Scikit-Learn with A123 Battery Data

A Comprehensive Guide for Beginners

Table of Contents

---

1. Introduction: Why Machine Learning for Battery Data? {#introduction}

Imagine you're an engineer working with A123 lithium-ion batteries. You have data from tests at different temperatures ranging from freezing cold (-10°C) to quite hot (50°C). Your goal is to understand and predict how these batteries behave under various conditions.

This is where machine learning becomes incredibly powerful. Instead of manually analyzing thousands of data points, we can train algorithms to:

- **Predict** voltage based on temperature and capacity
- **Classify** optimal operating temperature ranges
- **Discover** hidden patterns in battery performance
- **Estimate** remaining battery life

What Makes Battery Data Special?

Battery data is perfect for learning machine learning because:

- **Relationships are clear**: Temperature affects performance in predictable ways
- **Multiple variables**: Current, voltage, capacity, and temperature all interact
- **Real-world relevance**: These insights directly impact electric vehicles, phones, and renewable energy systems

- **Rich dataset**: Multiple temperature conditions provide diverse scenarios

Learning Objectives

By the end of this guide, you'll understand how to:

1. Prepare battery data for machine learning
2. Build models to predict battery voltage and capacity
3. Classify batteries into performance categories
4. Find hidden patterns in multi-temperature data
5. Create a complete battery analysis pipeline

---

2. Understanding Our Battery Dataset {#data-understanding}

Before diving into machine learning, let's understand what we're working with. Our A123 battery dataset contains measurements from low-current OCV (Open Circuit Voltage) tests at eight different temperatures.

What is Open Circuit Voltage (OCV)?

Think of OCV as the battery's "resting voltage" - like measuring blood pressure when you're sitting still. It tells us the battery's true state without the complications of current flow.

Our Dataset Structure

We have eight datasets, each representing tests at a specific temperature:

```python
# Temperature datasets
temp_datasets = {
    '-10°C': low_curr_ocv_minus_10,  # Deep blue
    '0°C': low_curr_ocv_0,           # Blue
    '10°C': low_curr_ocv_10,         # Light blue
    '20°C': low_curr_ocv_20,         # Green
    '25°C': low_curr_ocv_25,         # Yellow (room temperature)
    '30°C': low_curr_ocv_30,         # Orange
    '40°C': low_curr_ocv_40,         # Dark orange
    '50°C': low_curr_ocv_50          # Red
}
```

Key Variables in Each Dataset

Each dataset contains these important columns:

- **Test_Time(s)**: How long the test has been running

- **Current(A)**: Electrical current (mostly zero for OCV tests)
- **Voltage(V)**: The battery's voltage output
- **Charge_Capacity(Ah)**: How much energy has been stored
- **Discharge_Capacity(Ah)**: How much energy has been released
- **Temperature(C)**: The exact temperature during the test
- **Cycle_Index**: Which charge/discharge cycle this is

Why This Data is Valuable

This dataset is a treasure trove because:

1. **Temperature range**: From -10°C to 50°C covers real-world conditions
2. **Multiple cycles**: We can see how batteries change over time
3. **Precision**: Measurements every few seconds give detailed behavior
4. **Controlled conditions**: Each temperature is tested thoroughly

---

3. Setting Up Our Environment {#setup}

Let's prepare our tools for battery data analysis and machine learning.

python

```python
# Essential libraries for data analysis
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Scikit-learn components
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
from sklearn.metrics import mean_squared_error, r2_score

# Set up visualization style
plt.style.use('seaborn-v0_8-whitegrid')
sns.set_palette("husl")

# 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
}
```

Understanding the Setup

Why these libraries?

- **pandas**: For handling our battery data tables
- **numpy**: For mathematical calculations
- **matplotlib & seaborn**: For creating clear visualizations
- **scikit-learn**: Our main machine learning toolkit

Why the color mapping?

Visual consistency helps us quickly identify temperature effects. Cold temperatures use blue tones, moderate temperatures use green and yellow, and hot temperatures use orange and red.

4. Data Representation in Scikit-Learn {#data-representation}

Now let's understand how to structure our battery data for machine learning. Scikit-learn expects data in a specific format that's actually quite intuitive once you understand it.

The Features Matrix (X) and Target Array (y)

Think of machine learning like this:

- **Features (X)**: The information we know (temperature, capacity, time)
- **Target (y)**: What we want to predict (voltage, remaining capacity)

python

```python
# Example: Preparing data to predict voltage from temperature and capacity
def prepare_battery_data():
    """
    Combine all temperature datasets into one comprehensive dataset
    """
    # List to store all data
    all_data = []

    # Process each temperature dataset
    for temp_str, df in temp_datasets.items():
        # Add temperature as a numeric column
        df_copy = df.copy()
        df_copy['Temperature_Numeric'] = float(temp_str)
        all_data.append(df_copy)

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

    # Remove any missing values
    combined_data = combined_data.dropna()

    return combined_data

# Create our master dataset
battery_data = prepare_battery_data()
print(f"Combined dataset shape: {battery_data.shape}")
print(f"Available columns: {battery_data.columns.tolist()}")
```

Example 1: Simple Voltage Prediction

Let's start with a simple question: "Can we predict battery voltage from temperature and charge capacity?"

python

```python
# Define our features (X) and target (y)
# Features: What we know
X = battery_data[['Temperature_Numeric', 'Charge_Capacity(Ah)']]

# Target: What we want to predict
y = battery_data['Voltage(V)']

print(f"Features shape: {X.shape}")
print(f"Target shape: {y.shape}")
print(f"Sample features:")
print(X.head())
```

Understanding the Data Layout

python

```python
# Visualize our data structure
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))

# Features visualization
ax1.scatter(X['Temperature_Numeric'], X['Charge_Capacity(Ah)'], alpha=0.5)
ax1.set_xlabel('Temperature (°C)')
ax1.set_ylabel('Charge Capacity (Ah)')
ax1.set_title('Our Features: Temperature vs Capacity')

# Target visualization
ax2.hist(y, bins=50, alpha=0.7)
ax2.set_xlabel('Voltage (V)')
ax2.set_ylabel('Frequency')
ax2.set_title('Our Target: Voltage Distribution')

plt.tight_layout()
plt.show()
```

Data Cleaning: Essential First Steps

Real-world data is messy. Here's how we clean our battery data:

python

```python
def clean_battery_data(df):
    """
    Clean battery data for machine learning
    """
    # Remove obvious outliers (voltage should be reasonable)
    df = df[(df['Voltage(V)'] > 0) & (df['Voltage(V)'] < 5)]

    # Remove negative capacities (physically impossible)
    df = df[df['Charge_Capacity(Ah)'] >= 0]

    # Remove extreme temperature measurements (sensor errors)
    df = df[(df['Temperature (C)_1'] > -15) & (df['Temperature (C)_1'] < 60)]

    return df

# Clean our data
battery_data_clean = clean_battery_data(battery_data)
print(f"Before cleaning: {battery_data.shape[0]} rows")
print(f"After cleaning: {battery_data_clean.shape[0]} rows")
```

Key Takeaways

1. **Features are inputs**: Temperature, capacity, time - things we can measure
2. **Targets are outputs**: Voltage, performance - things we want to predict
3. **Data cleaning matters**: Remove impossible values and outliers
4. **Shape is important**: Scikit-learn needs 2D arrays for features

---

5. The Estimator API with Battery Examples {#estimator-api}

Scikit-learn's beauty lies in its consistent API. Once you learn the pattern, you can apply it to any algorithm. Let's see this in action with our battery data.

The Universal Pattern

Every scikit-learn algorithm follows this pattern:

1. **Import** the algorithm
2. **Instantiate** with parameters
3. **Fit** to training data
4. **Predict** on new data

Example 1: Predicting Voltage with Linear Regression

Let's predict battery voltage based on temperature and capacity.

```python
# Step 1: Import the algorithm
from sklearn.linear_model import LinearRegression

# Step 2: Prepare our data
X = battery_data_clean[['Temperature_Numeric', 'Charge_Capacity(Ah)']]
y = battery_data_clean['Voltage(V)']

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

# Step 3: Instantiate the model
voltage_predictor = LinearRegression()

# Step 4: Fit to training data
voltage_predictor.fit(X_train, y_train)

# Step 5: Make predictions
y_pred = voltage_predictor.predict(X_test)

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

print(f"Model Performance:")
print(f"Mean Squared Error: {mse:.4f}")
print(f"R² Score: {r2:.4f}")
```

Visualizing Our First Model

python

```python
# Visualize predictions vs actual values
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_pred, alpha=0.5)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--', lw=2)
plt.xlabel('Actual Voltage (V)')
plt.ylabel('Predicted Voltage (V)')
plt.title('Voltage Prediction: Actual vs Predicted')

# Add R² score to the plot
plt.text(0.05, 0.95, f'R² = {r2:.4f}', transform=plt.gca().transAxes,
         bbox=dict(boxstyle='round', facecolor='wheat', alpha=0.5))

plt.show()
```

Understanding Model Coefficients

Linear regression gives us interpretable coefficients:

python

```python
# Examine what our model learned
feature_names = ['Temperature (°C)', 'Charge Capacity (Ah)']
coefficients = voltage_predictor.coef_
intercept = voltage_predictor.intercept_

print("Our model equation:")
print(f"Voltage = {intercept:.4f}")
for name, coef in zip(feature_names, coefficients):
    print(f"    + {coef:.4f} × {name}")

# Create a bar plot of feature importance
plt.figure(figsize=(10, 6))
plt.bar(feature_names, coefficients)
plt.xlabel('Features')
plt.ylabel('Coefficient Value')
plt.title('Feature Importance in Voltage Prediction')
plt.show()
```

Example 2: Non-Linear Relationships with Random Forest

Sometimes relationships aren't linear. Let's try a Random Forest:

python

```python
# Import Random Forest
from sklearn.ensemble import RandomForestRegressor

# Create and train the model
rf_predictor = RandomForestRegressor(n_estimators=100, random_state=42)
rf_predictor.fit(X_train, y_train)

# Make predictions
rf_pred = rf_predictor.predict(X_test)

# Compare with linear regression
rf_r2 = r2_score(y_test, rf_pred)
print(f"Linear Regression R²: {r2:.4f}")
print(f"Random Forest R²: {rf_r2:.4f}")

# Feature importance in Random Forest
feature_importance = rf_predictor.feature_importances_
plt.figure(figsize=(10, 6))
plt.bar(feature_names, feature_importance)
plt.xlabel('Features')
plt.ylabel('Importance')
plt.title('Feature Importance in Random Forest Model')
plt.show()
```

Key Insights

1. **Consistent API**: Same pattern works for any algorithm
2. **Model comparison**: Easy to try different approaches
3. **Interpretability**: Linear models show clear relationships
4. **Flexibility**: Random forests capture complex patterns

---

6. Supervised Learning: Predicting Battery Behavior {#supervised-learning}

Supervised learning is like learning with a teacher - we show the algorithm examples of inputs and correct outputs. Let's explore both regression (predicting numbers) and classification (predicting categories) with our battery data.

Regression Example: Predicting Battery Capacity

Can we predict how much capacity a battery has based on its voltage and temperature?

python

```python
# Prepare data for capacity prediction
# Features: voltage and temperature
X_capacity = battery_data_clean[['Voltage(V)', 'Temperature_Numeric']]
# Target: remaining charge capacity
y_capacity = battery_data_clean['Charge_Capacity(Ah)']

# Split the data
X_train_cap, X_test_cap, y_train_cap, y_test_cap = train_test_split(
    X_capacity, y_capacity, test_size=0.2, random_state=42
)

# Train multiple models to compare
models = {
    'Linear Regression': LinearRegression(),
    'Random Forest': RandomForestRegressor(n_estimators=100, random_state=42),
}

# Train and evaluate each model
results = {}
for name, model in models.items():
    # Train the model
    model.fit(X_train_cap, y_train_cap)

    # Make predictions
    pred = model.predict(X_test_cap)

    # Calculate metrics
    mse = mean_squared_error(y_test_cap, pred)
    r2 = r2_score(y_test_cap, pred)

    results[name] = {'MSE': mse, 'R2': r2, 'predictions': pred}

    print(f"{name}:")
    print(f"  MSE: {mse:.4f}")
    print(f"  R²: {r2:.4f}")
    print()
```

Visualizing Model Performance

python

```python
# Create comparison plot
fig, axes = plt.subplots(1, 2, figsize=(15, 6))

for i, (name, result) in enumerate(results.items()):
    ax = axes[i]
    ax.scatter(y_test_cap, result['predictions'], alpha=0.5)
    ax.plot([y_test_cap.min(), y_test_cap.max()],
            [y_test_cap.min(), y_test_cap.max()], 'r--', lw=2)
    ax.set_xlabel('Actual Capacity (Ah)')
    ax.set_ylabel('Predicted Capacity (Ah)')
    ax.set_title(f'{name}\nR² = {result["R2"]:.4f}')

plt.tight_layout()
plt.show()
```

Classification Example: Temperature Range Classification

Let's classify batteries into temperature ranges based on their performance characteristics.

python

```python
# Create temperature range categories
def categorize_temperature(temp):
    if temp < 0:
        return 'Cold'
    elif temp < 25:
        return 'Cool'
    elif temp < 40:
        return 'Warm'
    else:
        return 'Hot'

# Prepare classification data
battery_data_clean['Temp_Category'] = battery_data_clean['Temperature_Numeric'].apply(categoriz

# Features for classification
X_class = battery_data_clean[['Voltage(V)', 'Charge_Capacity(Ah)', 'Discharge_Capacity(Ah)']]
y_class = battery_data_clean['Temp_Category']

# Split the data
X_train_class, X_test_class, y_train_class, y_test_class = train_test_split(
    X_class, y_class, test_size=0.2, random_state=42, stratify=y_class
)

# Scale features for better performance
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_class)
X_test_scaled = scaler.transform(X_test_class)

# Train a Random Forest classifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, confusion_matrix

rf_classifier = RandomForestClassifier(n_estimators=100, random_state=42)
rf_classifier.fit(X_train_scaled, y_train_class)

# Make predictions
y_pred_class = rf_classifier.predict(X_test_scaled)

# Evaluate the classifier
print("Classification Report:")
print(classification_report(y_test_class, y_pred_class))
```

Visualizing Classification Results

python

```python
# Create confusion matrix
cm = confusion_matrix(y_test_class, y_pred_class)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=rf_classifier.classes_,
            yticklabels=rf_classifier.classes_)
plt.xlabel('Predicted Temperature Category')
plt.ylabel('Actual Temperature Category')
plt.title('Temperature Category Classification Results')
plt.show()

# Feature importance for classification
feature_names_class = ['Voltage (V)', 'Charge Capacity (Ah)', 'Discharge Capacity (Ah)']
importances = rf_classifier.feature_importances_

plt.figure(figsize=(10, 6))
plt.bar(feature_names_class, importances)
plt.xlabel('Features')
plt.ylabel('Importance')
plt.title('Feature Importance for Temperature Classification')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

Real-World Application: Battery State Estimation

python

```python
# Example: Predict battery state for a new measurement
def predict_battery_state(voltage, temp, charge_capacity):
    """
    Predict battery temperature category based on measurements
    """
    # Prepare the input
    new_data = np.array([[voltage, charge_capacity, 0]])  # 0 for unknown discharge
    new_data_scaled = scaler.transform(new_data)

    # Make prediction
    prediction = rf_classifier.predict(new_data_scaled)[0]
    probability = rf_classifier.predict_proba(new_data_scaled)[0]

    print(f"Battery measurements:")
    print(f"  Voltage: {voltage}V")
    print(f"  Temperature: {temp}°C")
    print(f"  Charge Capacity: {charge_capacity}Ah")
    print(f"\nPredicted temperature category: {prediction}")
    print(f"Prediction confidence:")
    for cat, prob in zip(rf_classifier.classes_, probability):
        print(f"  {cat}: {prob:.2%}")

# Test with example measurements
predict_battery_state(voltage=3.5, temp=25, charge_capacity=1.0)
```

Key Takeaways

1. **Regression predicts numbers**: Useful for capacity, voltage, lifetime prediction
2. **Classification predicts categories**: Useful for temperature ranges, health status
3. **Feature scaling matters**: Especially for algorithms like SVM or neural networks
4. **Model comparison is essential**: Different algorithms excel at different problems

---

7. Unsupervised Learning: Finding Patterns {#unsupervised-learning}

Unsupervised learning is like being a detective - we look for hidden patterns in data without knowing what we're supposed to find. This is incredibly valuable for battery data because it can reveal insights we never thought to look for.

Clustering: Grouping Similar Battery Behaviors

Let's discover if batteries naturally group into different performance patterns.

python

```python
# Prepare data for clustering
# Select features that represent battery performance
cluster_features = ['Voltage(V)', 'Charge_Capacity(Ah)', 'Temperature_Numeric']
X_cluster = battery_data_clean[cluster_features].copy()

# Remove any remaining NaN values
X_cluster = X_cluster.dropna()

# Scale the features for clustering
scaler_cluster = StandardScaler()
X_cluster_scaled = scaler_cluster.fit_transform(X_cluster)

# Apply K-means clustering
from sklearn.cluster import KMeans

# Try different numbers of clusters
n_clusters_range = range(2, 8)
inertias = []

for n_clusters in n_clusters_range:
    kmeans = KMeans(n_clusters=n_clusters, random_state=42, n_init=10)
    kmeans.fit(X_cluster_scaled)
    inertias.append(kmeans.inertia_)

# Plot elbow curve to find optimal number of clusters
plt.figure(figsize=(10, 6))
plt.plot(n_clusters_range, inertias, 'bo-')
plt.xlabel('Number of Clusters')
plt.ylabel('Inertia')
plt.title('Elbow Method for Optimal Number of Clusters')
plt.grid(True)
plt.show()

# Use 4 clusters based on elbow curve
optimal_clusters = 4
kmeans_final = KMeans(n_clusters=optimal_clusters, random_state=42, n_init=10)
cluster_labels = kmeans_final.fit_predict(X_cluster_scaled)

# Add cluster labels to our data
X_cluster['Cluster'] = cluster_labels
```

Visualizing Battery Clusters

python

```python
# Create 3D scatter plot of clusters
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(12, 9))
ax = fig.add_subplot(111, projection='3d')

# Plot each cluster with different colors
colors = ['red', 'blue', 'green', 'purple']
for i in range(optimal_clusters):
    cluster_data = X_cluster[X_cluster['Cluster'] == i]
    ax.scatter(cluster_data['Voltage(V)'],
               cluster_data['Charge_Capacity(Ah)'],
               cluster_data['Temperature_Numeric'],
               c=colors[i], label=f'Cluster {i}', alpha=0.6)

ax.set_xlabel('Voltage (V)')
ax.set_ylabel('Charge Capacity (Ah)')
ax.set_zlabel('Temperature (°C)')
ax.set_title('Battery Performance Clusters')
ax.legend()
plt.show()

# Analyze cluster characteristics
print("Cluster Analysis:")
for i in range(optimal_clusters):
    cluster_data = X_cluster[X_cluster['Cluster'] == i]
    print(f"\nCluster {i} (n={len(cluster_data)}):")
    print(f"  Average Voltage: {cluster_data['Voltage(V)'].mean():.3f}V")
    print(f"  Average Capacity: {cluster_data['Charge_Capacity(Ah)'].mean():.3f}Ah")
    print(f"  Average Temperature: {cluster_data['Temperature_Numeric'].mean():.1f}°C")
    print(f"  Temp Range: {cluster_data['Temperature_Numeric'].min():.0f}°C to {cluster_data['T
```

Dimensionality Reduction: Understanding Complex Relationships

Our battery data has many variables. PCA (Principal Component Analysis) helps us understand the most important patterns.

python

```python
# Apply PCA to understand main sources of variation
from sklearn.decomposition import PCA

# Use more features for a comprehensive analysis
pca_features = ['Voltage(V)', 'Charge_Capacity(Ah)', 'Discharge_Capacity(Ah)',
                'Temperature_Numeric', 'Test_Time(s)']

# Prepare data (using a subset for faster computation)
X_pca = battery_data_clean[pca_features].copy()
X_pca = X_pca.dropna()

# Scale the data
scaler_pca = StandardScaler()
X_pca_scaled = scaler_pca.fit_transform(X_pca)

# Apply PCA
pca = PCA()
X_pca_transformed = pca.fit_transform(X_pca_scaled)

# Plot explained variance ratio
plt.figure(figsize=(12, 5))

# Subplot 1: Explained variance by component
plt.subplot(1, 2, 1)
plt.bar(range(1, len(pca.explained_variance_ratio_) + 1),
        pca.explained_variance_ratio_)
plt.xlabel('Principal Component')
plt.ylabel('Explained Variance Ratio')
plt.title('Variance Explained by Each Principal Component')

# Subplot 2: Cumulative explained variance
plt.subplot(1, 2, 2)
cumulative_variance = np.cumsum(pca.explained_variance_ratio_)
plt.plot(range(1, len(cumulative_variance) + 1), cumulative_variance, 'bo-')
plt.axhline(y=0.95, color='r', linestyle='--', label='95% variance')
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance')
plt.title('Cumulative Variance Explained')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()
```

```
print(f"First 2 components explain {cumulative_variance[1]:.2%} of variance")
print(f"First 3 components explain {cumulative_variance[2]:.2%} of variance")
```

Visualizing PCA Results

python

```python
# Plot first two principal components colored by temperature
plt.figure(figsize=(12, 8))

# Create temperature ranges for coloring
temp_bins = pd.cut(X_pca['Temperature_Numeric'],
                   bins=[-15, 0, 20, 40, 60],
                   labels=['Cold (<0°C)', 'Cool (0-20°C)', 'Warm (20-40°C)', 'Hot (>40°C)'])

# Create scatter plot
for temp_range, color in zip(temp_bins.cat.categories, ['blue', 'green', 'orange', 'red']):
    mask = temp_bins == temp_range
    plt.scatter(X_pca_transformed[mask, 0], X_pca_transformed[mask, 1],
                alpha=0.6, label=temp_range, c=color)

plt.xlabel(f'First Principal Component (explains {pca.explained_variance_ratio_[0]:.1%} of vari
plt.ylabel(f'Second Principal Component (explains {pca.explained_variance_ratio_[1]:.1%} of var
plt.title('Battery Data in Principal Component Space')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

# Analyze component loadings
feature_names_pca = pca_features
loadings = pca.components_[:2]  # First two components

plt.figure(figsize=(10, 6))
x = np.arange(len(feature_names_pca))
width = 0.35

plt.bar(x - width/2, loadings[0], width, label='First Component', alpha=0.8)
plt.bar(x + width/2, loadings[1], width, label='Second Component', alpha=0.8)

plt.xlabel('Features')
plt.ylabel('Component Loading')
plt.title('Feature Contributions to Principal Components')
plt.xticks(x, feature_names_pca, rotation=45)
plt.legend()
plt.tight_layout()
plt.show()

print("Component interpretation:")
print("First component (largest variation source):")
for i, feature in enumerate(feature_names_pca):
    print(f"  {feature}: {loadings[0][i]:.3f}")

print("\nSecond component (second largest variation source):")
```

```
for i, feature in enumerate(feature_names_pca):
    print(f"  {feature}: {loadings[1][i]:.3f}")
```

Anomaly Detection: Finding Unusual Battery Behavior

python

```python
# Use Isolation Forest to detect anomalous battery behavior
from sklearn.ensemble import IsolationForest

# Prepare data for anomaly detection
anomaly_features = ['Voltage(V)', 'Charge_Capacity(Ah)', 'Temperature_Numeric']
X_anomaly = battery_data_clean[anomaly_features].copy()
X_anomaly = X_anomaly.dropna()

# Scale the data
scaler_anomaly = StandardScaler()
X_anomaly_scaled = scaler_anomaly.fit_transform(X_anomaly)

# Detect anomalies
iso_forest = IsolationForest(contamination=0.05, random_state=42)
anomaly_labels = iso_forest.fit_predict(X_anomaly_scaled)

# Add anomaly labels to data
X_anomaly['Anomaly'] = anomaly_labels

# Count normal vs anomalous points
print(f"Normal data points: {sum(anomaly_labels == 1)}")
print(f"Anomalous data points: {sum(anomaly_labels == -1)}")

# Visualize anomalies
fig, axes = plt.subplots(1, 3, figsize=(18, 6))

features_plot = ['Voltage(V)', 'Charge_Capacity(Ah)', 'Temperature_Numeric']
for i, feature in enumerate(features_plot):
    normal_data = X_anomaly[X_anomaly['Anomaly'] == 1]
    anomaly_data = X_anomaly[X_anomaly['Anomaly'] == -1]

    axes[i].hist(normal_data[feature], bins=50, alpha=0.7, label='Normal', color='blue')
    axes[i].hist(anomaly_data[feature], bins=50, alpha=0.7, label='Anomalous', color='red')
    axes[i].set_xlabel(feature)
    axes[i].set_ylabel('Frequency')
    axes[i].set_title(f'Distribution of {feature}')
    axes[i].legend()

plt.tight_layout()
plt.show()

# Examine some anomalous cases
print("\nSample anomalous measurements:")
print(X_anomaly[X_anomaly['Anomaly'] == -1].head())
```

Key Insights from Unsupervised Learning

1. **Clustering reveals natural groups**: Batteries group by temperature and performance
2. **PCA shows main variation sources**: Temperature and capacity are primary drivers
3. **Anomaly detection finds unusual behavior**: Helps identify potential sensor errors or interesting edge cases
4. **Pattern discovery**: Unsupervised methods reveal relationships we might not expect

---

8. Advanced Application: Battery Performance Predictor {#advanced-application}

Now let's build a comprehensive battery analysis system that combines everything we've learned. This will be like creating a "smart battery management system" that can predict, classify, and monitor battery behavior.

Building the Complete Battery Analyzer

python

```python
class BatteryAnalyzer:
    """
    Comprehensive battery analysis system combining multiple ML techniques
    """

    def __init__(self):
        self.voltage_predictor = None
        self.capacity_predictor = None
        self.temp_classifier = None
        self.scaler = None
        self.pca = None
        self.anomaly_detector = None
        self.is_trained = False

    def prepare_data(self, battery_data):
        """
        Prepare battery data for all analyses
        """
        # Clean the data
        data = battery_data.copy()
        data = data.dropna()
        data = data[(data['Voltage(V)'] > 0) & (data['Voltage(V)'] < 5)]
        data = data[data['Charge_Capacity(Ah)'] >= 0]

        return data

    def train(self, battery_data):
        """
        Train all models on the battery data
        """
        print("Training Battery Analyzer...")

        # Prepare data
        data = self.prepare_data(battery_data)

        # 1. Train voltage predictor
        print("  - Training voltage predictor...")
        X_voltage = data[['Temperature_Numeric', 'Charge_Capacity(Ah)']]
        y_voltage = data['Voltage(V)']
        self.voltage_predictor = RandomForestRegressor(n_estimators=100, random_state=42)
        self.voltage_predictor.fit(X_voltage, y_voltage)

        # 2. Train capacity predictor
        print("  - Training capacity predictor...")
        X_capacity = data[['Voltage(V)', 'Temperature_Numeric']]
        y_capacity = data['Charge_Capacity(Ah)']
```

```
        self.capacity_predictor = RandomForestRegressor(n_estimators=100, random_state=42)
        self.capacity_predictor.fit(X_capacity, y_capacity)

        # 3. Train temperature classifier
        print("  - Training temperature classifier...")
        # Create temperature categories
        def temp_category(temp):
            if temp < 0: return 'Cold'
            elif temp < 25: return 'Moderate'
            else: return 'Hot'

        data['Temp_Category'] = data['Temperature_Numeric'].apply(temp_category)
        X_class = data[['Voltage(V)', 'Charge_Capacity(Ah)', 'Discharge_Capacity(Ah)']]
        y_class = data['Temp_Category']

        self.scaler = StandardScaler()
        X_class_scaled = self.scaler.fit_transform(X_class)
        self.temp_classifier = RandomForestClassifier(n_estimators=100, random_state=42)
        self.temp_classifier.fit(X_class_scaled, y_class)

        # 4. Train PCA for dimensionality reduction
        print("  - Training PCA...")
        pca_features = ['Voltage(V)', 'Charge_Capacity(Ah)', 'Temperature_Numeric']
        X_pca = data[pca_features]
        X_pca_scaled = StandardScaler().fit_transform(X_pca)
        self.pca = PCA(n_components=2)
        self.pca.fit(X_pca_scaled)

        # 5. Train anomaly detector
        print("  - Training anomaly detector...")
        self.anomaly_detector = IsolationForest(contamination=0.05, random_state=42)
        self.anomaly_detector.fit(X_pca_scaled)

        self.is_trained = True
        print("Training complete!")

        # Calculate and display model performance
        self._evaluate_models(data)

    def _evaluate_models(self, data):
        """
        Evaluate model performance on training data
        """
        print("\nModel Performance Summary:")

        # Voltage prediction performance
        X_voltage = data[['Temperature_Numeric', 'Charge_Capacity(Ah)']]
```

```
        y_voltage = data['Voltage(V)']
        voltage_pred = self.voltage_predictor.predict(X_voltage)
        voltage_r2 = r2_score(y_voltage, voltage_pred)
        print(f"  Voltage Predictor R²: {voltage_r2:.4f}")

        # Capacity prediction performance
        X_capacity = data[['Voltage(V)', 'Temperature_Numeric']]
        y_capacity = data['Charge_Capacity(Ah)']
        capacity_pred = self.capacity_predictor.predict(X_capacity)
        capacity_r2 = r2_score(y_capacity, capacity_pred)
        print(f"  Capacity Predictor R²: {capacity_r2:.4f}")

        # Classification accuracy
        X_class = data[['Voltage(V)', 'Charge_Capacity(Ah)', 'Discharge_Capacity(Ah)']]
        X_class_scaled = self.scaler.transform(X_class)
        y_class = data['Temp_Category']
        class_pred = self.temp_classifier.predict(X_class_scaled)
        class_accuracy = (class_pred == y_class).mean()
        print(f"  Temperature Classifier Accuracy: {class_accuracy:.4f}")

    def predict_voltage(self, temperature, capacity):
        """
        Predict battery voltage from temperature and capacity
        """
        if not self.is_trained:
            raise ValueError("Model must be trained first!")

        X = np.array([[temperature, capacity]])
        prediction = self.voltage_predictor.predict(X)[0]
        return prediction

    def predict_capacity(self, voltage, temperature):
        """
        Predict battery capacity from voltage and temperature
        """
        if not self.is_trained:
            raise ValueError("Model must be trained first!")

        X = np.array([[voltage, temperature]])
        prediction = self.capacity_predictor.predict(X)[0]
        return prediction

    def classify_temperature(self, voltage, charge_cap, discharge_cap):
        """
        Classify temperature range based on battery characteristics
        """
        if not self.is_trained:
```

```
            raise ValueError("Model must be trained first!")

        X = np.array([[voltage, charge_cap, discharge_cap]])
        X_scaled = self.scaler.transform(X)
        prediction = self.temp_classifier.predict(X_scaled)[0]
        probabilities = self.temp_classifier.predict_proba(X_scaled)[0]

        return prediction, dict(zip(self.temp_classifier.classes_, probabilities))

    def detect_anomaly(self, voltage, capacity, temperature):
        """
        Detect if battery measurements are anomalous
        """
        if not self.is_trained:
            raise ValueError("Model must be trained first!")

        X = np.array([[voltage, capacity, temperature]])
        X_scaled = StandardScaler().fit_transform(X)  # Note: In practice, save the scaler
        prediction = self.anomaly_detector.predict(X_scaled)[0]

        return "Normal" if prediction == 1 else "Anomalous"

    def comprehensive_analysis(self, voltage, temperature, charge_cap=None, discharge_cap=None)
        """
        Perform comprehensive battery analysis
        """
        print("=== Battery Analysis Report ===")
        print(f"Input measurements:")
        print(f"  Voltage: {voltage}V")
        print(f"  Temperature: {temperature}°C")
        if charge_cap: print(f"  Charge Capacity: {charge_cap}Ah")
        if discharge_cap: print(f"  Discharge Capacity: {discharge_cap}Ah")
        print()

        # Predict capacity if not provided
        if charge_cap is None:
            charge_cap = self.predict_capacity(voltage, temperature)
            print(f"Predicted Charge Capacity: {charge_cap:.3f}Ah")

        # Predict voltage if you change capacity
        alternative_cap = charge_cap * 0.8  # 80% capacity
        pred_voltage = self.predict_voltage(temperature, alternative_cap)
        print(f"Predicted voltage at {alternative_cap:.3f}Ah: {pred_voltage:.3f}V")

        # Classify temperature range
        if discharge_cap is None:
            discharge_cap = charge_cap * 0.95  # Assume 95% efficiency
```

```
    temp_category, temp_probs = self.classify_temperature(voltage, charge_cap, discharge_ca
    print(f"\nTemperature classification: {temp_category}")
    print("Classification probabilities:")
    for category, prob in temp_probs.items():
        print(f"  {category}: {prob:.2%}")

    # Anomaly detection
    anomaly_status = self.detect_anomaly(voltage, charge_cap, temperature)
    print(f"\nAnomaly status: {anomaly_status}")

    # Recommendations
    print("\n=== Recommendations ===")
    if temp_category == "Cold":
        print("- Battery performance may be reduced at low temperatures")
        print("- Consider pre-warming the battery for optimal performance")
    elif temp_category == "Hot":
        print("- High temperature may accelerate battery degradation")
        print("- Consider cooling the battery to extend lifespan")
    else:
        print("- Battery is operating in optimal temperature range")

    if anomaly_status == "Anomalous":
        print("- Unusual measurements detected - check sensor calibration")
        print("- Consider additional diagnostics")
```

Training and Testing Our Battery Analyzer

python

```python
# Create and train the analyzer
analyzer = BatteryAnalyzer()
analyzer.train(battery_data_clean)

# Test with various scenarios
print("\n" + "="*50)
print("TESTING BATTERY ANALYZER")
print("="*50)

# Test case 1: Normal operation at room temperature
print("\nTest Case 1: Normal operation")
analyzer.comprehensive_analysis(voltage=3.5, temperature=25)

print("\n" + "-"*50)

# Test case 2: Cold weather operation
print("\nTest Case 2: Cold weather")
analyzer.comprehensive_analysis(voltage=3.2, temperature=-5)

print("\n" + "-"*50)

# Test case 3: Hot weather operation
print("\nTest Case 3: Hot weather")
analyzer.comprehensive_analysis(voltage=3.7, temperature=45, charge_cap=1.0)

print("\n" + "-"*50)

# Test case 4: Unusual measurements
print("\nTest Case 4: Potential anomaly")
analyzer.comprehensive_analysis(voltage=4.5, temperature=25, charge_cap=0.5)
```

Visualizing the Complete System

python

```
# Create a comprehensive visualization dashboard
fig = plt.figure(figsize=(20, 15))

# Subplot 1: Voltage prediction performance
ax1 = plt.subplot(2, 3, 1)
temp_range = np.linspace(-10, 50, 20)
cap_range = np.linspace(0, 1.5, 20)
voltage_predictions = []

for temp in temp_range:
    for cap in cap_range:
        pred_voltage = analyzer.predict_voltage(temp, cap)
        voltage_predictions.append([temp, cap, pred_voltage])

voltage_df = pd.DataFrame(voltage_predictions, columns=['Temperature', 'Capacity', 'Voltage'])
pivot_voltage = voltage_df.pivot(index='Temperature', columns='Capacity', values='Voltage')
sns.heatmap(pivot_voltage, cmap='viridis', cbar_kws={'label': 'Predicted Voltage (V)'})
plt.title('Voltage Prediction Heatmap')
plt.xlabel('Capacity (Ah)')
plt.ylabel('Temperature (°C)')

# Subplot 2: Feature importance for voltage prediction
ax2 = plt.subplot(2, 3, 2)
feature_names = ['Temperature', 'Capacity']
importances = analyzer.voltage_predictor.feature_importances_
plt.bar(feature_names, importances)
plt.title('Feature Importance: Voltage Prediction')
plt.ylabel('Importance')

# Subplot 3: Temperature classification regions
ax3 = plt.subplot(2, 3, 3)
test_voltages = np.linspace(2.5, 4.0, 50)
test_capacities = np.linspace(0, 1.5, 50)
classification_grid = []

for v in test_voltages:
    for c in test_capacities:
        pred_class, _ = analyzer.classify_temperature(v, c, c*0.95)
        classification_grid.append([v, c, pred_class])

class_df = pd.DataFrame(classification_grid, columns=['Voltage', 'Capacity', 'Class'])
for class_name in class_df['Class'].unique():
    class_data = class_df[class_df['Class'] == class_name]
    plt.scatter(class_data['Voltage'], class_data['Capacity'],
                label=class_name, alpha=0.6)
```

```
plt.xlabel('Voltage (V)')
plt.ylabel('Capacity (Ah)')
plt.title('Temperature Classification Regions')
plt.legend()

# Subplot 4: PCA visualization
ax4 = plt.subplot(2, 3, 4)
# Apply PCA to sample data
sample_data = battery_data_clean.sample(1000, random_state=42)
pca_features = ['Voltage(V)', 'Charge_Capacity(Ah)', 'Temperature_Numeric']
X_pca_sample = sample_data[pca_features]
X_pca_scaled = StandardScaler().fit_transform(X_pca_sample)
X_pca_transformed = analyzer.pca.transform(X_pca_scaled)

# Color by temperature
scatter = plt.scatter(X_pca_transformed[:, 0], X_pca_transformed[:, 1],
                      c=sample_data['Temperature_Numeric'], cmap='coolwarm', alpha=0.6)
plt.colorbar(scatter, label='Temperature (°C)')
plt.xlabel('First Principal Component')
plt.ylabel('Second Principal Component')
plt.title('PCA: Battery Data Visualization')

# Subplot 5: Model performance comparison
ax5 = plt.subplot(2, 3, 5)
models_performance = {
    'Voltage Predictor': 0.95,  # These would be actual R² scores
    'Capacity Predictor': 0.88,
    'Temp Classifier': 0.92
}
plt.bar(models_performance.keys(), models_performance.values())
plt.ylabel('Performance Score')
plt.title('Model Performance Summary')
plt.xticks(rotation=45)

# Subplot 6: Real-time prediction example
ax6 = plt.subplot(2, 3, 6)
# Simulate battery discharge curve
discharge_times = np.linspace(0, 3600, 100)  # 1 hour discharge
current_capacity = 1.0
temperature = 25
predicted_voltages = []

for time in discharge_times:
    current_capacity *= 0.999  # Gradual discharge
    pred_voltage = analyzer.predict_voltage(temperature, current_capacity)
    predicted_voltages.append(pred_voltage)
```

```
plt.plot(discharge_times/60, predicted_voltages, 'b-', linewidth=2)
plt.xlabel('Time (minutes)')
plt.ylabel('Voltage (V)')
plt.title('Predicted Battery Discharge Curve')
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

Real-World Applications

python

```python
# Demonstrate practical applications

def battery_health_monitor():
    """
    Simulate a battery health monitoring system
    """
    print("=== Battery Health Monitoring System ===")

    # Simulate measurements over time
    measurements = [
        {"time": "Day 1", "voltage": 3.6, "temp": 22, "capacity": 1.0},
        {"time": "Day 30", "voltage": 3.55, "temp": 25, "capacity": 0.98},
        {"time": "Day 90", "voltage": 3.5, "temp": 20, "capacity": 0.94},
        {"time": "Day 180", "voltage": 3.4, "temp": 24, "capacity": 0.88},
    ]

    print("Tracking battery degradation over time:")
    for measurement in measurements:
        print(f"\n{measurement['time']}:")
        print(f"  Measured: {measurement['voltage']}V, {measurement['temp']}°C, {measurement['c

        # Check if measurements are normal
        anomaly = analyzer.detect_anomaly(
            measurement['voltage'],
            measurement['capacity'],
            measurement['temp']
        )
        print(f"  Status: {anomaly}")

        # Predict what voltage should be
        expected_voltage = analyzer.predict_voltage(measurement['temp'], measurement['capacity'
        voltage_diff = abs(measurement['voltage'] - expected_voltage)
        print(f"  Expected voltage: {expected_voltage:.3f}V (diff: {voltage_diff:.3f}V)")

def optimal_operating_conditions():
    """
    Find optimal operating conditions for different scenarios
    """
    print("\n=== Optimal Operating Conditions ===")

    scenarios = [
        {"name": "Maximum Voltage", "target": "voltage"},
        {"name": "Maximum Capacity", "target": "capacity"},
        {"name": "Balanced Performance", "target": "both"}
    ]
```

```
    for scenario in scenarios:
        print(f"\n{scenario['name']}:")

        best_voltage = 0
        best_capacity = 0
        best_temp = 0
        best_combined = 0

        # Test different temperatures
        for temp in range(-10, 51, 5):
            for cap in np.arange(0.5, 1.5, 0.1):
                pred_voltage = analyzer.predict_voltage(temp, cap)

                if scenario['target'] == 'voltage' and pred_voltage > best_voltage:
                    best_voltage = pred_voltage
                    best_temp = temp
                    best_capacity = cap
                elif scenario['target'] == 'capacity' and cap > best_capacity:
                    # Check if voltage is reasonable
                    if pred_voltage > 3.0:
                        best_voltage = pred_voltage
                        best_temp = temp
                        best_capacity = cap
                elif scenario['target'] == 'both':
                    combined_score = pred_voltage * cap
                    if combined_score > best_combined:
                        best_combined = combined_score
                        best_voltage = pred_voltage
                        best_temp = temp
                        best_capacity = cap

        print(f"  Optimal temperature: {best_temp}°C")
        print(f"  Optimal capacity: {best_capacity:.2f}Ah")
        print(f"  Resulting voltage: {best_voltage:.3f}V")

# Run the applications
battery_health_monitor()
optimal_operating_conditions()
```

Key Features of Our Battery Analyzer

1. **Multi-model system**: Combines regression, classification, and clustering
2. **Comprehensive analysis**: Provides predictions, classifications, and recommendations
3. **Anomaly detection**: Identifies unusual battery behavior
4. **Real-world applications**: Health monitoring and optimization

5. **Interpretable results**: Clear explanations and actionable insights

---

9. Step-by-Step Machine Learning Process {#ml-process}

Now that we've seen machine learning in action with battery data, let's break down the complete process you should follow for any machine learning project. This systematic approach will help you tackle new problems with confidence.

Step 1: Define the Problem

Before touching any data, clearly define what you're trying to achieve.

python

```python
# Example problem definitions for battery analysis:

problems = {
    "Voltage Prediction": {
        "type": "Regression",
        "goal": "Predict battery voltage from temperature and capacity",
        "success_metric": "R² score > 0.9",
        "business_value": "Optimize charging algorithms"
    },

    "Temperature Classification": {
        "type": "Classification",
        "goal": "Classify operating temperature range from battery characteristics",
        "success_metric": "Accuracy > 90%",
        "business_value": "Automated temperature monitoring"
    },

    "Anomaly Detection": {
        "type": "Unsupervised",
        "goal": "Identify unusual battery behavior",
        "success_metric": "Meaningful anomalies found",
        "business_value": "Predictive maintenance"
    }
}

# Function to evaluate problem feasibility
def evaluate_problem_feasibility(data, problem):
    """
    Assess if we have sufficient data for the problem
    """
    print(f"Evaluating: {problem['goal']}")

    # Check data size
    print(f"  Data size: {len(data)} samples")
    if len(data) < 1000:
        print("  ⚠️  Warning: Small dataset may limit model performance")

    # Check for missing values
    missing_ratio = data.isnull().sum().sum() / (len(data) * len(data.columns))
    print(f"  Missing data: {missing_ratio:.2%}")
    if missing_ratio > 0.1:
        print("  ⚠️  Warning: High missing data ratio")

    # Check target variable distribution (for classification)
    if problem['type'] == 'Classification':
        target_balance = data[target_column].value_counts(normalize=True)
```

```
        min_class_ratio = target_balance.min()
        print(f"  Class balance: {min_class_ratio:.2%} (smallest class)")
        if min_class_ratio < 0.1:
            print("  ⚠️  Warning: Imbalanced classes detected")

# Evaluate our problems
for name, problem in problems.items():
    print(f"\n{name}:")
    evaluate_problem_feasibility(battery_data_clean, problem)
```

Step 2: Explore and Understand Your Data

Data exploration is like getting to know a new friend - the better you understand your data, the better your models will be.

python

```python
class BatteryDataExplorer:
    """
    Systematic battery data exploration toolkit
    """

    def __init__(self, data):
        self.data = data

    def basic_info(self):
        """
        Get basic information about the dataset
        """
        print("=== Basic Dataset Information ===")
        print(f"Shape: {self.data.shape}")
        print(f"Columns: {self.data.columns.tolist()}")
        print(f"Memory usage: {self.data.memory_usage(deep=True).sum() / 1024**2:.1f} MB")
        print(f"\nData types:")
        print(self.data.dtypes)

    def missing_values_analysis(self):
        """
        Analyze missing values
        """
        print("\n=== Missing Values Analysis ===")
        missing = self.data.isnull().sum()
        missing_percent = (missing / len(self.data)) * 100

        missing_df = pd.DataFrame({
            'Column': missing.index,
            'Missing_Count': missing.values,
            'Missing_Percent': missing_percent.values
        })
        missing_df = missing_df[missing_df['Missing_Count'] > 0].sort_values('Missing_Percent',

        if len(missing_df) > 0:
            print(missing_df)
        else:
            print("No missing values found!")

    def numerical_summary(self):
        """
        Statistical summary of numerical columns
        """
        print("\n=== Numerical Variables Summary ===")
        numerical_cols = self.data.select_dtypes(include=[np.number]).columns
        print(self.data[numerical_cols].describe())
```

```
def categorical_summary(self):
    """
    Summary of categorical columns
    """
    print("\n=== Categorical Variables Summary ===")
    categorical_cols = self.data.select_dtypes(include=['object']).columns

    for col in categorical_cols:
        print(f"\n{col}:")
        print(self.data[col].value_counts())

def correlation_analysis(self):
    """
    Analyze correlations between numerical variables
    """
    print("\n=== Correlation Analysis ===")
    numerical_cols = self.data.select_dtypes(include=[np.number]).columns

    # Calculate correlation matrix
    corr_matrix = self.data[numerical_cols].corr()

    # Plot heatmap
    plt.figure(figsize=(12, 10))
    sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', center=0, fmt='.2f')
    plt.title('Correlation Matrix of Numerical Variables')
    plt.tight_layout()
    plt.show()

    # Find high correlations
    high_corr_pairs = []
    for i in range(len(corr_matrix.columns)):
        for j in range(i+1, len(corr_matrix.columns)):
            if abs(corr_matrix.iloc[i, j]) > 0.7:
                high_corr_pairs.append({
                    'var1': corr_matrix.columns[i],
                    'var2': corr_matrix.columns[j],
                    'correlation': corr_matrix.iloc[i, j]
                })

    if high_corr_pairs:
        print("High correlations (|r| > 0.7):")
        for pair in high_corr_pairs:
            print(f"  {pair['var1']} ↔ {pair['var2']}: {pair['correlation']:.3f}")

def distribution_plots(self):
    """
```

```
        Plot distributions of key variables
        """
        print("\n=== Distribution Analysis ===")
        key_vars = ['Voltage(V)', 'Charge_Capacity(Ah)', 'Temperature_Numeric']

        fig, axes = plt.subplots(2, 3, figsize=(18, 12))
        axes = axes.flatten()

        for i, var in enumerate(key_vars):
            if var in self.data.columns:
                # Histogram
                axes[i].hist(self.data[var], bins=50, alpha=0.7, edgecolor='black')
                axes[i].set_title(f'Distribution of {var}')
                axes[i].set_xlabel(var)
                axes[i].set_ylabel('Frequency')

                # Box plot
                axes[i+3].boxplot(self.data[var])
                axes[i+3].set_title(f'Box Plot of {var}')
                axes[i+3].set_ylabel(var)

        plt.tight_layout()
        plt.show()

    def outlier_detection(self):
        """
        Detect outliers using statistical methods
        """
        print("\n=== Outlier Detection ===")
        numerical_cols = self.data.select_dtypes(include=[np.number]).columns

        outlier_summary = {}

        for col in numerical_cols:
            Q1 = self.data[col].quantile(0.25)
            Q3 = self.data[col].quantile(0.75)
            IQR = Q3 - Q1

            lower_bound = Q1 - 1.5 * IQR
            upper_bound = Q3 + 1.5 * IQR

            outliers = self.data[(self.data[col] < lower_bound) | (self.data[col] > upper_bound
            outlier_percentage = (len(outliers) / len(self.data)) * 100

            outlier_summary[col] = {
                'count': len(outliers),
                'percentage': outlier_percentage,
```

```
                'lower_bound': lower_bound,
                'upper_bound': upper_bound
            }

        print("Outlier summary (using IQR method):")
        for col, info in outlier_summary.items():
            if info['count'] > 0:
                print(f"  {col}: {info['count']} outliers ({info['percentage']:.1f}%)")

    def full_exploration(self):
        """
        Run complete data exploration
        """
        self.basic_info()
        self.missing_values_analysis()
        self.numerical_summary()
        self.correlation_analysis()
        self.distribution_plots()
        self.outlier_detection()

# Run complete exploration
print("COMPREHENSIVE BATTERY DATA EXPLORATION")
print("="*50)
explorer = BatteryDataExplorer(battery_data_clean)
explorer.full_exploration()
```

Step 3: Data Preprocessing and Feature Engineering

Clean, transform, and enhance your data for better model performance.

python

```python
class BatteryDataPreprocessor:
    """
    Comprehensive data preprocessing for battery data
    """

    def __init__(self):
        self.scalers = {}
        self.encoders = {}
        self.feature_names = []

    def handle_missing_values(self, data, strategy='mean'):
        """
        Handle missing values in the dataset
        """
        print("Handling missing values...")
        data_processed = data.copy()

        numerical_cols = data_processed.select_dtypes(include=[np.number]).columns
        categorical_cols = data_processed.select_dtypes(include=['object']).columns

        # Handle numerical missing values
        for col in numerical_cols:
            if data_processed[col].isnull().any():
                if strategy == 'mean':
                    fill_value = data_processed[col].mean()
                elif strategy == 'median':
                    fill_value = data_processed[col].median()
                else:
                    fill_value = 0

                data_processed[col].fillna(fill_value, inplace=True)
                print(f"  Filled {col} with {strategy}: {fill_value:.3f}")

        # Handle categorical missing values
        for col in categorical_cols:
            if data_processed[col].isnull().any():
                fill_value = data_processed[col].mode()[0]
                data_processed[col].fillna(fill_value, inplace=True)
                print(f"  Filled {col} with mode: {fill_value}")

        return data_processed

    def remove_outliers(self, data, method='iqr', columns=None):
        """
        Remove outliers from specified columns
        """
```

```
        print("Removing outliers...")
        data_processed = data.copy()

        if columns is None:
            columns = data_processed.select_dtypes(include=[np.number]).columns

        initial_size = len(data_processed)

        for col in columns:
            if method == 'iqr':
                Q1 = data_processed[col].quantile(0.25)
                Q3 = data_processed[col].quantile(0.75)
                IQR = Q3 - Q1
                lower_bound = Q1 - 1.5 * IQR
                upper_bound = Q3 + 1.5 * IQR

                outliers_mask = (data_processed[col] >= lower_bound) & (data_processed[col] <=
                data_processed = data_processed[outliers_mask]

            elif method == 'zscore':
                from scipy import stats
                z_scores = np.abs(stats.zscore(data_processed[col]))
                data_processed = data_processed[z_scores < 3]

        final_size = len(data_processed)
        removed = initial_size - final_size
        print(f"  Removed {removed} outliers ({removed/initial_size:.1%})")

        return data_processed

    def create_features(self, data):
        """
        Create new features from existing ones
        """
        print("Creating new features...")
        data_processed = data.copy()

        # Temperature-based features
        if 'Temperature_Numeric' in data_processed.columns:
            # Temperature categories
            data_processed['Temp_Category'] = pd.cut(
                data_processed['Temperature_Numeric'],
                bins=[-np.inf, 0, 25, 40, np.inf],
                labels=['Cold', 'Cool', 'Warm', 'Hot']
            )

            # Temperature squared (for non-linear effects)
```

```
            data_processed['Temperature_Squared'] = data_processed['Temperature_Numeric'] ** 2

            print("  Created temperature category and squared features")

        # Capacity-based features
        if 'Charge_Capacity(Ah)' in data_processed.columns and 'Discharge_Capacity(Ah)' in data
            # Efficiency ratio
            data_processed['Charge_Efficiency'] = (
                data_processed['Discharge_Capacity(Ah)'] /
                (data_processed['Charge_Capacity(Ah)'] + 1e-6)  # Add small value to avoid divi
            )

            # Capacity difference
            data_processed['Capacity_Difference'] = (
                data_processed['Charge_Capacity(Ah)'] - data_processed['Discharge_Capacity(Ah)'
            )

            print("  Created efficiency and capacity difference features")

        # Voltage-based features
        if 'Voltage(V)' in data_processed.columns:
            # Voltage level categories
            data_processed['Voltage_Level'] = pd.cut(
                data_processed['Voltage(V)'],
                bins=[0, 3.0, 3.5, 4.0, 5.0],
                labels=['Low', 'Medium', 'High', 'Very_High']
            )

            print("  Created voltage level categories")

        # Time-based features
        if 'Test_Time(s)' in data_processed.columns:
            # Time in hours
            data_processed['Test_Time_Hours'] = data_processed['Test_Time(s)'] / 3600

            # Time intervals
            data_processed['Time_Interval'] = pd.cut(
                data_processed['Test_Time_Hours'],
                bins=[0, 1, 6, 12, 24, np.inf],
                labels=['First_Hour', 'Early', 'Mid', 'Late', 'Extended']
            )

            print("  Created time-based features")

        return data_processed

    def encode_categorical_features(self, data, columns=None):
```

```
    """
    Encode categorical features for machine learning
    """
    print("Encoding categorical features...")
    data_processed = data.copy()

    if columns is None:
        columns = data_processed.select_dtypes(include=['object', 'category']).columns

    from sklearn.preprocessing import LabelEncoder, OneHotEncoder

    for col in columns:
        if data_processed[col].dtype == 'object' or data_processed[col].dtype.name == 'cate
            # For binary categorical variables, use label encoding
            unique_values = data_processed[col].nunique()

            if unique_values == 2:
                le = LabelEncoder()
                data_processed[col + '_Encoded'] = le.fit_transform(data_processed[col])
                self.encoders[col] = le
                print(f"  Label encoded {col} (2 categories)")

            # For multi-category variables, use one-hot encoding
            elif unique_values <= 10:  # Avoid too many dummy variables
                dummies = pd.get_dummies(data_processed[col], prefix=col)
                data_processed = pd.concat([data_processed, dummies], axis=1)
                data_processed.drop(col, axis=1, inplace=True)
                print(f"  One-hot encoded {col} ({unique_values} categories)")

            else:
                print(f"  Skipped {col} (too many categories: {unique_values})")

    return data_processed

def scale_features(self, data, columns=None, method='standard'):
    """
    Scale numerical features
    """
    print("Scaling features...")
    data_processed = data.copy()

    if columns is None:
        columns = data_processed.select_dtypes(include=[np.number]).columns

    if method == 'standard':
        from sklearn.preprocessing import StandardScaler
        scaler = StandardScaler()
```

```
        elif method == 'minmax':
            from sklearn.preprocessing import MinMaxScaler
            scaler = MinMaxScaler()
        elif method == 'robust':
            from sklearn.preprocessing import RobustScaler
            scaler = RobustScaler()
        else:
            print(f"  Unknown scaling method: {method}")
            return data_processed

        # Scale specified columns
        scaled_data = scaler.fit_transform(data_processed[columns])

        # Create new column names
        scaled_columns = [f"{col}_scaled" for col in columns]

        # Add scaled columns to dataframe
        scaled_df = pd.DataFrame(scaled_data, columns=scaled_columns, index=data_processed.inde
        data_processed = pd.concat([data_processed, scaled_df], axis=1)

        # Store scaler for later use
        self.scalers[method] = scaler

        print(f"  Scaled {len(columns)} features using {method} scaling")
        return data_processed

    def full_preprocessing(self, data):
        """
        Run complete preprocessing pipeline
        """
        print("RUNNING FULL PREPROCESSING PIPELINE")
        print("="*40)

        # Step 1: Handle missing values
        data_processed = self.handle_missing_values(data)

        # Step 2: Remove outliers
        data_processed = self.remove_outliers(data_processed)

        # Step 3: Create new features
        data_processed = self.create_features(data_processed)

        # Step 4: Encode categorical features
        data_processed = self.encode_categorical_features(data_processed)

        # Step 5: Scale features
        numerical_cols = data_processed.select_dtypes(include=[np.number]).columns
```

```
        # Exclude already scaled columns and ID columns
        cols_to_scale = [col for col in numerical_cols
                         if not col.endswith('_scaled')
                         and col not in ['Data_Point', 'Cycle_Index', 'Step_Index']]

        data_processed = self.scale_features(data_processed, cols_to_scale)

        print(f"\nPreprocessing complete!")
        print(f"Original shape: {data.shape}")
        print(f"Processed shape: {data_processed.shape}")

        return data_processed

# Apply preprocessing
preprocessor = BatteryDataPreprocessor()
battery_data_processed = preprocessor.full_preprocessing(battery_data_clean)

# Show the results
print("\nSample of processed data:")
print(battery_data_processed.head())
```

Step 4: Model Selection and Training

Choose and train appropriate algorithms for your problem.

python

```python
class BatteryModelTrainer:
    """
    Train and compare multiple models for battery analysis
    """

    def __init__(self):
        self.models = {}
        self.results = {}

    def prepare_regression_models(self):
        """
        Set up regression models for voltage/capacity prediction
        """
        from sklearn.linear_model import LinearRegression, Ridge, Lasso
        from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
        from sklearn.svm import SVR

        self.regression_models = {
            'Linear Regression': LinearRegression(),
            'Ridge Regression': Ridge(alpha=1.0),
            'Lasso Regression': Lasso(alpha=1.0),
            'Random Forest': RandomForestRegressor(n_estimators=100, random_state=42),
            'Gradient Boosting': GradientBoostingRegressor(n_estimators=100, random_state=42),
            'Support Vector Machine': SVR(kernel='rbf')
        }

    def prepare_classification_models(self):
        """
        Set up classification models for temperature/category prediction
        """
        from sklearn.linear_model import LogisticRegression
        from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
        from sklearn.svm import SVC
        from sklearn.naive_bayes import GaussianNB

        self.classification_models = {
            'Logistic Regression': LogisticRegression(random_state=42),
            'Random Forest': RandomForestClassifier(n_estimators=100, random_state=42),
            'Gradient Boosting': GradientBoostingClassifier(n_estimators=100, random_state=42),
            'Support Vector Machine': SVC(kernel='rbf', random_state=42),
            'Naive Bayes': GaussianNB()
        }

    def train_regression_models(self, X_train, X_test, y_train, y_test, target_name):
        """
        Train and evaluate regression models
```

```
    """
    print(f"\nTraining regression models for {target_name}...")
    self.prepare_regression_models()

    regression_results = {}

    for name, model in self.regression_models.items():
        # Train the model
        model.fit(X_train, y_train)

        # Make predictions
        y_pred = model.predict(X_test)

        # Calculate metrics
        mse = mean_squared_error(y_test, y_pred)
        rmse = np.sqrt(mse)
        r2 = r2_score(y_test, y_pred)

        # Store results
        regression_results[name] = {
            'model': model,
            'mse': mse,
            'rmse': rmse,
            'r2': r2,
            'predictions': y_pred
        }

        print(f"  {name}: R² = {r2:.4f}, RMSE = {rmse:.4f}")

    self.results[f'regression_{target_name}'] = regression_results
    return regression_results

def train_classification_models(self, X_train, X_test, y_train, y_test, target_name):
    """
    Train and evaluate classification models
    """
    print(f"\nTraining classification models for {target_name}...")
    self.prepare_classification_models()

    from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

    classification_results = {}

    for name, model in self.classification_models.items():
        # Train the model
        model.fit(X_train, y_train)
```

```
            # Make predictions
            y_pred = model.predict(X_test)

            # Calculate metrics
            accuracy = accuracy_score(y_test, y_pred)
            precision = precision_score(y_test, y_pred, average='weighted')
            recall = recall_score(y_test, y_pred, average='weighted')
            f1 = f1_score(y_test, y_pred, average='weighted')

            # Store results
            classification_results[name] = {
                'model': model,
                'accuracy': accuracy,
                'precision': precision,
                'recall': recall,
                'f1': f1,
                'predictions': y_pred
            }

            print(f"  {name}: Accuracy = {accuracy:.4f}, F1 = {f1:.4f}")

        self.results[f'classification_{target_name}'] = classification_results
        return classification_results

    def hyperparameter_tuning(self, model, param_grid, X_train, y_train, cv=5):
        """
        Perform hyperparameter tuning with cross-validation
        """
        from sklearn.model_selection import GridSearchCV

        print("Performing hyperparameter tuning...")

        grid_search = GridSearchCV(
            estimator=model,
            param_grid=param_grid,
            cv=cv,
            scoring='r2' if hasattr(model, 'predict') and not hasattr(model, 'predict_proba') e
            n_jobs=-1,
            verbose=1
        )

        grid_search.fit(X_train, y_train)

        print(f"Best parameters: {grid_search.best_params_}")
        print(f"Best score: {grid_search.best_score_:.4f}")

        return grid_search.best_estimator_
```

```python
def visualize_results(self, task_type, target_name):
    """
    Visualize model comparison results
    """
    results_key = f'{task_type}_{target_name}'
    if results_key not in self.results:
        print("No results to visualize")
        return

    results = self.results[results_key]

    if task_type == 'regression':
        # Plot R² scores
        model_names = list(results.keys())
        r2_scores = [results[name]['r2'] for name in model_names]

        plt.figure(figsize=(12, 6))
        plt.subplot(1, 2, 1)
        bars = plt.bar(model_names, r2_scores)
        plt.title(f'R² Scores for {target_name} Prediction')
        plt.ylabel('R² Score')
        plt.xticks(rotation=45)

        # Add value labels on bars
        for bar, score in zip(bars, r2_scores):
            plt.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.01,
                    f'{score:.3f}', ha='center')

        # Plot actual vs predicted for best model
        best_model_name = max(results.keys(), key=lambda x: results[x]['r2'])
        best_predictions = results[best_model_name]['predictions']

        plt.subplot(1, 2, 2)
        plt.scatter(y_test, best_predictions, alpha=0.6)
        plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--', lw=2)
        plt.xlabel('Actual Values')
        plt.ylabel('Predicted Values')
        plt.title(f'Best Model: {best_model_name}')

    elif task_type == 'classification':
        # Plot accuracy scores
        model_names = list(results.keys())
        accuracy_scores = [results[name]['accuracy'] for name in model_names]

        plt.figure(figsize=(10, 6))
        bars = plt.bar(model_names, accuracy_scores)
```

```
            plt.title(f'Accuracy Scores for {target_name} Classification')
            plt.ylabel('Accuracy')
            plt.xticks(rotation=45)

            # Add value labels on bars
            for bar, score in zip(bars, accuracy_scores):
                plt.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.01,
                         f'{score:.3f}', ha='center')

        plt.tight_layout()
        plt.show()

# Train models on our battery data
trainer = BatteryModelTrainer()

# Example 1: Voltage prediction
print("VOLTAGE PREDICTION MODELS")
print("="*30)

# Prepare data for voltage prediction
voltage_features = ['Temperature_Numeric', 'Charge_Capacity(Ah)', 'Test_Time_Hours']
X_voltage = battery_data_processed[voltage_features].dropna()
y_voltage = battery_data_processed.loc[X_voltage.index, 'Voltage(V)']

# Split data
X_train_volt, X_test_volt, y_train_volt, y_test_volt = train_test_split(
    X_voltage, y_voltage, test_size=0.2, random_state=42
)

# Train voltage prediction models
voltage_results = trainer.train_regression_models(
    X_train_volt, X_test_volt, y_train_volt, y_test_volt, 'voltage'
)

# Visualize voltage prediction results
trainer.visualize_results('regression', 'voltage')

# Example 2: Temperature classification
print("\nTEMPERATURE CLASSIFICATION MODELS")
print("="*35)

# Prepare data for temperature classification
class_features = ['Voltage(V)', 'Charge_Capacity(Ah)', 'Discharge_Capacity(Ah)']
X_class = battery_data_processed[class_features].dropna()
y_class = battery_data_processed.loc[X_class.index, 'Temp_Category']

# Remove any NaN values in target
```

```
mask = y_class.notna()
X_class = X_class[mask]
y_class = y_class[mask]

# Split data
X_train_class, X_test_class, y_train_class, y_test_class = train_test_split(
    X_class, y_class, test_size=0.2, random_state=42, stratify=y_class
)

# Scale features for classification
scaler = StandardScaler()
X_train_class_scaled = scaler.fit_transform(X_train_class)
X_test_class_scaled = scaler.transform(X_test_class)

# Train classification models
class_results = trainer.train_classification_models(
    X_train_class_scaled, X_test_class_scaled, y_train_class, y_test_class, 'temperature'
)

# Visualize classification results
trainer.visualize_results('classification', 'temperature')

# Example 3: Hyperparameter tuning for best model
print("\nHYPERPARAMETER TUNING")
print("="*25)

# Find best voltage prediction model
best_voltage_model_name = max(voltage_results.keys(), key=lambda x: voltage_results[x]['r2'])
print(f"Best voltage model: {best_voltage_model_name}")

# Tune Random Forest if it's one of the top performers
if 'Random Forest' in voltage_results:
    param_grid = {
        'n_estimators': [50, 100, 200],
        'max_depth': [None, 10, 20],
        'min_samples_split': [2, 5, 10],
        'min_samples_leaf': [1, 2, 4]
    }

    # Perform tuning (using smaller param grid for demo)
    tuned_model = trainer.hyperparameter_tuning(
        RandomForestRegressor(random_state=42),
        param_grid,
        X_train_volt,
        y_train_volt,
        cv=3  # Use 3-fold CV for faster execution
    )
```

```
# Evaluate tuned model
tuned_predictions = tuned_model.predict(X_test_volt)
tuned_r2 = r2_score(y_test_volt, tuned_predictions)
original_r2 = voltage_results['Random Forest']['r2']

print(f"Original Random Forest R²: {original_r2:.4f}")
print(f"Tuned Random Forest R²: {tuned_r2:.4f}")
print(f"Improvement: {tuned_r2 - original_r2:.4f}")
```

Step 5: Model Evaluation and Validation

Thoroughly evaluate your models to ensure they perform well on unseen data.

python

```python
class BatteryModelEvaluator:
    """
    Comprehensive model evaluation toolkit
    """

    def __init__(self):
        self.evaluation_results = {}

    def cross_validation_analysis(self, model, X, y, cv=5, task_type='regression'):
        """
        Perform cross-validation analysis
        """
        from sklearn.model_selection import cross_val_score, cross_validate

        print(f"Performing {cv}-fold cross-validation...")

        if task_type == 'regression':
            scoring = ['r2', 'neg_mean_squared_error', 'neg_mean_absolute_error']
        else:
            scoring = ['accuracy', 'precision_weighted', 'recall_weighted', 'f1_weighted']

        cv_results = cross_validate(model, X, y, cv=cv, scoring=scoring, return_train_score=Tru

        # Print results
        print("Cross-validation results:")
        for metric in scoring:
            test_scores = cv_results[f'test_{metric}']
            train_scores = cv_results[f'train_{metric}']

            print(f"  {metric}:")
            print(f"    Test:  {test_scores.mean():.4f} ± {test_scores.std():.4f}")
            print(f"    Train: {train_scores.mean():.4f} ± {train_scores.std():.4f}")

            # Check for overfitting
            if train_scores.mean() - test_scores.mean() > 0.1:
                print(f"    ⚠️  Possible overfitting detected!")

        return cv_results

    def learning_curve_analysis(self, model, X, y, task_type='regression'):
        """
        Analyze learning curves to understand model behavior
        """
        from sklearn.model_selection import learning_curve

        print("Generating learning curves...")
```

```
    # Define training sizes
    train_sizes = np.linspace(0.1, 1.0, 10)

    # Generate learning curve
    if task_type == 'regression':
        scoring = 'r2'
    else:
        scoring = 'accuracy'

    train_sizes_abs, train_scores, test_scores = learning_curve(
        model, X, y, train_sizes=train_sizes, cv=5, scoring=scoring, n_jobs=-1
    )

    # Calculate mean and std
    train_mean = train_scores.mean(axis=1)
    train_std = train_scores.std(axis=1)
    test_mean = test_scores.mean(axis=1)
    test_std = test_scores.std(axis=1)

    # Plot learning curves
    plt.figure(figsize=(10, 6))
    plt.plot(train_sizes_abs, train_mean, 'o-', color='blue', label='Training score')
    plt.fill_between(train_sizes_abs, train_mean - train_std, train_mean + train_std, alpha

    plt.plot(train_sizes_abs, test_mean, 'o-', color='red', label='Validation score')
    plt.fill_between(train_sizes_abs, test_mean - test_std, test_mean + test_std, alpha=0.1

    plt.xlabel('Training Set Size')
    plt.ylabel(f'{scoring.capitalize()} Score')
    plt.title('Learning Curves')
    plt.legend(loc='best')
    plt.grid(True, alpha=0.3)
    plt.show()

    # Analyze the curves
    self._analyze_learning_curves(train_mean, test_mean, train_sizes_abs)

    return train_sizes_abs, train_scores, test_scores

def _analyze_learning_curves(self, train_scores, test_scores, train_sizes):
    """
    Analyze learning curve patterns
    """
    print("Learning curve analysis:")

    # Check final gap between training and validation
```

```
    final_gap = train_scores[-1] - test_scores[-1]

    if final_gap > 0.1:
        print("  ⚠️  Large gap between training and validation scores - possible overfitti
        print("    Recommendations: Reduce model complexity, add regularization, or gather
    elif final_gap < 0.02:
        print("  ✓ Good balance between training and validation scores")

    # Check if more data would help
    recent_improvement = test_scores[-1] - test_scores[-3]
    if recent_improvement > 0.01:
        print("  📈 Validation score still improving with more data")
        print("    Recommendation: Collect more training data if possible")
    else:
        print("  📊 Validation score has plateaued")
        print("    Recommendation: More data unlikely to help significantly")

    # Check for underfitting
    if test_scores[-1] < 0.7:  # Threshold depends on problem
        print("  ⚠️  Low overall performance - possible underfitting")
        print("    Recommendations: Increase model complexity, add features, or check data

def feature_importance_analysis(self, model, feature_names, model_type='tree_based'):
    """
    Analyze feature importance
    """
    print("Analyzing feature importance...")

    if model_type == 'tree_based' and hasattr(model, 'feature_importances_'):
        importances = model.feature_importances_

    elif model_type == 'linear' and hasattr(model, 'coef_'):
        importances = np.abs(model.coef_)
        if len(importances.shape) > 1:
            importances = importances.mean(axis=0)

    else:
        print("Model type not supported for feature importance analysis")
        return None

    # Create feature importance dataframe
    feature_importance_df = pd.DataFrame({
        'feature': feature_names,
        'importance': importances
    }).sort_values('importance', ascending=False)

    # Plot feature importance
```

```
        plt.figure(figsize=(10, 6))
        plt.barh(range(len(feature_importance_df)), feature_importance_df['importance'])
        plt.yticks(range(len(feature_importance_df)), feature_importance_df['feature'])
        plt.xlabel('Feature Importance')
        plt.title('Feature Importance Analysis')
        plt.gca().invert_yaxis()
        plt.tight_layout()
        plt.show()

        # Print top features
        print("Top 5 most important features:")
        for i, (_, row) in enumerate(feature_importance_df.head().iterrows()):
            print(f"  {i+1}. {row['feature']}: {row['importance']:.4f}")

        return feature_importance_df

    def residual_analysis(self, y_true, y_pred, title="Residual Analysis"):
        """
        Analyze residuals for regression models
        """
        residuals = y_true - y_pred

        fig, axes = plt.subplots(2, 2, figsize=(15, 12))

        # 1. Residuals vs Predicted
        axes[0, 0].scatter(y_pred, residuals, alpha=0.6)
        axes[0, 0].axhline(y=0, color='r', linestyle='--')
        axes[0, 0].set_xlabel('Predicted Values')
        axes[0, 0].set_ylabel('Residuals')
        axes[0, 0].set_title('Residuals vs Predicted')

        # 2. Residual histogram
        axes[0, 1].hist(residuals, bins=30, alpha=0.7, edgecolor='black')
        axes[0, 1].set_xlabel('Residuals')
        axes[0, 1].set_ylabel('Frequency')
        axes[0, 1].set_title('Residual Distribution')

        # 3. Q-Q plot
        from scipy import stats
        stats.probplot(residuals, dist="norm", plot=axes[1, 0])
        axes[1, 0].set_title('Q-Q Plot (Normal Distribution)')

        # 4. Actual vs Predicted
        axes[1, 1].scatter(y_true, y_pred, alpha=0.6)
        axes[1, 1].plot([y_true.min(), y_true.max()], [y_true.min(), y_true.max()], 'r--', lw=2
        axes[1, 1].set_xlabel('Actual Values')
        axes[1, 1].set_ylabel('Predicted Values')
```

```
        axes[1, 1].set_title('Actual vs Predicted')

        plt.suptitle(title)
        plt.tight_layout()
        plt.show()

        # Analyze residual patterns
        print("Residual analysis:")
        print(f"  Mean residual: {residuals.mean():.6f}")
        print(f"  Std residual: {residuals.std():.4f}")

        # Test for normality
        _, p_value = stats.shapiro(residuals[:5000] if len(residuals) > 5000 else residuals)
        print(f"  Normality test p-value: {p_value:.4f}")
        if p_value < 0.05:
            print("    ⚠️  Residuals may not be normally distributed")
        else:
            print("    ✓ Residuals appear normally distributed")

        # Check for heteroscedasticity
        correlation = np.corrcoef(y_pred, np.abs(residuals))[0, 1]
        print(f"  Correlation between predicted values and |residuals|: {correlation:.4f}")
        if abs(correlation) > 0.3:
            print("    ⚠️  Possible heteroscedasticity detected")
        else:
            print("    ✓ Residuals appear homoscedastic")

    def confusion_matrix_analysis(self, y_true, y_pred, class_names=None):
        """
        Detailed confusion matrix analysis for classification
        """
        from sklearn.metrics import confusion_matrix, classification_report

        # Generate confusion matrix
        cm = confusion_matrix(y_true, y_pred)

        # Plot confusion matrix
        plt.figure(figsize=(10, 8))
        sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                    xticklabels=class_names or sorted(np.unique(y_true)),
                    yticklabels=class_names or sorted(np.unique(y_true)))
        plt.xlabel('Predicted Label')
        plt.ylabel('True Label')
        plt.title('Confusion Matrix')
        plt.show()

        # Print detailed classification report
```

```
    print("Detailed Classification Report:")
    print(classification_report(y_true, y_pred, target_names=class_names))

    # Analyze common misclassifications
    print("\nMisclassification Analysis:")
    total_samples = len(y_true)

    for i, true_class in enumerate(sorted(np.unique(y_true))):
        for j, pred_class in enumerate(sorted(np.unique(y_pred))):
            if i != j:  # Only misclassifications
                misclass_count = cm[i, j]
                if misclass_count > 0:
                    percentage = (misclass_count / total_samples) * 100
                    print(f"  {true_class} → {pred_class}: {misclass_count} samples ({perce

def comprehensive_evaluation(self, model, X_train, X_test, y_train, y_test,
                             feature_names, task_type='regression', model_name="Model"):
    """
    Run comprehensive evaluation suite
    """
    print(f"COMPREHENSIVE EVALUATION FOR {model_name.upper()}")
    print("="*50)

    # 1. Basic performance metrics
    print("1. Basic Performance Metrics")
    print("-"*30)
    y_pred = model.predict(X_test)

    if task_type == 'regression':
        mse = mean_squared_error(y_test, y_pred)
        rmse = np.sqrt(mse)
        r2 = r2_score(y_test, y_pred)
        print(f"  R² Score: {r2:.4f}")
        print(f"  RMSE: {rmse:.4f}")
        print(f"  MSE: {mse:.4f}")

        # Additional regression metrics
        from sklearn.metrics import mean_absolute_error, explained_variance_score
        mae = mean_absolute_error(y_test, y_pred)
        evs = explained_variance_score(y_test, y_pred)
        print(f"  MAE: {mae:.4f}")
        print(f"  Explained Variance: {evs:.4f}")

    else:  # classification
        from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
        accuracy = accuracy_score(y_test, y_pred)
        precision = precision_score(y_test, y_pred, average='weighted')
```

```
            recall = recall_score(y_test, y_pred, average='weighted')
            f1 = f1_score(y_test, y_pred, average='weighted')

            print(f"  Accuracy: {accuracy:.4f}")
            print(f"  Precision: {precision:.4f}")
            print(f"  Recall: {recall:.4f}")
            print(f"  F1-Score: {f1:.4f}")

        # 2. Cross-validation analysis
        print(f"\n2. Cross-Validation Analysis")
        print("-"*30)
        self.cross_validation_analysis(model, np.vstack([X_train, X_test]),
                                       np.hstack([y_train, y_test]), task_type=task_type)

        # 3. Learning curves
        print(f"\n3. Learning Curves")
        print("-"*30)
        self.learning_curve_analysis(model, np.vstack([X_train, X_test]),
                                     np.hstack([y_train, y_test]), task_type=task_type)

        # 4. Feature importance (if available)
        print(f"\n4. Feature Importance Analysis")
        print("-"*30)
        if hasattr(model, 'feature_importances_') or hasattr(model, 'coef_'):
            model_type = 'tree_based' if hasattr(model, 'feature_importances_') else 'linear'
            self.feature_importance_analysis(model, feature_names, model_type)
        else:
            print("  Feature importance not available for this model type")

        # 5. Detailed error analysis
        print(f"\n5. Detailed Error Analysis")
        print("-"*30)
        if task_type == 'regression':
            self.residual_analysis(y_test, y_pred, f"Residual Analysis - {model_name}")
        else:
            self.confusion_matrix_analysis(y_test, y_pred)

# Run comprehensive evaluation on our best models
evaluator = BatteryModelEvaluator()

# Evaluate voltage prediction model (regression)
print("EVALUATING VOLTAGE PREDICTION MODEL")
best_voltage_model = voltage_results['Random Forest']['model']
evaluator.comprehensive_evaluation(
    best_voltage_model, X_train_volt, X_test_volt, y_train_volt, y_test_volt,
    voltage_features, task_type='regression', model_name='Random Forest (Voltage)'
)
```

```
print("\n" + "="*80 + "\n")

# Evaluate temperature classification model
print("EVALUATING TEMPERATURE CLASSIFICATION MODEL")
best_class_model = class_results['Random Forest']['model']
evaluator.comprehensive_evaluation(
    best_class_model, X_train_class_scaled, X_test_class_scaled, y_train_class, y_test_class,
    class_features, task_type='classification', model_name='Random Forest (Temperature)'
)
```

Step 6: Model Deployment and Monitoring

Learn how to deploy your models and monitor their performance in production.

python

```python
class BatteryModelDeployment:
    """
    Tools for deploying and monitoring battery prediction models
    """

    def __init__(self):
        self.model_versions = {}
        self.performance_history = []

    def save_model(self, model, model_name, version="1.0"):
        """
        Save trained model for deployment
        """
        import joblib
        import os
        from datetime import datetime

        # Create models directory if it doesn't exist
        os.makedirs('battery_models', exist_ok=True)

        # Generate filename with timestamp
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        filename = f'battery_models/{model_name}_v{version}_{timestamp}.pkl'

        # Save model
        joblib.dump(model, filename)

        # Save model metadata
        metadata = {
            'model_name': model_name,
            'version': version,
            'timestamp': timestamp,
            'filename': filename,
            'model_type': type(model).__name__
        }

        # Save metadata
        metadata_filename = filename.replace('.pkl', '_metadata.json')
        import json
        with open(metadata_filename, 'w') as f:
            json.dump(metadata, f, indent=2)

        print(f"Model saved: {filename}")
        print(f"Metadata saved: {metadata_filename}")

        self.model_versions[f"{model_name}_v{version}"] = metadata
```

```
    return filename

def load_model(self, filename):
    """
    Load a saved model
    """
    import joblib

    try:
        model = joblib.load(filename)
        print(f"Model loaded successfully from {filename}")
        return model
    except Exception as e:
        print(f"Error loading model: {e}")
        return None

def create_prediction_api(self, model, scaler=None):
    """
    Create a simple prediction API wrapper
    """
    class BatteryPredictionAPI:
        def __init__(self, model, scaler=None):
            self.model = model
            self.scaler = scaler
            self.prediction_count = 0

        def predict_voltage(self, temperature, capacity, test_time=0):
            """
            Predict battery voltage
            """
            # Prepare input
            features = np.array([[temperature, capacity, test_time]])

            # Scale if scaler is provided
            if self.scaler:
                features = self.scaler.transform(features)

            # Make prediction
            prediction = self.model.predict(features)[0]

            # Log prediction
            self.prediction_count += 1
            print(f"Prediction #{self.prediction_count}: Voltage = {prediction:.3f}V")

            return {
                'predicted_voltage': round(prediction, 3),
```

```
                'input_temperature': temperature,
                'input_capacity': capacity,
                'confidence': 'high' if hasattr(self.model, 'predict_proba') else 'unknown'
            }

        def batch_predict(self, input_data):
            """
            Make predictions on batch of data
            """
            features = np.array(input_data)

            if self.scaler:
                features = self.scaler.transform(features)

            predictions = self.model.predict(features)

            results = []
            for i, (inputs, pred) in enumerate(zip(input_data, predictions)):
                results.append({
                    'id': i,
                    'inputs': inputs,
                    'prediction': round(pred, 3)
                })

            return results

        def get_stats(self):
            """
            Get API usage statistics
            """
            return {
                'total_predictions': self.prediction_count,
                'model_type': type(self.model).__name__
            }

    return BatteryPredictionAPI(model, scaler)

def monitor_model_drift(self, reference_data, new_data, feature_names):
    """
    Monitor for data drift in production
    """
    print("Monitoring for data drift...")

    drift_detected = False
    drift_features = []

    for i, feature in enumerate(feature_names):
```

```
            # Extract feature values
            ref_values = reference_data[:, i]
            new_values = new_data[:, i]

            # Statistical test for drift (Kolmogorov-Smirnov test)
            from scipy.stats import ks_2samp
            statistic, p_value = ks_2samp(ref_values, new_values)

            # Check for significant drift
            if p_value < 0.05:
                drift_detected = True
                drift_features.append(feature)
                print(f"  ⚠️  Drift detected in {feature} (p-value: {p_value:.4f})")
            else:
                print(f"  ✓ No drift in {feature} (p-value: {p_value:.4f})")

        if drift_detected:
            print(f"\n🚨 Data drift detected in {len(drift_features)} features!")
            print("Recommendations:")
            print("  - Retrain the model with recent data")
            print("  - Investigate causes of drift")
            print("  - Consider adaptive learning strategies")
        else:
            print("\n✅ No significant data drift detected")

        return drift_detected, drift_features

    def performance_monitoring(self, y_true, y_pred, timestamp=None):
        """
        Monitor model performance over time
        """
        from datetime import datetime

        if timestamp is None:
            timestamp = datetime.now()

        # Calculate performance metrics
        r2 = r2_score(y_true, y_pred)
        mse = mean_squared_error(y_true, y_pred)

        # Store performance
        self.performance_history.append({
            'timestamp': timestamp,
            'r2_score': r2,
            'mse': mse,
            'n_samples': len(y_true)
        })
```

```
        print(f"Performance recorded at {timestamp}:")
        print(f"  R² Score: {r2:.4f}")
        print(f"  MSE: {mse:.4f}")

        # Check for performance degradation
        if len(self.performance_history) > 1:
            previous_r2 = self.performance_history[-2]['r2_score']
            r2_change = r2 - previous_r2

            if r2_change < -0.05:  # 5% degradation threshold
                print(f"  ⚠️  Performance degraded by {abs(r2_change):.3f}")
                print("  Consider retraining the model")
            elif r2_change > 0.01:
                print(f"  ✅ Performance improved by {r2_change:.3f}")

    def plot_performance_history(self):
        """
        Plot model performance over time
        """
        if len(self.performance_history) < 2:
            print("Insufficient performance history to plot")
            return

        timestamps = [entry['timestamp'] for entry in self.performance_history]
        r2_scores = [entry['r2_score'] for entry in self.performance_history]
        mse_scores = [entry['mse'] for entry in self.performance_history]

        fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 10))

        # R² score over time
        ax1.plot(timestamps, r2_scores, 'b-o')
        ax1.set_ylabel('R² Score')
        ax1.set_title('Model Performance Over Time')
        ax1.grid(True, alpha=0.3)

        # MSE over time
        ax2.plot(timestamps, mse_scores, 'r-o')
        ax2.set_xlabel('Time')
        ax2.set_ylabel('MSE')
        ax2.grid(True, alpha=0.3)

        plt.tight_layout()
        plt.show()

# Demonstrate model deployment
deployment = BatteryModelDeployment()
```

```python
# Save our best voltage prediction model
voltage_model_file = deployment.save_model(
    best_voltage_model,
    'voltage_predictor',
    version="1.0"
)

# Create prediction API
voltage_api = deployment.create_prediction_api(best_voltage_model)

# Test the API
print("\nTesting Battery Prediction API:")
print("-"*30)

# Single prediction
result1 = voltage_api.predict_voltage(temperature=25, capacity=1.0, test_time=1)
print(f"API Response: {result1}")

# Batch predictions
batch_data = [
    [25, 1.0, 1],    # 25°C, 1.0Ah, 1 hour
    [0, 0.8, 2],     # 0°C, 0.8Ah, 2 hours
    [40, 1.2, 0.5]   # 40°C, 1.2Ah, 0.5 hours
]

batch_results = voltage_api.batch_predict(batch_data)
print(f"\nBatch prediction results:")
for result in batch_results:
    print(f"  Input {result['inputs']} → Prediction: {result['prediction']}V")

# API statistics
stats = voltage_api.get_stats()
print(f"\nAPI Statistics: {stats}")

# Demonstrate drift monitoring
print(f"\nDrift Monitoring Example:")
print("-"*30)

# Create synthetic "new" data with slight drift
np.random.seed(42)
reference_data = X_test_volt.values
new_data = reference_data.copy()
new_data[:, 0] += np.random.normal(2, 1, len(new_data))  # Add drift to temperature

drift_detected, drift_features = deployment.monitor_model_drift(
    reference_data, new_data, voltage_features
```

```
)

# Simulate performance monitoring over time
print(f"\nPerformance Monitoring Simulation:")
print("-"*30)

# Simulate predictions at different time points
from datetime import datetime, timedelta

base_time = datetime.now()
for i in range(5):
    # Simulate some noise in performance
    noise = np.random.normal(0, 0.02)
    simulated_r2 = 0.95 + noise - (i * 0.01)  # Gradual degradation

    # Create synthetic true/predicted values with the target R²
    y_sim_true = np.random.normal(3.5, 0.3, 100)
    y_sim_pred = y_sim_true + np.random.normal(0, np.sqrt(1 - simulated_r2), 100)

    # Record performance
    deployment.performance_monitoring(
        y_sim_true, y_sim_pred,
        timestamp=base_time + timedelta(days=i*7)
    )

# Plot performance history
deployment.plot_performance_history()
```

---

10. Summary and Next Steps {#summary}

Congratulations! You've completed a comprehensive journey through machine learning with scikit-learn using real battery data. Let's summarize what you've learned and explore next steps.

What You've Accomplished

1. **Data Understanding and Preparation**

- ✅ Loaded and explored A123 battery datasets across multiple temperatures
- ✅ Cleaned data by handling missing values and outliers
- ✅ Created new features to enhance model performance
- ✅ Visualized relationships and patterns in battery behavior

2. **Machine Learning Fundamentals**

- ✅ Understood the features matrix (X) and target vector (y) concept

- ✅ Applied the consistent scikit-learn estimator API
- ✅ Split data into training and testing sets
- ✅ Scaled features for better model performance

3. **Supervised Learning**

- ✅ **Regression**: Predicted battery voltage from temperature and capacity
- ✅ **Classification**: Classified temperature ranges from battery characteristics
- ✅ Compared multiple algorithms (Linear, Random Forest, SVM, etc.)
- ✅ Evaluated model performance with appropriate metrics

4. **Unsupervised Learning**

- ✅ **Clustering**: Discovered natural groupings in battery behavior
- ✅ **Dimensionality Reduction**: Used PCA to understand main variation sources
- ✅ **Anomaly Detection**: Identified unusual battery measurements

5. **Advanced Applications**

- ✅ Built a comprehensive BatteryAnalyzer class
- ✅ Created real-time prediction systems
- ✅ Developed battery health monitoring capabilities

6. **Model Evaluation and Deployment**

- ✅ Performed cross-validation and learning curve analysis
- ✅ Analyzed feature importance and residuals
- ✅ Created deployment-ready prediction APIs
- ✅ Implemented drift monitoring and performance tracking

Key Insights from Battery Data

Temperature Effects are Crucial

- Cold temperatures (-10°C to 0°C) significantly reduce battery performance
- Room temperature (25°C) provides optimal charging behavior
- Hot temperatures (40°C+) can deliver more capacity but may degrade faster

Predictable Relationships

- Voltage correlates strongly with capacity and temperature
- Machine learning models achieve R^2 scores > 0.95 for voltage prediction

- Temperature classification from battery characteristics is 90%+ accurate

Hidden Patterns

- Clustering revealed distinct battery behavior groups by temperature
- PCA showed temperature and capacity as primary variation sources
- Anomaly detection helps identify sensor errors or unusual conditions

Real-World Applications

Your skills now enable you to:

1. **Battery Management Systems**: Predict remaining capacity and optimal charging
2. **Predictive Maintenance**: Identify batteries likely to fail soon
3. **Quality Control**: Detect abnormal battery behavior in manufacturing
4. **Research and Development**: Optimize battery chemistry and design
5. **Energy Management**: Improve efficiency in electric vehicles and grid storage

Next Steps: Continue Your ML Journey

Immediate Next Steps

1. **Practice with Other Datasets**
   - Try different battery types (lithium-ion variants, lead-acid, etc.)
   - Apply techniques to other time-series data (solar panels, fuel cells)
   - Explore public datasets on Kaggle or UCI ML Repository
2. **Enhance Your Models**
   - Implement time-series analysis for battery degradation over cycles
   - Try deep learning approaches (neural networks) for complex patterns
   - Explore ensemble methods combining multiple algorithms
3. **Advanced Techniques**
   - Learn about hyperparameter optimization (Grid Search, Bayesian Optimization)
   - Study feature selection methods to improve model interpretability
   - Explore automated machine learning (AutoML) tools

Intermediate Developments

1. **Time Series Forecasting**

python

```python
# Example: Predict battery degradation over time
from sklearn.preprocessing import TimeSeriesStandardScaler
# Implement ARIMA, Prophet, or LSTM for time-series prediction
```

2. **Advanced Feature Engineering**

python

```python
# Rolling statistics, lag features, change rates
data['voltage_rolling_mean'] = data['Voltage(V)'].rolling(10).mean()
data['capacity_change_rate'] = data['Capacity'].diff() / data['Time'].diff()
```

3. **Model Interpretability**

python

```python
# SHAP values for better model explanation
import shap
explainer = shap.TreeExplainer(model)
shap_values = explainer.shap_values(X_test)
```

Advanced Directions

1. **Deep Learning for Batteries**
   - Recurrent Neural Networks (RNNs) for sequential battery data
   - Convolutional Neural Networks (CNNs) for signal processing
   - Transformer models for long-term dependency modeling
2. **Specialized Battery Applications**
   - State-of-Health (SOH) estimation algorithms
   - Remaining Useful Life (RUL) prediction
   - Fast-charging optimization strategies
3. **Production Systems**
   - Real-time streaming data processing
   - MLOps pipelines for continuous model updates
   - Edge computing for embedded battery management

Recommended Learning Resources

Books

- "Hands-On Machine Learning" by Aurélien Géron
- "Pattern Recognition and Machine Learning" by Christopher Bishop

- "The Elements of Statistical Learning" by Hastie, Tibshirani, and Friedman

Online Courses

- Andrew Ng's Machine Learning Course (Coursera)
- Fast.ai Practical Deep Learning
- edX MIT Introduction to Machine Learning

Practice Platforms

- Kaggle competitions and datasets
- Google Colab for experimentation
- GitHub for version control and collaboration

Battery-Specific Resources

- Battery University (batteryuniversity.com)
- Journal of Power Sources
- IEEE conferences on battery technology

Final Thoughts

Machine learning with battery data is more than just an academic exercise—it's solving real problems that impact millions of devices and vehicles worldwide. The techniques you've learned here apply broadly to:

- **Energy Systems**: Solar panels, wind turbines, grid storage
- **Transportation**: Electric vehicles, aircraft, marine systems
- **Consumer Electronics**: Smartphones, laptops, wearables
- **Industrial Applications**: Backup power, medical devices, IoT sensors

Remember:

1. **Start simple**: Linear models often work surprisingly well
2. **Understand your data**: Good insights beat complex algorithms
3. **Validate rigorously**: Cross-validation prevents overly optimistic results
4. **Think practically**: Consider deployment and maintenance from the start
5. **Keep learning**: ML is a rapidly evolving field

The journey from data to deployed model is challenging but immensely rewarding. You now have the foundation to tackle complex real-world problems with confidence. Whether you're optimizing electric vehicle performance, extending smartphone battery life, or designing the next generation of energy storage systems, these skills will serve you well.

Happy modeling, and may your batteries always be optimally charged! 🔋⚡📊

Appendix: Quick Reference Guide

Essential Scikit-Learn Imports

python

```python
# Data processing
import pandas as pd
import numpy as np

# Scikit-learn core
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.metrics import mean_squared_error, r2_score, accuracy_score

# Models
from sklearn.linear_model import LinearRegression, LogisticRegression
from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA

# Visualization
import matplotlib.pyplot as plt
import seaborn as sns
```

Standard ML Workflow

python

```python
# 1. Load and explore data
data = pd.read_csv('battery_data.csv')
data.info()
data.describe()

# 2. Prepare features and target
X = data[['feature1', 'feature2', 'feature3']]
y = data['target']

# 3. Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# 4. Scale features (if needed)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# 5. Train model
model = RandomForestRegressor(n_estimators=100, random_state=42)
model.fit(X_train_scaled, y_train)

# 6. Evaluate
y_pred = model.predict(X_test_scaled)
r2 = r2_score(y_test, y_pred)
print(f"R² Score: {r2:.4f}")
```

Common Pitfalls to Avoid

1. **Data leakage**: Don't include future information in features
2. **Overfitting**: Always validate on unseen data
3. **Scaling mistakes**: Fit scaler only on training data
4. **Ignoring domain knowledge**: Physics and engineering insights are valuable
5. **Optimization without understanding**: Interpret your models

This guide has equipped you with both theoretical understanding and practical skills. Now go forth and create intelligent battery systems that power a more efficient future!