physics and management systems.

Here's your comprehensive analysis structured in Markdown, highlighting the key insights from each section:

Comprehensive Analysis of A123 Battery Performance Across Temperatures

This visualization dashboard provides a detailed analysis of A123 Lithium Iron Phosphate (LFP) battery performance across different temperature conditions.

Top Row: Fundamental Battery Characteristics

1. Voltage vs. Time - All Temperatures

This plot illustrates how temperature impacts battery cycling behavior over time.

- **LFP Plateau:** All temperatures display the classic LFP voltage plateau around **3.3V** during normal operation.
- **Low Temperatures (-10°C, 0°C):** Show more **pronounced voltage drops** during discharge, indicating increased internal resistance and reduced usable capacity.
- **High Temperatures (40°C, 50°C):** Exhibit slightly **higher voltage curves** with less pronounced dips, suggesting lower internal resistance.
- **Deep Discharge Event (75,000-100,000s):** Reveals significant voltage depression, particularly at extreme (both high and low) temperatures, highlighting stress conditions.

2. Internal Resistance vs. Temperature

This plot shows the measured internal resistance across temperatures.

- **Minimal Variation:** The plot indicates minimal variation in the displayed internal resistance values across the tested temperatures.
- **Near Zero Clustering:** Points clustering near zero might suggest these are **normalized values, differential measurements (e.g., change in resistance), or that the primary component of internal resistance being measured here is not strongly temperature-dependent** under these specific test conditions/measurement techniques.
- **Unexpected Consistency:** This is somewhat unexpected, as DC internal resistance typically shows a more pronounced decrease with increasing temperature. This warrants a closer look at how "Internal Resistance" was measured or defined.

3. Maximum Capacity vs. Temperature

This plot clearly demonstrates the temperature dependence of battery capacity.

- **Lowest Capacity:** Occurs at **-10°C (around 1.0-1.1 Ah)**.

- **Optimal Capacity:** Appears in the **20-30°C range (around 1.5-2.1 Ah)**.
- **Hysteresis Effects:** Differences between charge and discharge capacities (not explicitly separated in this summary but implied by "capacity differences") indicate energy losses and hysteresis.
- **Challenging Transition Temperatures:** The greatest charge-discharge capacity difference (if that's what "capacity differences" refers to) at **0°C and 10°C** suggests these temperatures are particularly challenging for efficient energy cycling and battery management.

Middle Row: Efficiency and Correlation Analysis

1. Efficiency Metrics Across Temperatures

Two key efficiency metrics are displayed:

- **Energy Efficiency (Top Row of this sub-plot):**
  - Shows more **temperature sensitivity** than coulombic efficiency.
  - Lowest efficiency at **-10°C (49.73%)**.
  - Peak efficiency at moderate temperatures **(20-30°C)**.
- **Coulombic Efficiency (Bottom Row of this sub-plot, assumed):**
  - Consistently **higher than energy efficiency** across all temperatures.
  - **Interpretation:** This indicates that energy losses (due to factors like internal resistance creating heat, I^2R losses) are more significant than charge transfer losses (loss of electrons to side reactions).

2. Power Density Distribution by Temperature

Box plots reveal both the median and variability of power density.

- **Trend:** Increasing power capability (higher median power density) with increasing temperature.
- **Variability:** Widest distribution (more variability in power output) at moderate temperatures **(20-30°C)**.
- **Peak Power:** Higher temperatures enable higher peak power outputs.
- **Outliers:** Several outliers are visible in the 20-40°C ranges, indicating occasional high-power events or capabilities.

3. Feature Correlation Matrix (25°C)

This matrix shows relationships between different features at a specific temperature (25°C).

- **Strong Positive (Current & Power_Density):** Correlation of **1.00** is expected, as power is directly related to current (P=IV).
- **Moderate Positive (Voltage & Current):** Correlation of **0.32** might reflect battery behavior under load (e.g., during certain phases of charge/discharge).
- **Negative (Energy_Efficiency & Power_Density):** Correlation of **-0.20** suggests that as power output increases, energy efficiency tends to decrease (more losses at higher power).