Feature Engineering for A123 Battery Open Circuit Voltage Analysis

A Comprehensive Demonstration Using Multi-Temperature Dataset

---

Table of Contents

---

1. Introduction to Feature Engineering {#introduction}

Feature engineering is the process of transforming raw data into meaningful numerical representations that machine learning algorithms can effectively utilize. As outlined in fundamental machine learning principles, this process involves converting arbitrary data into well-behaved vectors suitable for analysis.

Key Principles

- **Vectorization**: Converting non-numerical data into numerical format
- **Transformation**: Creating derived features that capture underlying patterns
- **Normalization**: Ensuring features are on comparable scales
- **Encoding**: Properly representing categorical information

A123 Battery Context

In battery analysis, feature engineering is crucial for:

- Understanding temperature effects on battery performance
- Predicting battery behavior under different conditions
- Identifying degradation patterns
- Optimizing charging/discharging strategies

2. Dataset Overview {#dataset-overview}

Temperature Range Analysis

Our dataset contains A123 battery low current OCV measurements across 8 temperature conditions:

```python
# 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
    '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
}
```

Core Features Available

- **Temporal**: Test_Time(s), Step_Time(s), Date_Time
- **Electrical**: Current(A), Voltage(V), Internal_Resistance(Ohm)
- **Capacity**: Charge_Capacity(Ah), Discharge_Capacity(Ah)
- **Energy**: Charge_Energy(Wh), Discharge_Energy(Wh)
- **Dynamics**: dV/dt(V/s), AC_Impedance(Ohm), ACI_Phase_Angle(Deg)
- **Environmental**: Temperature(C)_1, Temperature(C)_2

---

3. Categorical Feature Engineering {#categorical-features}

3.1 Temperature as Categorical Variable

Temperature is naturally categorical in this context, representing distinct operating conditions rather than continuous thermal states.

python

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

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=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_Category'])

    # 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_categorical_features()

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

3.2 Step Index Categorical Encoding

Battery test steps represent different operational phases and should be treated categorically.

python

```python
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_features(
    low_curr_ocv_minus_10
)

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

---

4. Derived Feature Engineering {#derived-features}

4.1 Polynomial Features for Voltage-Current Relationships

Following the polynomial feature engineering principles, we can capture non-linear relationships in battery behavior.

python

```python
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_Resistance(Ohm)']].val

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

    # Get feature names
    feature_names = poly_transformer.get_feature_names_out(['Voltage', 'Current', 'Resistance']

    return poly_features, feature_names, poly_transformer

def demonstrate_polynomial_regression():
    """
    Demonstrate polynomial feature engineering for battery voltage prediction
    """

    # 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_colors['25'], label='Ac
        plt.plot(X_sample.flatten(), y_pred, color='red', linewidth=2, label=f'Poly Degree {deg
        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_features(low_curr_ocv_2
print(f"Polynomial features created: {len(poly_names)} features from 3 original features")
print(f"Sample feature names: {poly_names[:10]}")
```

4.2 Power and Energy Derived Features

python

```python
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_features['Current(A

    # Energy efficiency ratios
    derived_features['Charge_Efficiency'] = (
        derived_features['Charge_Energy(Wh)'] / (derived_features['Charge_Capacity(Ah)'] + 1e-8
    )

    derived_features['Discharge_Efficiency'] = (
        derived_features['Discharge_Energy(Wh)'] / (derived_features['Discharge_Capacity(Ah)']
    )

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

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

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

    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 feature example for 25°C:")
print(enhanced_datasets['25°C'][['Power(W)', 'Charge_Efficiency', 'Conductance(S)']].head())
```

---

5. Time-Series Feature Engineering {#time-series-features}

5.1 Temporal Pattern Extraction

python

```python
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'] / 24)
    temporal_data['Hour_Cos'] = np.cos(2 * np.pi * temporal_data['Hour'] / 24)

    # Test duration features
    temporal_data['Test_Duration_Hours'] = temporal_data['Test_Time(s)'] / 3600
    temporal_data['Step_Duration_Minutes'] = temporal_data['Step_Time(s)'] / 60

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

    # 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_ROC']].head(10))
```

5.2 Cycle-Based Feature Engineering

python

```python
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:")
print(cycle_stats.head())
```

---

6. Missing Data Imputation {#missing-data-imputation}

6.1 Systematic Missing Data Analysis

python

```python
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(dataset[numerical_cols])

    # 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(dataset[numerical_col

    # 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[numerical_cols])

    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_25)
print("\nImputation completed for multiple strategies")
```

6.2 Advanced Imputation for Time Series

python

```
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='linear')

    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")
```

7. Feature Pipeline Implementation {#feature-pipelines}

7.1 Comprehensive Feature Engineering Pipeline

python

```python
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_degree, include_bias=Fa
            ('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=False, handle_unknown='ignore'))
        ])

        return categorical_pipeline

    def build_complete_pipeline(self, dataset):
        """Build complete feature engineering pipeline"""
```

```
        # Define feature groups
        electrical_features = ['Voltage(V)', 'Current(A)', 'Internal_Resistance(Ohm)']
        capacity_features = ['Charge_Capacity(Ah)', 'Discharge_Capacity(Ah)']
        energy_features = ['Charge_Energy(Wh)', 'Discharge_Energy(Wh)']
        temporal_features = ['Test_Time(s)', 'Step_Time(s)']

        # Create column transformer
        preprocessor = ColumnTransformer([
            ('electrical', self.create_electrical_pipeline(), electrical_features),
            ('capacity', self.create_temporal_pipeline(), capacity_features),
            ('energy', self.create_temporal_pipeline(), energy_features),
            ('temporal', self.create_temporal_pipeline(), temporal_features)
        ], 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)

        # 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_temporal=True)

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

print(f"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]}")
```

7.2 Temperature-Specific Pipeline

python

```python
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('°C', ''))
                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.fit_transform_dataset
                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_datasets)

# Create combined pipeline
X_combined, y_combined, combined_dataset = multi_temp_pipeline.create_combined_features(temp_da

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_dataset)} samples")
```

---

8. Advanced Feature Engineering {#advanced-features}

8.1 Domain-Specific Battery Features

python

```python
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)'] / max_capacity

    # 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_Phase_Angle(Deg)'])
    battery_features['Impedance_Real'] = (
        battery_features['AC_Impedance(Ohm)'] * np.cos(battery_features['Phase_Angle_Rad'])
```

```
    )
    battery_features['Impedance_Imaginary'] = (
        battery_features['AC_Impedance(Ohm)'] * np.sin(battery_features['Phase_Angle_Rad'])
    )

    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', 'Power_Density']
print(domain_enhanced['25°C'][domain_cols].describe())
```

8.2 Statistical Feature Engineering

python

```python
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).mean()
        )
        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).quantile(0.25)
        )
        stat_features[f'{var}_Rolling75th_{window_size}'] = (
            stat_features[var].rolling(window=window_size, center=True).quantile(0.75)
        )

    return stat_features

# Apply statistical feature engineering
statistical_enhanced = create_statistical_features(low_curr_ocv_25, window_size=10)
```

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

---

9. Visualization and Analysis {#visualization}

9.1 Temperature-Based Feature Visualization

python

```python
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib.colors import ListedColormap

def create_temperature_comparison_plots():
    """
    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_datasets.keys()]

    # 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, alpha=0.8)
    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, alpha=0.8, label=f'{te
    plt.scatter(temp_num, max_discharge, color=color, marker='s', s=80, alpha=0.8, label=f'

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: Energy Efficiency Heatmap
plt.subplot(3, 3, 4)
efficiency_data = []

for temp_label, dataset in temp_datasets.items():
    enhanced_data = create_battery_domain_features(dataset)
    mean_efficiency = enhanced_data['Energy_Efficiency'].mean()
    coulombic_efficiency = enhanced_data['Coulombic_Efficiency'].mean()
    efficiency_data.append([mean_efficiency, coulombic_efficiency])

efficiency_df = pd.DataFrame(efficiency_data,
                             columns=['Energy_Efficiency', 'Coulombic_Efficiency'],
                             index=list(temp_datasets.keys()))

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

# Plot 5: Voltage Distribution
plt.subplot(3, 3, 5)
voltage_data = []
labels = []
colors = []

for temp_label, dataset in temp_datasets.items():
    voltage_data.append(dataset['Voltage(V)'].values[::100])  # Sample data
    labels.append(temp_label)
```

```
    colors.append(temp_colors[temp_label.replace('°C', '')])

plt.boxplot(voltage_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('Voltage (V)')
plt.title('Voltage 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
corr_data = domain_enhanced['25°C'][['Voltage(V)', 'Current(A)', 'SOC_Approx',
                                     'Energy_Efficiency', 'Power_Density', 'Internal_Resista
correlation_matrix = corr_data.corr()

sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', center=0,
            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)
sample_data = low_curr_ocv_25.iloc[::50]  # Heavy sampling for clarity

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, center=True).mean()
plt.plot(sample_data['Test_Time(s)'], moving_avg,
         color='red', label=f'Moving Average ({window})', linewidth=2)

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: Power Analysis
plt.subplot(3, 3, 8)
for temp_label, dataset in temp_datasets.items():
    enhanced_data = create_battery_domain_features(dataset)
    sample_power = enhanced_data['Power_Density'].iloc[::200]  # Heavy sampling
```

```
        sample_time = enhanced_data['Test_Time(s)'].iloc[::200]
        color = temp_colors[temp_label.replace('°C', '')]

        plt.plot(sample_time, sample_power, color=color, alpha=0.6, linewidth=1, label=temp_lab

    plt.xlabel('Test Time (s)')
    plt.ylabel('Power Density (W)')
    plt.title('Power Density vs Time')
    plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
    plt.grid(True, alpha=0.3)

    # Plot 9: Feature Importance Simulation
    plt.subplot(3, 3, 9)

    # Simulate feature importance for different categories
    feature_categories = ['Electrical', 'Thermal', 'Temporal', 'Capacity', 'Energy', 'Impedance
    importance_scores = [0.35, 0.25, 0.15, 0.12, 0.08, 0.05]  # Simulated importance
    category_colors = ['#FF6B6B', '#4ECDC4', '#45B7D1', '#96CEB4', '#FFEAA7', '#DDA0DD']

    bars = plt.bar(feature_categories, importance_scores, color=category_colors, alpha=0.8)
    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')

    plt.tight_layout()
    plt.show()

    return fig

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

9.2 Feature Engineering Impact Analysis

python

```python
def analyze_feature_engineering_impact():
    """
    Analyze the impact of different feature engineering techniques
    """

    # Compare original vs engineered features for prediction
    from sklearn.ensemble import RandomForestRegressor
    from sklearn.model_selection import cross_val_score
    from sklearn.metrics import mean_squared_error, r2_score

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

    # Original features
    original_features = ['Current(A)', 'Internal_Resistance(Ohm)', 'Charge_Capacity(Ah)']
    X_original = original_data[original_features].fillna(original_data[original_features].mean(

    # Enhanced features
    enhanced_features = original_features + ['SOC_Approx', 'Energy_Efficiency', 'Power_Density'
    X_enhanced = enhanced_data[enhanced_features].fillna(enhanced_data[enhanced_features].mean(

    # 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, scoring='r2')

    # Enhanced features performance
    cv_scores_enhanced = cross_val_score(model, X_enhanced, y, cv=5, scoring='r2')

    # 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

# 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"  {metric}: {value}")
```

---

10. Conclusions {#conclusions}

10.1 Key Findings

Through comprehensive feature engineering applied to the A123 battery dataset, we have demonstrated several critical insights:

Temperature Effects on Battery Performance

- **Resistance Patterns**: Internal resistance shows clear temperature dependence, with higher resistance at lower temperatures
- **Capacity Variations**: Both charge and discharge capacities exhibit temperature-sensitive behavior
- **Efficiency Metrics**: Energy and coulombic efficiencies vary significantly across temperature ranges

Feature Engineering Benefits

1. **Categorical Encoding**: Temperature categorization enables clear performance segmentation
2. **Polynomial Features**: Non-linear relationships captured through polynomial expansion improve model accuracy
3. **Domain-Specific Features**: Battery-specific metrics (SOC, DOD, efficiency ratios) provide meaningful insights
4. **Temporal Features**: Time-series engineering reveals charging/discharging patterns

10.2 Best Practices Identified

Data Preprocessing

- **Imputation Strategy**: Mean imputation works well for continuous electrical measurements
- **Scaling**: StandardScaler essential for polynomial features to prevent dominance

- **Categorical Handling**: One-hot encoding preferred for temperature conditions

Feature Selection

- **Electrical Features**: Core measurements (voltage, current, resistance) remain most predictive
- **Derived Features**: Energy efficiency and power density add significant value
- **Temporal Features**: Rolling statistics capture important trends

10.3 Implementation Pipeline

python

```python
def create_production_pipeline():
    """
    Production-ready feature engineering pipeline for battery analysis
    """

    from sklearn.pipeline import Pipeline
    from sklearn.compose import ColumnTransformer
    from sklearn.preprocessing import StandardScaler, OneHotEncoder
    from sklearn.impute import SimpleImputer

    # Define feature groups
    electrical_features = ['Voltage(V)', 'Current(A)', 'Internal_Resistance(Ohm)']
    capacity_features = ['Charge_Capacity(Ah)', 'Discharge_Capacity(Ah)']
    energy_features = ['Charge_Energy(Wh)', 'Discharge_Energy(Wh)']
    categorical_features = ['Temperature_Category']

    # Create preprocessing pipeline
    preprocessor = ColumnTransformer([
        ('electrical', Pipeline([
            ('imputer', SimpleImputer(strategy='mean')),
            ('poly', PolynomialFeatures(degree=2, include_bias=False)),
            ('scaler', StandardScaler())
        ]), electrical_features),

        ('capacity', Pipeline([
            ('imputer', SimpleImputer(strategy='mean')),
            ('scaler', StandardScaler())
        ]), capacity_features),

        ('energy', Pipeline([
            ('imputer', SimpleImputer(strategy='mean')),
            ('scaler', StandardScaler())
        ]), energy_features),

        ('categorical', Pipeline([
            ('onehot', OneHotEncoder(drop='first', sparse=False))
        ]), categorical_features)
    ], remainder='drop')

    return preprocessor

# Create production pipeline
production_pipeline = create_production_pipeline()
print("Production pipeline created successfully")
```

10.4 Future Directions

Advanced Techniques

1. **Deep Feature Learning**: Neural network-based feature extraction
2. **Time Series Forecasting**: LSTM-based sequence modeling
3. **Anomaly Detection**: Isolation Forest for battery degradation detection
4. **Transfer Learning**: Cross-temperature model adaptation

Domain Extensions

1. **Multi-Battery Analysis**: Fleet-level feature engineering
2. **Real-time Processing**: Streaming feature computation
3. **Predictive Maintenance**: Degradation pattern recognition
4. **Optimization**: Feature selection for embedded systems

10.5 Technical Recommendations

For Implementation

- Use feature pipelines for reproducibility and scalability
- Implement cross-validation for robust performance evaluation
- Consider computational efficiency for real-time applications
- Maintain feature documentation for interpretability

For Analysis

- Regular validation against domain expertise
- Continuous monitoring of feature importance
- Adaptation to new battery chemistries and conditions
- Integration with physics-based models

References and Further Reading

1. **Scikit-Learn Documentation**: Feature Engineering and Preprocessing
2. **Battery Management Systems**: Fundamentals and Applications
3. **Time Series Analysis**: Methods and Applications
4. **Machine Learning for Energy Systems**: Advanced Techniques

---

This comprehensive feature engineering demonstration showcases the application of fundamental machine learning preprocessing techniques to real-world battery analysis data. The methodologies presented can be

adapted and extended for various battery types, operating conditions, and analytical objectives.

Dataset Information: A123 Battery Low Current OCV measurements across temperature range -10°C to 50°C

Implementation: Python with Scikit-Learn, Pandas, NumPy, and Matplotlib

Color Scheme: Temperature-based visualization for consistent interpretation across analyses