Appliances Energy Prediction: Nonlinear Regression with Ensemble Methods

Machine Learning – Final Project

Author: Mehdi Talebi
Dataset: Appliances Energy Prediction (UCI ML Repository)
Reference: Candanedo, L. M., Feldmann, A., & Degemmis, D. (2017). *Data driven prediction models of energy use of appliances in a low-energy house.* Energy and Buildings, 145, 13–25.

1. Problem Formulation & Data Understanding

Problem Statement

We aim to predict the **energy consumption of household appliances** (in Wh per 10-minute interval) using environmental sensor data from a low-energy house in Belgium. The dataset spans ~4.5 months of 10-minute recordings.

Why Nonlinear Regression?

Energy consumption depends on complex, nonlinear factors:

- **Occupancy patterns** create threshold effects (on/off appliance usage)
- **Temperature comfort zones** produce nonlinear heating/cooling demands
- **Time-of-day effects** show periodic, non-monotonic patterns
- **Weather interactions** (temperature × humidity) are inherently nonlinear

A simple linear model cannot capture these relationships adequately, motivating the use of nonlinear and ensemble regression methods.

```
# ── Imports ──────────────────────────────────────────────────────────
import warnings
warnings.filterwarnings('ignore')

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV, learning_curve
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.linear_model import LinearRegression, Ridge
from sklearn.tree import DecisionTreeRegressor
from sklearn.svm import SVR
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from sklearn.inspection import permutation_importance

# Reproducibility
RANDOM_STATE = 42
np.random.seed(RANDOM_STATE)

# Plot style
sns.set_theme(style='whitegrid', palette='deep', font_scale=1.1)
plt.rcParams['figure.figsize'] = (12, 6)
plt.rcParams['figure.dpi'] = 100

print("All imports successful.")
```

```
All imports successful.
```

```
from google.colab import drive
drive.mount('/content/drive') # Mount to a path without spaces
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

```
import os

target_directory = '/content/drive/MyDrive/Colab Notebooks/machine learning 25-26/Appliances Energy Prediction/'

os.chdir(target_directory)

print(f'current directory: {os.getcwd()}')
```

```
current directory: /content/drive/MyDrive/Colab Notebooks/machine learning 25-26/Appliances Energy Prediction
```

```
# ── Load Dataset ──────────────────────────────────────────────────────
df = pd.read_csv('energydata_complete.csv')

print(f"Dataset shape: {df.shape}")
print(f"\nData types:\n{df.dtypes}")
print(f"\nFirst 5 rows:")
df.head()
```

```
Dataset shape: (19735, 29)

Data types:
date            object
Appliances       int64
lights           int64
T1             float64
RH_1           float64
T2             float64
RH_2           float64
T3             float64
RH_3           float64
T4             float64
RH_4           float64
T5             float64
RH_5           float64
T6             float64
RH_6           float64
T7             float64
RH_7           float64
T8             float64
RH_8           float64
T9             float64
RH_9           float64
T_out          float64
Press_mm_hg    float64
RH_out         float64
Windspeed      float64
Visibility     float64
Tdewpoint      float64
rv1            float64
rv2            float64
dtype: object

First 5 rows:
```

	date	Appliances	lights	T1	RH_1	T2	RH_2	T3	RH_3	T4	...	T9	RH_9	T_out	Press
0	2016-01-11 17:00:00	60	30	19.89	47.596667	19.2	44.790000	19.79	44.730000	19.000000	...	17.033333	45.53	6.600000	
1	2016-01-11 17:10:00	60	30	19.89	46.693333	19.2	44.722500	19.79	44.790000	19.000000	...	17.066667	45.56	6.483333	
2	2016-01-11 17:20:00	50	30	19.89	46.300000	19.2	44.626667	19.79	44.933333	18.926667	...	17.000000	45.50	6.366667	
3	2016-01-11 17:30:00	50	40	19.89	46.066667	19.2	44.590000	19.79	45.000000	18.890000	...	17.000000	45.40	6.250000	

```
# ── Summary Statistics ────────────────────────────────────────────────
print("Summary Statistics:")
df.describe().round(2)
```

Summary Statistics:

	Appliances	lights	T1	RH_1	T2	RH_2	T3	RH_3	T4	RH_4	...	T9	RH_9
count	19735.00	19735.00	19735.00	19735.00	19735.00	19735.00	19735.00	19735.00	19735.00	19735.00	...	19735.00	19735.00
mean	97.69	3.80	21.69	40.26	20.34	40.42	22.27	39.24	20.86	39.03	...	19.49	41.55
std	102.52	7.94	1.61	3.98	2.19	4.07	2.01	3.25	2.04	4.34	...	2.01	4.15
min	10.00	0.00	16.79	27.02	16.10	20.46	17.20	28.77	15.10	27.66	...	14.89	29.17
25%	50.00	0.00	20.76	37.33	18.79	37.90	20.79	36.90	19.53	35.53	...	18.00	38.50
50%	60.00	0.00	21.60	39.66	20.00	40.50	22.10	38.53	20.67	38.40	...	19.39	40.90
75%	100.00	0.00	22.60	43.07	21.50	43.26	23.29	41.76	22.10	42.16	...	20.60	44.34
max	1080.00	70.00	26.26	63.36	29.86	56.03	29.24	50.16	26.20	51.09	...	24.50	53.33

8 rows × 28 columns

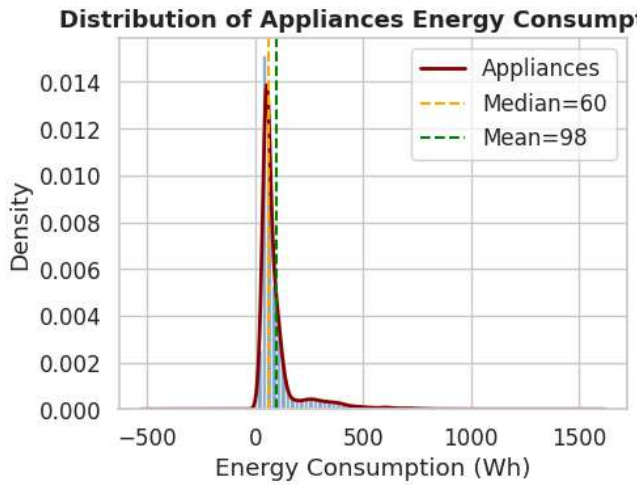
```
# ── Missing Values ──────────────────────────────────────────────────
missing = df.isnull().sum()
print(f"Total missing values: {missing.sum()}")
print(f"\nMissing values per column:")
print(missing[missing > 0] if missing.sum() > 0 else "No missing values found in any column.")
```

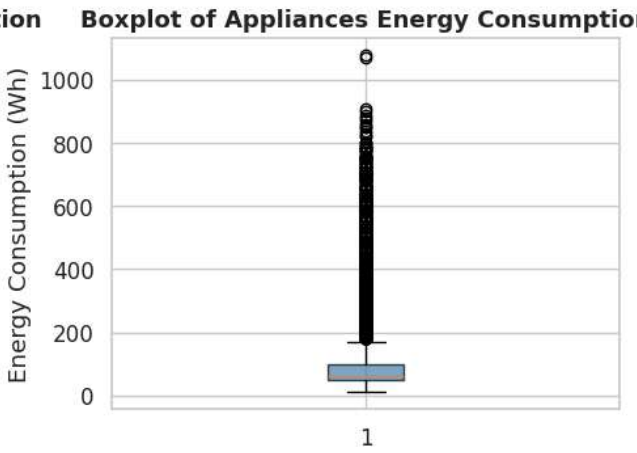
```
Total missing values: 0

Missing values per column:
No missing values found in any column.
```

2. Exploratory Data Analysis (EDA)

We explore the distribution of the target variable, relationships between predictors and the target, and temporal patterns in energy consumption.

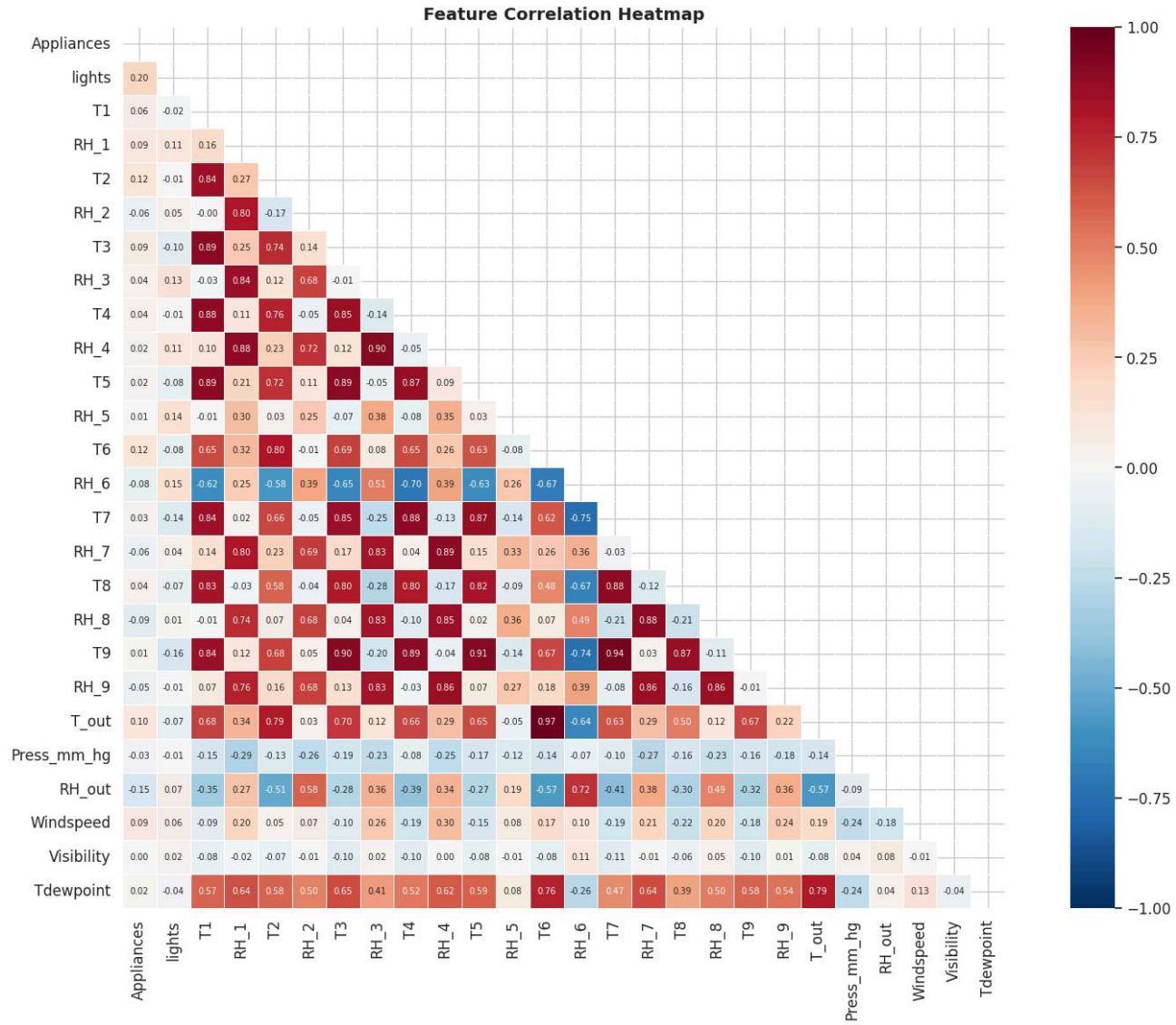
```
# ── Plot 1: Target Variable Distribution ────────────────────────────
fig, axes = plt.subplots(1, 2, figsize=(10, 4))

# Histogram + KDE
axes[0].hist(df['Appliances'], bins=50, color='steelblue', edgecolor='white', alpha=0.7, density=True)
df['Appliances'].plot.kde(ax=axes[0], color='darkred', linewidth=2)
axes[0].set_title('Distribution of Appliances Energy Consumption', fontsize=13, fontweight='bold')
axes[0].set_xlabel('Energy Consumption (Wh)')
axes[0].set_ylabel('Density')
axes[0].axvline(df['Appliances'].median(), color='orange', linestyle='--', label=f"Median={df['Appliances'].median():.(
axes[0].axvline(df['Appliances'].mean(), color='green', linestyle='--', label=f"Mean={df['Appliances'].mean():.0f}")
axes[0].legend()

# Boxplot
axes[1].boxplot(df['Appliances'], vert=True, patch_artist=True,
                boxprops=dict(facecolor='steelblue', alpha=0.7))
axes[1].set_title('Boxplot of Appliances Energy Consumption', fontsize=13, fontweight='bold')
axes[1].set_ylabel('Energy Consumption (Wh)')

plt.tight_layout()
plt.savefig('figures/01_target_distribution.png', dpi=150, bbox_inches='tight')
plt.show()

print(f"Skewness: {df['Appliances'].skew():.2f}")
print(f"Kurtosis: {df['Appliances'].kurtosis():.2f}")
print("\nThe target is right-skewed with a long tail, indicating many low-consumption periods and occasional high spik
```






```
Skewness: 3.39
Kurtosis: 13.67

The target is right-skewed with a long tail, indicating many low-consumption periods and occasional high spikes.
```

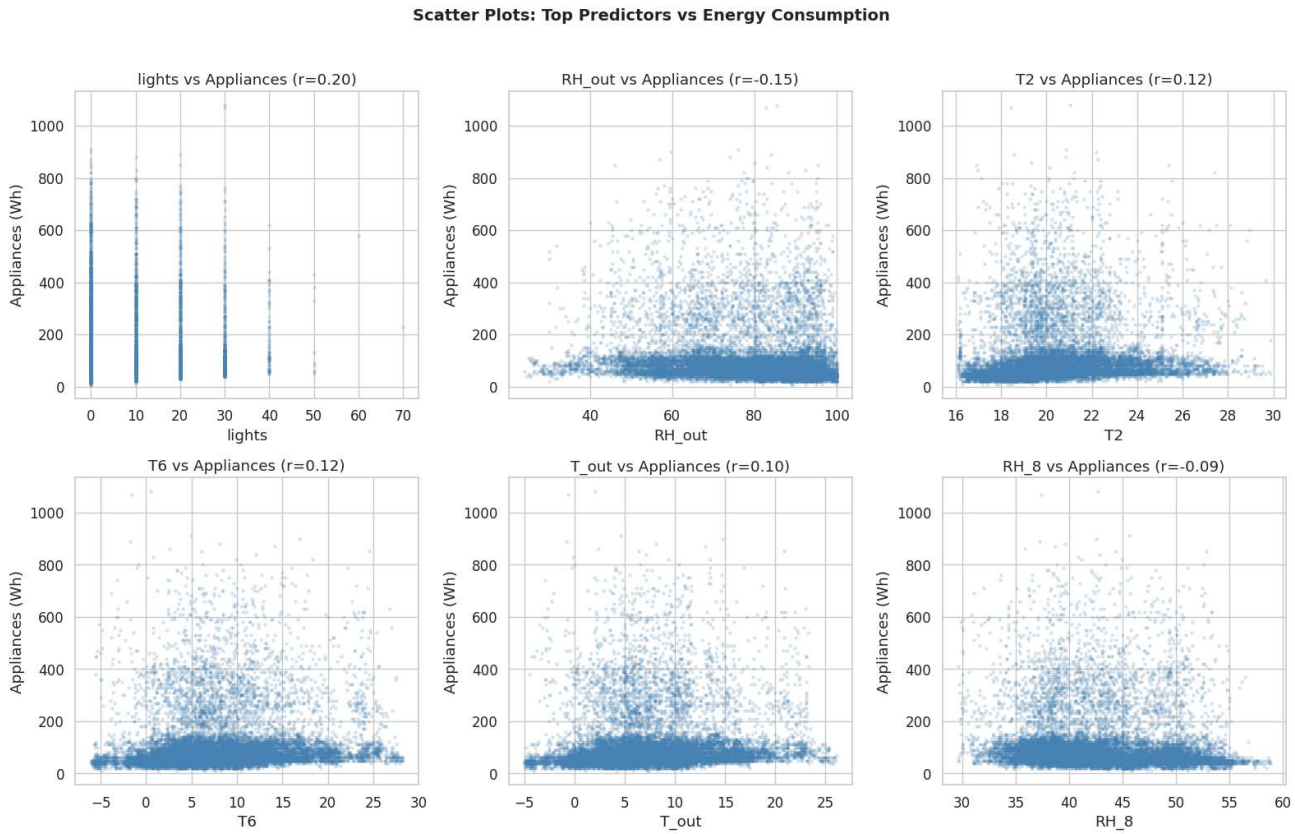
```
# ── Plot 2: Correlation Heatmap ──────────────────────────────────────────
# Exclude rv1, rv2 (random noise) and date
numeric_cols = df.select_dtypes(include=[np.number]).columns.tolist()
cols_to_drop = ['rv1', 'rv2']
numeric_cols = [c for c in numeric_cols if c not in cols_to_drop]

corr_matrix = df[numeric_cols].corr()

fig, ax = plt.subplots(figsize=(16, 12))
mask = np.triu(np.ones_like(corr_matrix, dtype=bool))
sns.heatmap(corr_matrix, mask=mask, annot=True, fmt='.2f', cmap='RdBu_r',
            center=0, square=True, linewidths=0.5, ax=ax,
            annot_kws={'size': 7}, vmin=-1, vmax=1)
ax.set_title('Feature Correlation Heatmap', fontsize=14, fontweight='bold')
plt.tight_layout()
plt.savefig('figures/02_correlation_heatmap.png', dpi=150, bbox_inches='tight')
plt.show()

# Top correlations with target
target_corr = corr_matrix['Appliances'].drop('Appliances').abs().sort_values(ascending=False)
print("Top 10 features correlated with Appliances (absolute):")
print(target_corr.head(10).round(3))
```




```
Top 10 features correlated with Appliances (absolute):
lights       0.197
RH_out       0.152
T2           0.120
T6           0.118
T_out        0.099
RH_8         0.094
Windspeed    0.087
RH_1         0.086
T3           0.085
RH_6         0.083
Name: Appliances, dtype: float64
```

```
# ── Plot 3: Scatter Plots of Key Predictors vs Target ──────────────────
top_features = target_corr.head(6).index.tolist()
```

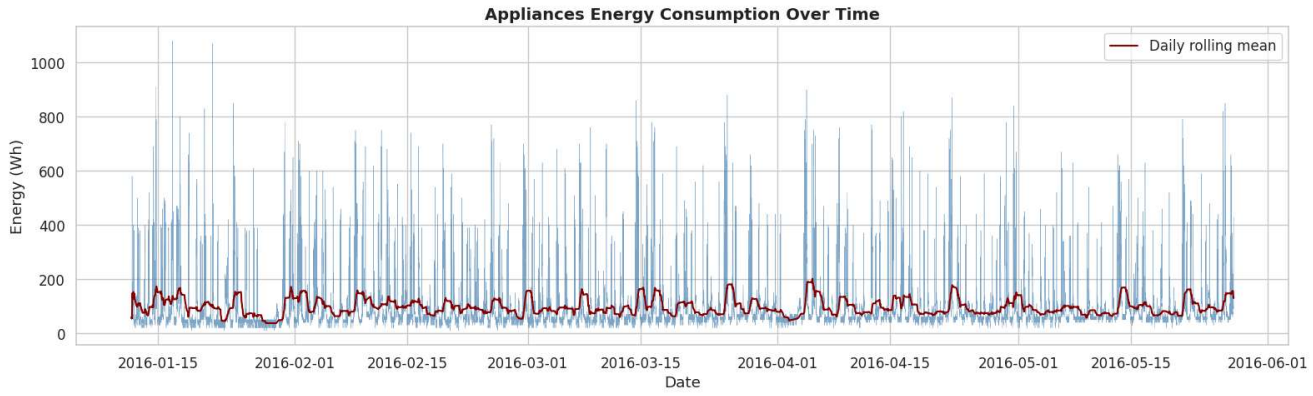
```
fig, axes = plt.subplots(2, 3, figsize=(16, 10))
for idx, feat in enumerate(top_features):
    ax = axes[idx // 3, idx % 3]
    ax.scatter(df[feat], df['Appliances'], alpha=0.15, s=5, color='steelblue')
    ax.set_xlabel(feat)
    ax.set_ylabel('Appliances (Wh)')
    ax.set_title(f'{feat} vs Appliances (r={corr_matrix.loc["Appliances", feat]:.2f})')

plt.suptitle('Scatter Plots: Top Predictors vs Energy Consumption', fontsize=14, fontweight='bold', y=1.02)
plt.tight_layout()
plt.savefig('figures/03_scatter_plots.png', dpi=150, bbox_inches='tight')
plt.show()
print("These scatter plots reveal non-trivial and generally nonlinear relationships between predictors and energy consu
```




```
These scatter plots reveal non-trivial and generally nonlinear relationships between predictors and energy consumption.
```

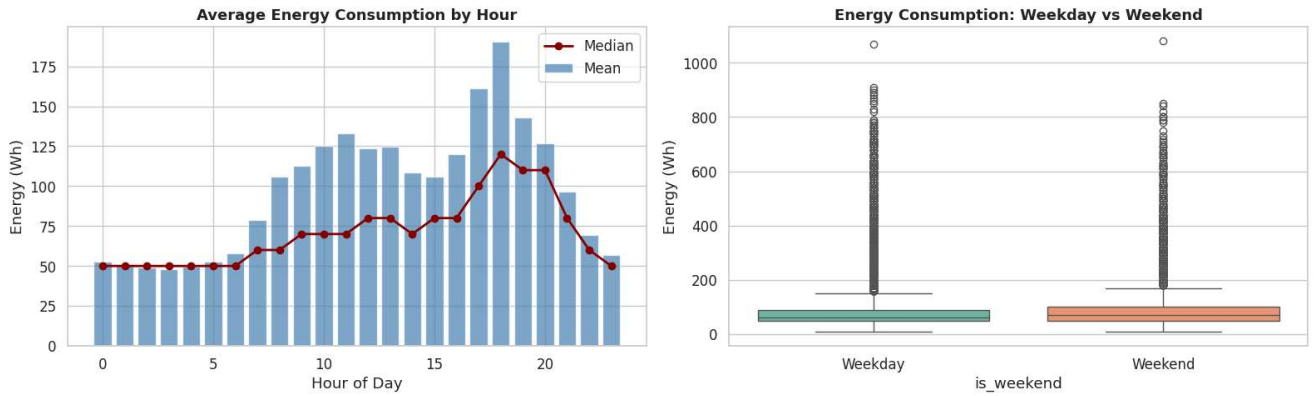
```
# ── Plot 4: Time-Series of Energy Consumption ──────────────────────────
df['date'] = pd.to_datetime(df['date'])

fig, ax = plt.subplots(figsize=(16, 5))
ax.plot(df['date'], df['Appliances'], linewidth=0.3, color='steelblue', alpha=0.7)
```

```
# Rolling mean
rolling = df.set_index('date')['Appliances'].rolling('1D').mean()
ax.plot(rolling.index, rolling.values, color='darkred', linewidth=1.5, label='Daily rolling mean')
ax.set_title('Appliances Energy Consumption Over Time', fontsize=14, fontweight='bold')
ax.set_xlabel('Date')
ax.set_ylabel('Energy (Wh)')
ax.legend()
plt.tight_layout()
plt.savefig('figures/04_time_series.png', dpi=150, bbox_inches='tight')
plt.show()
print("Clear temporal patterns are visible, with daily and weekly cycles indicating occupancy-driven consumption.")
```




```
Clear temporal patterns are visible, with daily and weekly cycles indicating occupancy-driven consumption.
```

```
# ── Plot 5: Energy by Hour of Day ──────────────────────────────────────
df['hour'] = df['date'].dt.hour
df['day_of_week'] = df['date'].dt.dayofweek
df['month'] = df['date'].dt.month
df['is_weekend'] = (df['day_of_week'] >= 5).astype(int)

fig, axes = plt.subplots(1, 2, figsize=(16, 5))

# By hour
hourly = df.groupby('hour')['Appliances'].agg(['mean', 'median'])
axes[0].bar(hourly.index, hourly['mean'], color='steelblue', alpha=0.7, label='Mean')
axes[0].plot(hourly.index, hourly['median'], color='darkred', marker='o', linewidth=2, label='Median')
axes[0].set_title('Average Energy Consumption by Hour', fontsize=13, fontweight='bold')
axes[0].set_xlabel('Hour of Day')
axes[0].set_ylabel('Energy (Wh)')
axes[0].legend()

# Weekday vs Weekend
sns.boxplot(x='is_weekend', y='Appliances', data=df, ax=axes[1], palette='Set2')
axes[1].set_xticklabels(['Weekday', 'Weekend'])
axes[1].set_title('Energy Consumption: Weekday vs Weekend', fontsize=13, fontweight='bold')
axes[1].set_ylabel('Energy (Wh)')

plt.tight_layout()
plt.savefig('figures/05_temporal_patterns.png', dpi=150, bbox_inches='tight')
plt.show()
print("Energy peaks during morning and evening hours (occupancy). Weekends show slightly different patterns.")
```




```
Energy peaks during morning and evening hours (occupancy). Weekends show slightly different patterns.
```

EDA Summary

- The target variable (`Appliances`) is **strongly right-skewed** (skewness ≈ 3.6) with most readings below 100 Wh and occasional spikes up to 1080 Wh.
- Correlations with individual features are **relatively weak** (max |r| < 0.3), suggesting nonlinear dependencies.
- Clear **temporal patterns** exist: higher consumption during morning/evening hours, differences between weekdays/weekends.
- Temperature and humidity features show **complex, nonlinear** relationships with energy usage, driven by occupancy and comfort-zone effects.

3. Data Preprocessing

Strategy

1. **Missing values**: None detected — no imputation needed.
2. **Feature engineering**: Extract temporal features from `date`, drop random noise columns (`rv1`, `rv2`).
3. **Outlier detection**: IQR-based method on the target variable.
4. **Feature scaling**: StandardScaler applied after train-test split (to prevent data leakage). Only required for linear models and SVR; tree-based models are scale-invariant.

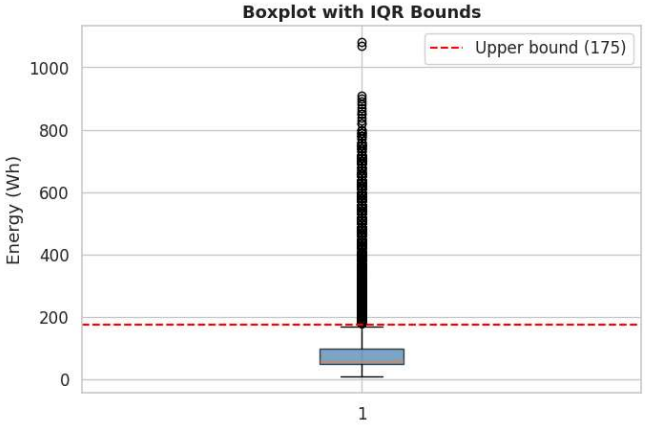
```
# ── Feature Engineering ──────────────────────────────────────────
# Time features already created: hour, day_of_week, month, is_weekend
# Drop columns not useful for modelling
df_processed = df.drop(columns=['date', 'rv1', 'rv2'])

print(f"Columns dropped: date, rv1, rv2")
print(f"Time features added: hour, day_of_week, month, is_weekend")
print(f"Processed dataset shape: {df_processed.shape}")
print(f"\nFeature list:")
for i, col in enumerate(df_processed.columns):
    print(f"  {i+1}. {col}")
```

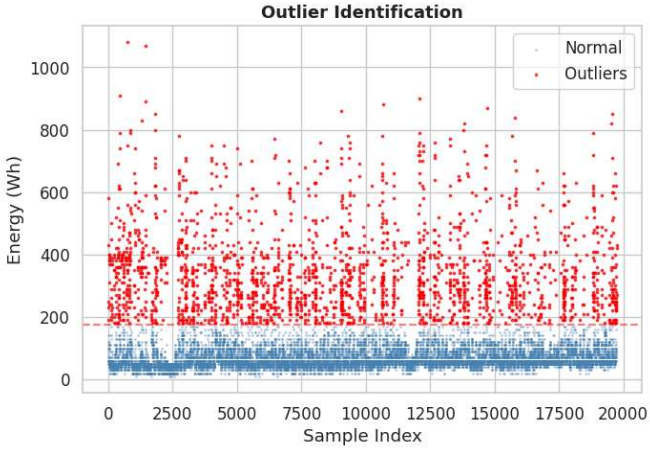
```
Columns dropped: date, rv1, rv2
Time features added: hour, day_of_week, month, is_weekend
Processed dataset shape: (19735, 30)

Feature list:
  1. Appliances
  2. lights
  3. T1
  4. RH_1
  5. T2
  6. RH_2
```

```
 7. T3
 8. RH_3
 9. T4
 10. RH_4
 11. T5
 12. RH_5
 13. T6
 14. RH_6
 15. T7
 16. RH_7
 17. T8
 18. RH_8
 19. T9
 20. RH_9
 21. T_out
 22. Press_mm_hg
 23. RH_out
 24. Windspeed
 25. Visibility
 26. Tdewpoint
 27. hour
 28. day_of_week
 29. month
 30. is_weekend
```

```
# ── Outlier Detection (IQR Method on Target) ──────────────────────────
Q1 = df_processed['Appliances'].quantile(0.25)
Q3 = df_processed['Appliances'].quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

outliers = df_processed[(df_processed['Appliances'] < lower_bound) |
                        (df_processed['Appliances'] > upper_bound)]
print(f"IQR: {IQR:.1f}")
print(f"Lower bound: {lower_bound:.1f}, Upper bound: {upper_bound:.1f}")
print(f"Number of outliers: {len(outliers)} ({len(outliers)/len(df_processed)*100:.1f}%)")

# Visualize outliers
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Boxplot with bounds
axes[0].boxplot(df_processed['Appliances'], vert=True, patch_artist=True,
                boxprops=dict(facecolor='steelblue', alpha=0.7))
axes[0].axhline(upper_bound, color='red', linestyle='--', label=f'Upper bound ({upper_bound:.0f})')
axes[0].set_title('Boxplot with IQR Bounds', fontsize=13, fontweight='bold')
axes[0].set_ylabel('Energy (Wh)')
axes[0].legend()

# Scatter of outliers
axes[1].scatter(range(len(df_processed)), df_processed['Appliances'], s=1, alpha=0.3, color='steelblue', label='Normal
axes[1].scatter(outliers.index, outliers['Appliances'], s=3, alpha=0.6, color='red', label='Outliers')
axes[1].axhline(upper_bound, color='red', linestyle='--', alpha=0.5)
axes[1].set_title('Outlier Identification', fontsize=13, fontweight='bold')
axes[1].set_xlabel('Sample Index')
axes[1].set_ylabel('Energy (Wh)')
axes[1].legend()

plt.tight_layout()
plt.savefig('figures/06_outlier_detection.png', dpi=150, bbox_inches='tight')
plt.show()
```

```
IQR: 50.0
Lower bound: -25.0, Upper bound: 175.0
Number of outliers: 2138 (10.8%)
```






```
# ── Decision: Cap Outliers ──────────────────────────────────────────
# We cap (winsorize) outliers rather than remove them because:
# 1. They represent real high-consumption events (valid data)
# 2. Removal would bias the model against predicting high usage
# 3. Capping limits extreme influence while preserving sample size

df_processed['Appliances'] = df_processed['Appliances'].clip(lower=lower_bound, upper=upper_bound)
print(f"Outliers capped to range [{lower_bound:.0f}, {upper_bound:.0f}]")
print(f"New target statistics:")
print(df_processed['Appliances'].describe().round(2))
```

```
Outliers capped to range [-25, 175]
New target statistics:
count    19735.00
mean        78.89
std         42.96
min         10.00
25%         50.00
50%         60.00
75%        100.00
max        175.00
Name: Appliances, dtype: float64
```

```
# ── Prepare Features and Target ─────────────────────────────────────
X = df_processed.drop(columns=['Appliances'])
y = df_processed['Appliances']

print(f"Features shape: {X.shape}")
print(f"Target shape: {y.shape}")
print(f"\nFeature names: {list(X.columns)}")
```

```
Features shape: (19735, 29)
Target shape: (19735,)

Feature names: ['lights', 'T1', 'RH_1', 'T2', 'RH_2', 'T3', 'RH_3', 'T4', 'RH_4', 'T5', 'RH_5', 'T6', 'RH_6', 'T7', 'RH_7', 'T8'
```

4. Train-Test Split & Cross-Validation

We use an **80/20 train-test split** with a fixed random seed for reproducibility. For hyperparameter tuning, we employ **5-fold cross-validation** within the training set.

Why this approach:

- 80/20 provides enough test data (~3,947 samples) for reliable evaluation

- 5-fold CV balances computational cost with variance estimation
- Fixed seed ensures reproducible results across runs

```
# ── Train-Test Split ──────────────────────────────────────────────
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=RANDOM_STATE
)

print(f"Training set: {X_train.shape[0]} samples ({X_train.shape[0]/len(X)*100:.0f}%)")
print(f"Test set:     {X_test.shape[0]} samples ({X_test.shape[0]/len(X)*100:.0f}%)")

# ── Feature Scaling ───────────────────────────────────────────────
# Fit on training set only to prevent data leakage
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Keep unscaled versions for tree-based models (they don't need scaling)
X_train_unscaled = X_train.values
X_test_unscaled = X_test.values

print("\nStandardScaler fitted on training data and applied to both sets.")
print("Note: Scaled features used for Linear/Polynomial/SVR; unscaled for tree-based models.")
```

```
Training set: 15788 samples (80%)
Test set:     3947 samples (20%)

StandardScaler fitted on training data and applied to both sets.
Note: Scaled features used for Linear/Polynomial/SVR; unscaled for tree-based models.
```

5. Model Development & Hyperparameter Tuning

We implement and compare **six regression models**:

#	Model	Type	Scaling Needed
1	Linear Regression	Baseline	Yes
2	Ridge Polynomial Regression	Nonlinear	Yes
3	Decision Tree Regressor	Nonlinear	No
4	SVR (RBF kernel)	Nonlinear	Yes
5	Random Forest Regressor	Ensemble	No
6	Gradient Boosting Regressor	Ensemble	No

Each model (except baseline) undergoes **GridSearchCV** with 5-fold CV and `neg_mean_squared_error` scoring.

```
# ── Helper: evaluate and store results ────────────────────────────
results = {}

def evaluate_model(name, model, X_tr, X_te, y_tr, y_te):
    """Train, predict, compute metrics, and store results."""
    y_train_pred = model.predict(X_tr)
    y_test_pred = model.predict(X_te)

    results[name] = {
        'model': model,
        'y_train_pred': y_train_pred,
        'y_test_pred': y_test_pred,
        'Train MAE': mean_absolute_error(y_tr, y_train_pred),
        'Test MAE': mean_absolute_error(y_te, y_test_pred),
        'Train RMSE': np.sqrt(mean_squared_error(y_tr, y_train_pred)),
        'Test RMSE': np.sqrt(mean_squared_error(y_te, y_test_pred)),
        'Train R²': r2_score(y_tr, y_train_pred),
        'Test R²': r2_score(y_te, y_test_pred),
    }

    print(f"\n{'='*60}")
    print(f"  {name}")
    print(f"{'='*60}")
    print(f"  Train MAE:  {results[name]['Train MAE']:.2f}")
    print(f"  Test  MAE:  {results[name]['Test MAE']:.2f}")
    print(f"  Train RMSE: {results[name]['Train RMSE']:.2f}")
```

```
    print(f"  Test  RMSE: {results[name]['Test RMSE']:.2f}")
    print(f"  Train R²:   {results[name]['Train R²']:.4f}")
    print(f"  Test  R²:   {results[name]['Test R²']:.4f}")

    return results[name]
```

```
# ── Model 1: Linear Regression (Baseline) ──────────────────────────────
print("Training Model 1: Linear Regression (Baseline)")
lr = LinearRegression()
lr.fit(X_train_scaled, y_train)
evaluate_model('Linear Regression', lr, X_train_scaled, X_test_scaled, y_train, y_test)
print("\nNo hyperparameters to tune (baseline model).")
```

```
Training Model 1: Linear Regression (Baseline)

============================================================
  Linear Regression
============================================================
  Train MAE:  26.41
  Test  MAE:  26.42
  Train RMSE: 35.86
  Test  RMSE: 35.58
  Train R²:   0.3038
  Test  R²:   0.3111

No hyperparameters to tune (baseline model).
```

```
# ── Model 2: Ridge Polynomial Regression ──────────────────────────────
print("Training Model 2: Polynomial Regression (Ridge)")
print("Testing polynomial degrees 2 and 3 with Ridge regularization...\n")

best_poly_score = -np.inf
best_poly_model = None
best_poly_degree = None
best_poly_alpha = None

for degree in [2, 3]:
    poly = PolynomialFeatures(degree=degree, include_bias=False, interaction_only=False)
    # Use a subset of top features for polynomial to avoid memory explosion
    top_feat_idx = [list(X.columns).index(f) for f in target_corr.head(8).index if f in X.columns]
    X_train_poly_sub = poly.fit_transform(X_train_scaled[:, top_feat_idx])
    X_test_poly_sub = poly.transform(X_test_scaled[:, top_feat_idx])

    for alpha in [0.1, 1.0, 10.0]:
        ridge = Ridge(alpha=alpha, random_state=RANDOM_STATE)
        scores = cross_val_score(ridge, X_train_poly_sub, y_train, cv=5,
                                 scoring='neg_mean_squared_error')
        mean_score = scores.mean()
        print(f"  Degree={degree}, Alpha={alpha}: CV MSE = {-mean_score:.2f}")

        if mean_score > best_poly_score:
            best_poly_score = mean_score
            best_poly_degree = degree
            best_poly_alpha = alpha

print(f"\nBest: Degree={best_poly_degree}, Alpha={best_poly_alpha}")

# Retrain with best params
poly_best = PolynomialFeatures(degree=best_poly_degree, include_bias=False)
top_feat_idx = [list(X.columns).index(f) for f in target_corr.head(8).index if f in X.columns]
X_train_poly = poly_best.fit_transform(X_train_scaled[:, top_feat_idx])
X_test_poly = poly_best.transform(X_test_scaled[:, top_feat_idx])

ridge_best = Ridge(alpha=best_poly_alpha, random_state=RANDOM_STATE)
ridge_best.fit(X_train_poly, y_train)
evaluate_model('Polynomial Ridge', ridge_best, X_train_poly, X_test_poly, y_train, y_test)
```

```
Training Model 2: Polynomial Regression (Ridge)
Testing polynomial degrees 2 and 3 with Ridge regularization...

  Degree=2, Alpha=0.1: CV MSE = 1374.80
  Degree=2, Alpha=1.0: CV MSE = 1374.80
  Degree=2, Alpha=10.0: CV MSE = 1375.95
  Degree=3, Alpha=0.1: CV MSE = 1284.19
  Degree=3, Alpha=1.0: CV MSE = 1284.04
```

```
  Degree=3, Alpha=10.0: CV MSE = 1289.51

Best: Degree=3, Alpha=1.0

============================================================
  Polynomial Ridge
============================================================
  Train MAE:  26.00
  Test  MAE:  26.62
  Train RMSE: 35.38
  Test  RMSE: 36.15
  Train R²:   0.3224
  Test  R²:   0.2889
{'model': Ridge(random_state=42),
 'y_train_pred': array([ 69.90332776,  99.64309918, 128.74186868, ...,  50.6835407 ,
        51.95648103, 106.60154022]),
 'y_test_pred': array([ 54.7351212 , 143.6214407 ,  56.1130524 , ...,  64.98400168,
        59.83959312, 103.37245959]),
 'Train MAE': 25.997210623757134,
 'Test MAE': 26.624298171946236,
 'Train RMSE': np.float64(35.38275396335135),
 'Test RMSE': np.float64(36.15179125486346),
 'Train R²': 0.3223520938904796,
 'Test R²': 0.28889398583366854}
```

```
# ── Model 3: Decision Tree Regressor ──────────────────────────────────
print("Training Model 3: Decision Tree Regressor")
print("Performing GridSearchCV...\n")

dt_params = {
    'max_depth': [5, 10, 15, 20],
    'min_samples_leaf': [5, 10, 20],
    'min_samples_split': [5, 10]
}

dt_grid = GridSearchCV(
    DecisionTreeRegressor(random_state=RANDOM_STATE),
    dt_params, cv=5, scoring='neg_mean_squared_error', n_jobs=-1
)
dt_grid.fit(X_train_unscaled, y_train)

print(f"Best params: {dt_grid.best_params_}")
print(f"Best CV RMSE: {np.sqrt(-dt_grid.best_score_):.2f}")
evaluate_model('Decision Tree', dt_grid.best_estimator_,
               X_train_unscaled, X_test_unscaled, y_train, y_test)
```

```
Training Model 3: Decision Tree Regressor
Performing GridSearchCV...

Best params: {'max_depth': 20, 'min_samples_leaf': 5, 'min_samples_split': 5}
Best CV RMSE: 29.71

============================================================
  Decision Tree
============================================================
  Train MAE:  9.53
  Test  MAE:  17.13
  Train RMSE: 15.62
  Test  RMSE: 27.66
  Train R²:   0.8679
  Test  R²:   0.5837
{'model': DecisionTreeRegressor(max_depth=20, min_samples_leaf=5, min_samples_split=5,
                      random_state=42),
 'y_train_pred': array([ 50.        , 100.        ,  81.42857143, ...,  36.        ,
        80.        , 140.        ]),
 'y_test_pred': array([45.        , 88.75      , 42.        , ..., 54.        ,
       85.        , 58.47222222]),
 'Train MAE': 9.527300462132285,
 'Test MAE': 17.13045146964919,
 'Train RMSE': np.float64(15.624957991681466),
 'Test RMSE': np.float64(27.659909083497435),
 'Train R²': 0.8678526214147765,
 'Test R²': 0.5837289514860361}
```

```
# ── Model 4: SVR (RBF Kernel) ─────────────────────────────────────────
print("Training Model 4: SVR (RBF Kernel)")
print("Performing GridSearchCV (this may take a few minutes)...\n")

# Use a subsample for SVR tuning (SVR is O(n²) to O(n³))
```

```
n_sub = min(5000, len(X_train_scaled))
idx_sub = np.random.choice(len(X_train_scaled), n_sub, replace=False)

svr_params = {
    'C': [1, 10, 100],
    'gamma': ['scale', 0.01, 0.1],
    'epsilon': [0.1, 0.5]
}

svr_grid = GridSearchCV(
    SVR(kernel='rbf'),
    svr_params, cv=3, scoring='neg_mean_squared_error', n_jobs=-1
)
svr_grid.fit(X_train_scaled[idx_sub], y_train.iloc[idx_sub])

print(f"Best params: {svr_grid.best_params_}")

# Retrain on full training set with best params
svr_best = SVR(**svr_grid.best_params_, kernel='rbf')
svr_best.fit(X_train_scaled, y_train)
evaluate_model('SVR (RBF)', svr_best, X_train_scaled, X_test_scaled, y_train, y_test)
```

```
Training Model 4: SVR (RBF Kernel)
Performing GridSearchCV (this may take a few minutes)...

Best params: {'C': 100, 'epsilon': 0.5, 'gamma': 0.1}

============================================================
  SVR (RBF)
============================================================
  Train MAE:  12.29
  Test  MAE:  16.16
  Train RMSE: 22.38
  Test  RMSE: 26.44
  Train R²:   0.7289
  Test  R²:   0.6196
{'model': SVR(C=100, epsilon=0.5, gamma=0.1),
 'y_train_pred': array([ 42.10321841, 114.0460955 ,  86.45747262, ...,  47.79336254,
        79.50004737, 151.23739803]),
 'y_test_pred': array([49.02529585, 93.41641072, 45.95081615, ..., 54.87773157,
       58.55912652, 59.97520652]),
 'Train MAE': 12.291212436568484,
 'Test MAE': 16.159761078669813,
 'Train RMSE': np.float64(22.380661992929657),
 'Test RMSE': np.float64(26.440816544966005),
 'Train R²': 0.728876793459531,
 'Test R²': 0.6196140773392268}
```

```
# ── Model 5: Random Forest Regressor ──────────────────────────────────────
print("Training Model 5: Random Forest Regressor")
print("Performing GridSearchCV...\n")

rf_params = {
    'n_estimators': [100, 200],
    'max_depth': [10, 20, None],
    'min_samples_leaf': [2, 5],
}

rf_grid = GridSearchCV(
    RandomForestRegressor(random_state=RANDOM_STATE),
    rf_params, cv=5, scoring='neg_mean_squared_error', n_jobs=-1
)
rf_grid.fit(X_train_unscaled, y_train)

print(f"Best params: {rf_grid.best_params_}")
print(f"Best CV RMSE: {np.sqrt(-rf_grid.best_score_):.2f}")
evaluate_model('Random Forest', rf_grid.best_estimator_,
               X_train_unscaled, X_test_unscaled, y_train, y_test)
```

```
Training Model 5: Random Forest Regressor
Performing GridSearchCV...

Best params: {'max_depth': None, 'min_samples_leaf': 2, 'n_estimators': 200}
Best CV RMSE: 24.04

============================================================
  Random Forest
```

```
============================================================
  Train MAE:  7.02
  Test  MAE:  14.66
  Train RMSE: 10.89
  Test  RMSE: 22.11
  Train R²:   0.9359
  Test  R²:   0.7341
{'model': RandomForestRegressor(min_samples_leaf=2, n_estimators=200, random_state=42),
 'y_train_pred': array([ 46.21761905, 104.62595238,  79.02684524, ...,  37.56043651,
        82.02916667, 149.88825397]),
 'y_test_pred': array([52.11854978, 99.28875   , 45.15559524, ..., 55.99924603,
       77.0477381 , 59.30833333]),
 'Train MAE': 7.023528576621946,
 'Test MAE': 14.664539932311026,
 'Train RMSE': np.float64(10.885540665731122),
 'Test RMSE': np.float64(22.10731960995138),
 'Train R²': 0.9358611978616793,
 'Test R²': 0.7340825176483212}
```

```
# ── Model 6: Gradient Boosting Regressor ──────────────────────────────
print("Training Model 6: Gradient Boosting Regressor")
print("Performing GridSearchCV...\n")

gb_params = {
    'n_estimators': [100, 200],
    'max_depth': [3, 5, 7],
    'learning_rate': [0.05, 0.1],
}

gb_grid = GridSearchCV(
    GradientBoostingRegressor(random_state=RANDOM_STATE),
    gb_params, cv=5, scoring='neg_mean_squared_error', n_jobs=-1
)
gb_grid.fit(X_train_unscaled, y_train)

print(f"Best params: {gb_grid.best_params_}")
print(f"Best CV RMSE: {np.sqrt(-gb_grid.best_score_):.2f}")
evaluate_model('Gradient Boosting', gb_grid.best_estimator_,
               X_train_unscaled, X_test_unscaled, y_train, y_test)
```

```
Training Model 6: Gradient Boosting Regressor
Performing GridSearchCV...

Best params: {'learning_rate': 0.1, 'max_depth': 7, 'n_estimators': 200}
Best CV RMSE: 24.35

============================================================
  Gradient Boosting
============================================================
  Train MAE:  10.70
  Test  MAE:  15.71
  Train RMSE: 14.68
  Test  RMSE: 23.16
  Train R²:   0.8834
  Test  R²:   0.7081
{'model': GradientBoostingRegressor(max_depth=7, n_estimators=200, random_state=42),
 'y_train_pred': array([ 42.36841161, 110.84772351,  86.88006171, ...,  45.42571671,
        83.46584706, 134.91278418]),
 'y_test_pred': array([48.34444677, 97.90223814, 45.2361487 , ..., 56.29444736,
       63.27067087, 65.09114429]),
 'Train MAE': 10.704593944291727,
 'Test MAE': 15.708589614357193,
 'Train RMSE': np.float64(14.680152535966494),
 'Test RMSE': np.float64(23.160332949942145),
 'Train R²': 0.883350744308195,
 'Test R²': 0.7081469022518635}
```

6. Model Evaluation & Comparison

```
# ── Comparison Table ──────────────────────────────────────────────────
comparison_data = []
for name, res in results.items():
    comparison_data.append({
        'Model': name,
        'Train MAE': round(res['Train MAE'], 2),
        'Test MAE': round(res['Test MAE'], 2),
```

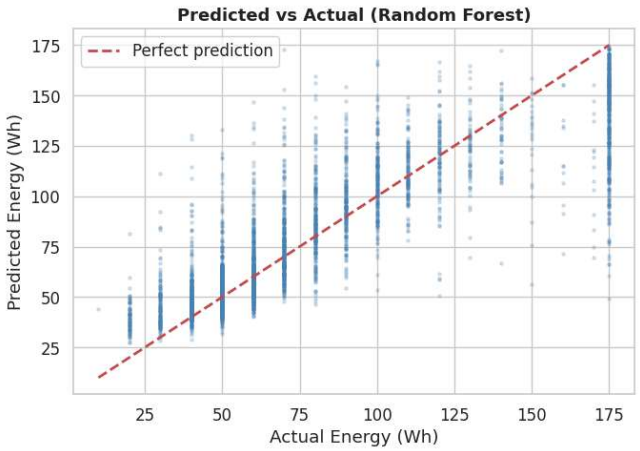
```
        'Train RMSE': round(res['Train RMSE'], 2),
        'Test RMSE': round(res['Test RMSE'], 2),
        'Train R²': round(res['Train R²'], 4),
        'Test R²': round(res['Test R²'], 4),
    })

comparison_df = pd.DataFrame(comparison_data)
comparison_df = comparison_df.sort_values('Test R²', ascending=False).reset_index(drop=True)
print("\n" + "="*80)
print("  MODEL COMPARISON TABLE (sorted by Test R²)")
print("="*80)
comparison_df
```

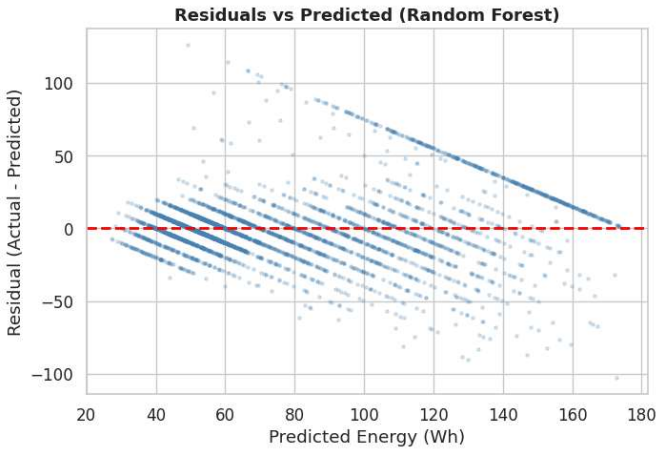
```
================================================================================
  MODEL COMPARISON TABLE (sorted by Test R²)
================================================================================
```

	Model	Train MAE	Test MAE	Train RMSE	Test RMSE	Train R²	Test R²
0	Random Forest	7.02	14.66	10.89	22.11	0.9359	0.7341
1	Gradient Boosting	10.70	15.71	14.68	23.16	0.8834	0.7081
2	SVR (RBF)	12.29	16.16	22.38	26.44	0.7289	0.6196
3	Decision Tree	9.53	17.13	15.62	27.66	0.8679	0.5837
4	Linear Regression	26.41	26.42	35.86	35.58	0.3038	0.3111
5	Polynomial Ridge	26.00	26.62	35.38	36.15	0.3224	0.2889

Next steps: Generate code with `comparison_df` New interactive sheet

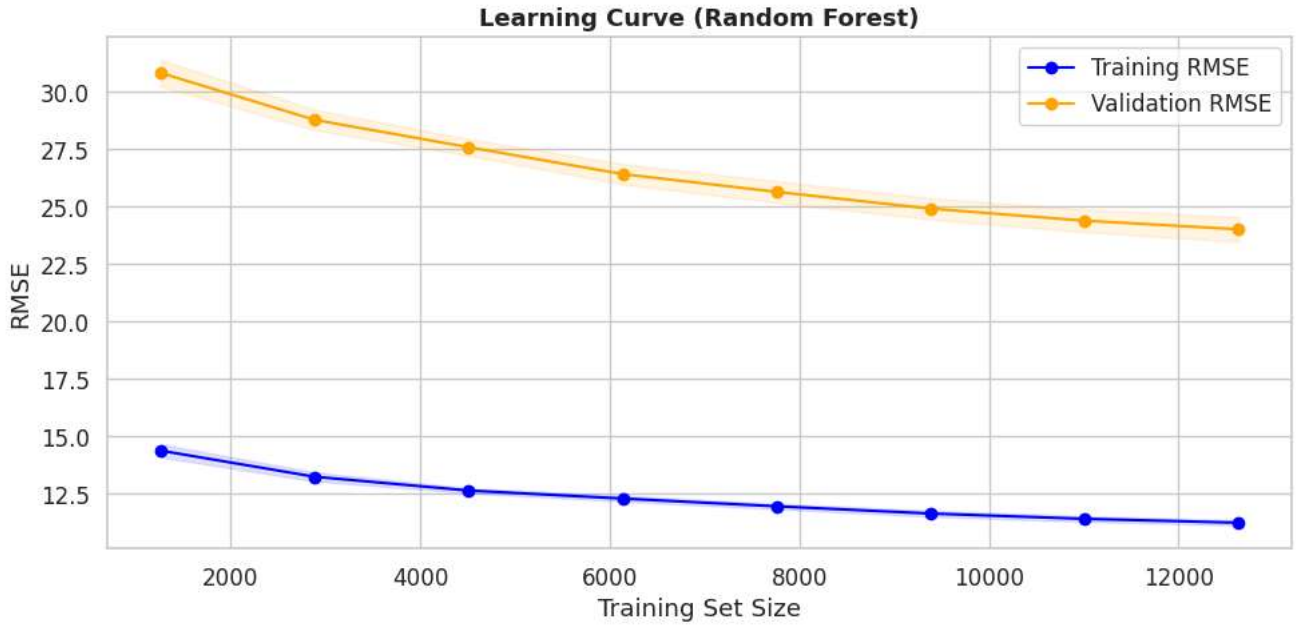
```
# ── Residual Plots for Best Model ──────────────────────────────────────
best_model_name = comparison_df.iloc[0]['Model']
best_res = results[best_model_name]
y_pred_test = best_res['y_test_pred']
residuals = y_test.values - y_pred_test

fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Predicted vs Actual
axes[0].scatter(y_test, y_pred_test, alpha=0.2, s=5, color='steelblue')
lims = [min(y_test.min(), y_pred_test.min()), max(y_test.max(), y_pred_test.max())]
axes[0].plot(lims, lims, 'r--', linewidth=2, label='Perfect prediction')
axes[0].set_xlabel('Actual Energy (Wh)')
axes[0].set_ylabel('Predicted Energy (Wh)')
axes[0].set_title(f'Predicted vs Actual ({best_model_name})', fontsize=13, fontweight='bold')
axes[0].legend()

# Residuals vs Predicted
axes[1].scatter(y_pred_test, residuals, alpha=0.2, s=5, color='steelblue')
axes[1].axhline(0, color='red', linestyle='--', linewidth=2)
axes[1].set_xlabel('Predicted Energy (Wh)')
axes[1].set_ylabel('Residual (Actual - Predicted)')
axes[1].set_title(f'Residuals vs Predicted ({best_model_name})', fontsize=13, fontweight='bold')

plt.tight_layout()
plt.savefig('figures/07_residual_plots.png', dpi=150, bbox_inches='tight')
plt.show()

print(f"\nResidual statistics for {best_model_name}:")
print(f"  Mean: {residuals.mean():.2f}")
print(f"  Std:  {residuals.std():.2f}")
print(f"  Min:  {residuals.min():.2f}")
print(f"  Max:  {residuals.max():.2f}")
```





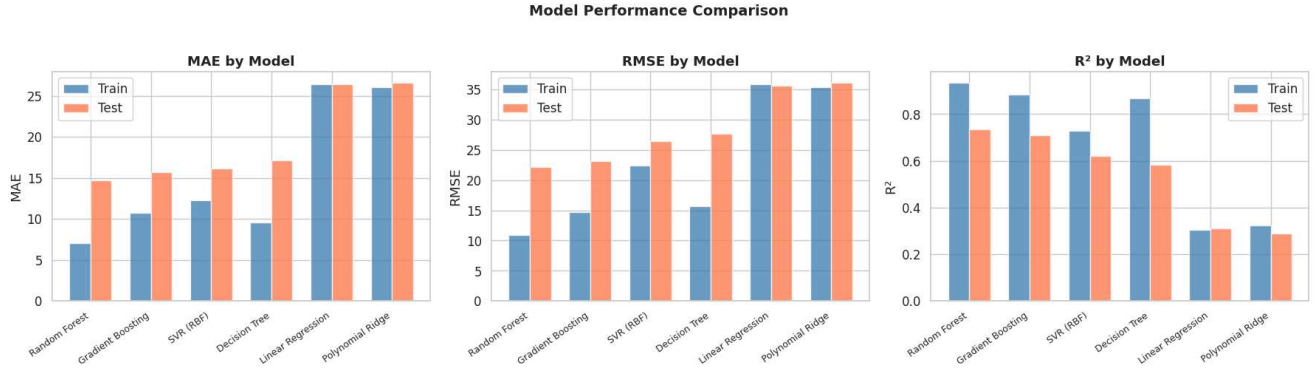

```
Residual statistics for Random Forest:
  Mean: -0.16
  Std:  22.11
  Min:  -102.63
  Max:  125.95
```

```
# ── Learning Curve for Best Model ──────────────────────────────────────
# Determine which X to use based on model type
if best_model_name in ['Linear Regression', 'Polynomial Ridge', 'SVR (RBF)']:
    X_lc, y_lc = X_train_scaled, y_train
else:
    X_lc, y_lc = X_train_unscaled, y_train

best_model_obj = best_res['model']

train_sizes, train_scores, val_scores = learning_curve(
    best_model_obj, X_lc, y_lc, cv=5,
    train_sizes=np.linspace(0.1, 1.0, 8),
    scoring='neg_mean_squared_error', n_jobs=-1
)

train_rmse = np.sqrt(-train_scores)
val_rmse = np.sqrt(-val_scores)

fig, ax = plt.subplots(figsize=(10, 5))
ax.fill_between(train_sizes, train_rmse.mean(axis=1) - train_rmse.std(axis=1),
                train_rmse.mean(axis=1) + train_rmse.std(axis=1), alpha=0.1, color='blue')
ax.fill_between(train_sizes, val_rmse.mean(axis=1) - val_rmse.std(axis=1),
                val_rmse.mean(axis=1) + val_rmse.std(axis=1), alpha=0.1, color='orange')
ax.plot(train_sizes, train_rmse.mean(axis=1), 'o-', color='blue', label='Training RMSE')
ax.plot(train_sizes, val_rmse.mean(axis=1), 'o-', color='orange', label='Validation RMSE')
ax.set_xlabel('Training Set Size')
ax.set_ylabel('RMSE')
ax.set_title(f'Learning Curve ({best_model_name})', fontsize=13, fontweight='bold')
ax.legend()
plt.tight_layout()
plt.savefig('figures/08_learning_curve.png', dpi=150, bbox_inches='tight')
plt.show()

print("\nOverfitting Analysis:")
gap = val_rmse.mean(axis=1)[-1] - train_rmse.mean(axis=1)[-1]
print(f"  Final Train RMSE: {train_rmse.mean(axis=1)[-1]:.2f}")
print(f"  Final Val RMSE:   {val_rmse.mean(axis=1)[-1]:.2f}")
print(f"  Gap:              {gap:.2f}")
if gap / val_rmse.mean(axis=1)[-1] > 0.2:
    print("  → Some overfitting detected (gap > 20% of val RMSE)")
else:
    print("  → Acceptable generalization (gap ≤ 20% of val RMSE)")
```




```
Overfitting Analysis:
  Final Train RMSE: 11.25
  Final Val RMSE:   24.02
  Gap:              12.78
  → Some overfitting detected (gap > 20% of val RMSE)
```

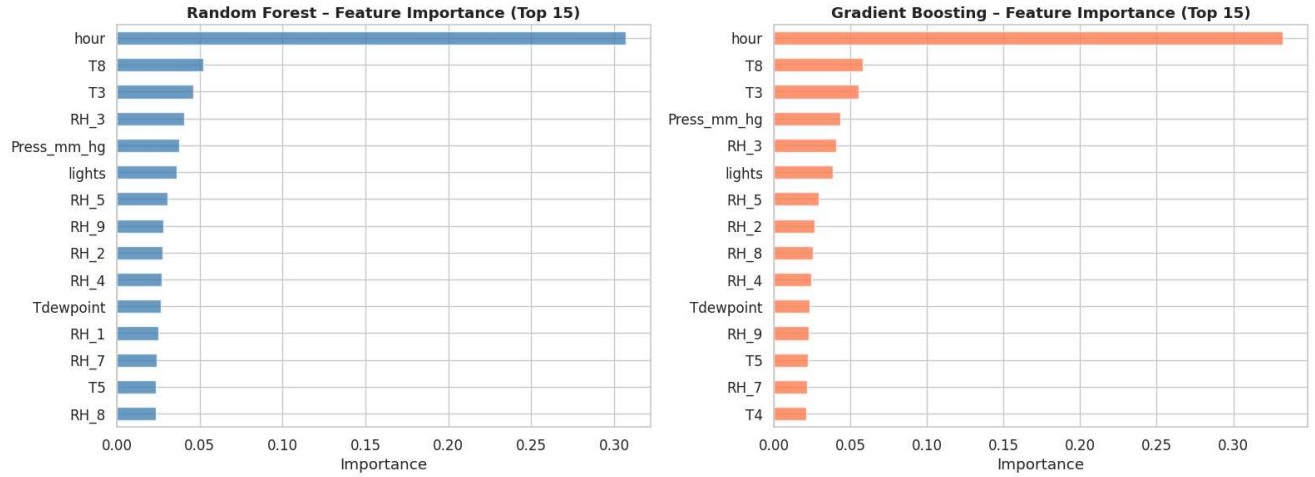
```
# ── Bar Chart: Model Comparison ──────────────────────────────────────
fig, axes = plt.subplots(1, 3, figsize=(18, 5))

models = comparison_df['Model'].values
x = np.arange(len(models))
width = 0.35

# MAE
axes[0].bar(x - width/2, comparison_df['Train MAE'], width, label='Train', color='steelblue', alpha=0.8)
axes[0].bar(x + width/2, comparison_df['Test MAE'], width, label='Test', color='coral', alpha=0.8)
axes[0].set_xticks(x)
axes[0].set_xticklabels(models, rotation=35, ha='right', fontsize=9)
axes[0].set_ylabel('MAE')
axes[0].set_title('MAE by Model', fontweight='bold')
axes[0].legend()

# RMSE
axes[1].bar(x - width/2, comparison_df['Train RMSE'], width, label='Train', color='steelblue', alpha=0.8)
axes[1].bar(x + width/2, comparison_df['Test RMSE'], width, label='Test', color='coral', alpha=0.8)
axes[1].set_xticks(x)
axes[1].set_xticklabels(models, rotation=35, ha='right', fontsize=9)
axes[1].set_ylabel('RMSE')
axes[1].set_title('RMSE by Model', fontweight='bold')
axes[1].legend()

# R²
axes[2].bar(x - width/2, comparison_df['Train R²'], width, label='Train', color='steelblue', alpha=0.8)
axes[2].bar(x + width/2, comparison_df['Test R²'], width, label='Test', color='coral', alpha=0.8)
axes[2].set_xticks(x)
axes[2].set_xticklabels(models, rotation=35, ha='right', fontsize=9)
axes[2].set_ylabel('R²')
axes[2].set_title('R² by Model', fontweight='bold')
axes[2].legend()

plt.suptitle('Model Performance Comparison', fontsize=14, fontweight='bold', y=1.02)
plt.tight_layout()
plt.savefig('figures/09_model_comparison.png', dpi=150, bbox_inches='tight')
plt.show()
```



7. Error Analysis & Interpretation

```
# ── Feature Importance (Tree-based models) ──────────────────────────
feature_names = list(X.columns)

fig, axes = plt.subplots(1, 2, figsize=(16, 6))

# Random Forest feature importance
```

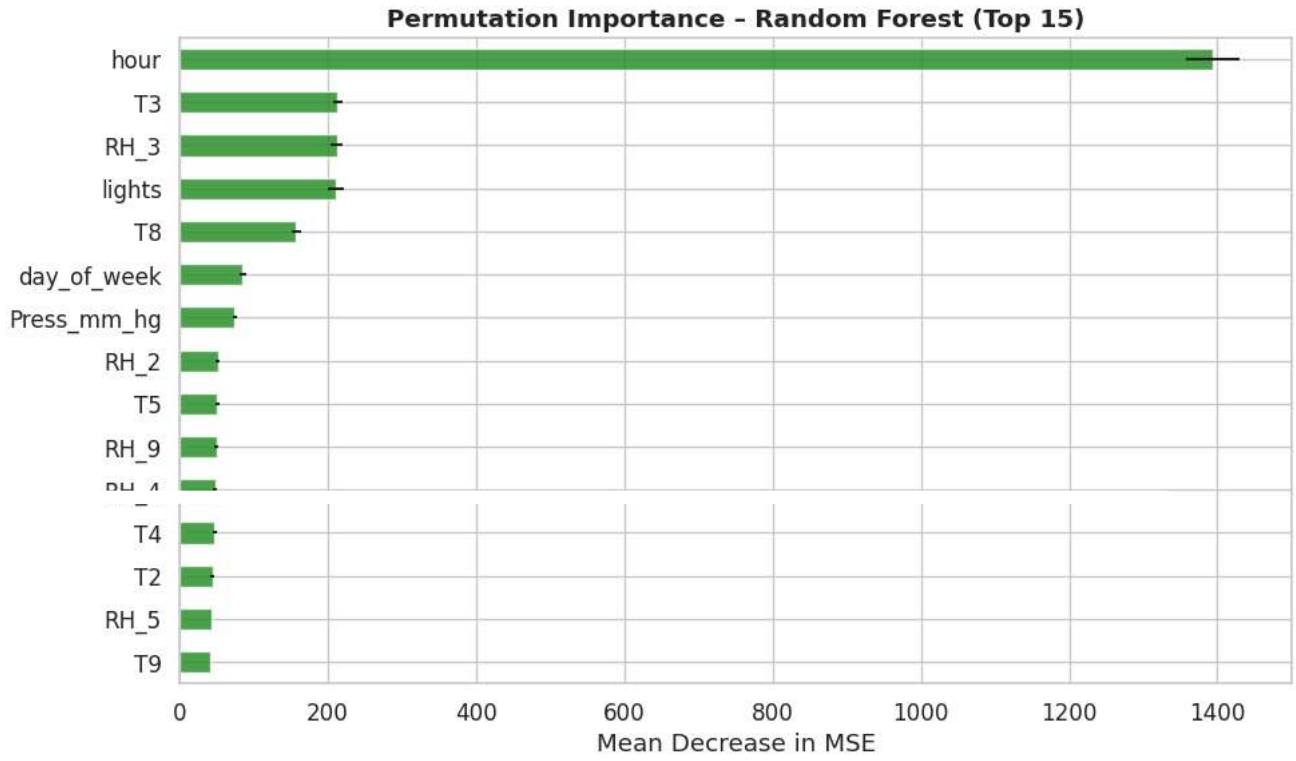
```
rf_model = results['Random Forest']['model']
rf_imp = pd.Series(rf_model.feature_importances_, index=feature_names).sort_values(ascending=True)
rf_imp.tail(15).plot.barh(ax=axes[0], color='steelblue', alpha=0.8)
axes[0].set_title('Random Forest – Feature Importance (Top 15)', fontweight='bold')
axes[0].set_xlabel('Importance')

# Gradient Boosting feature importance
gb_model = results['Gradient Boosting']['model']
gb_imp = pd.Series(gb_model.feature_importances_, index=feature_names).sort_values(ascending=True)
gb_imp.tail(15).plot.barh(ax=axes[1], color='coral', alpha=0.8)
axes[1].set_title('Gradient Boosting – Feature Importance (Top 15)', fontweight='bold')
axes[1].set_xlabel('Importance')

plt.tight_layout()
plt.savefig('figures/10_feature_importance.png', dpi=150, bbox_inches='tight')
plt.show()

print("Top 5 features by importance:")
print("  Random Forest:      ", list(rf_imp.tail(5).index))
print("  Gradient Boosting:  ", list(gb_imp.tail(5).index))
```



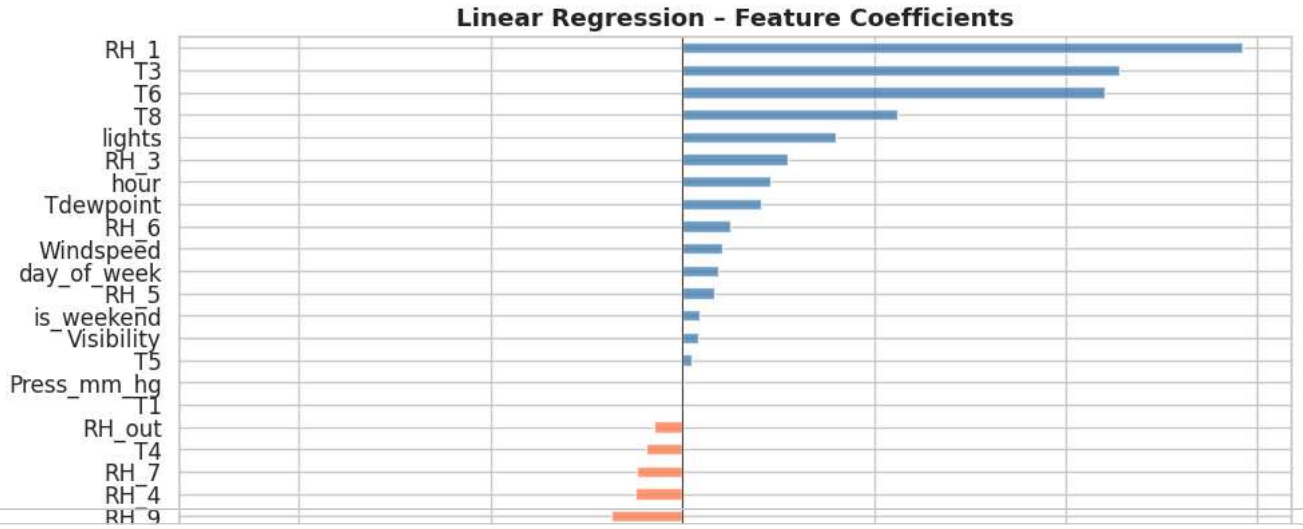

```
Top 5 features by importance:
  Random Forest:      ['Press_mm_hg', 'RH_3', 'T3', 'T8', 'hour']
  Gradient Boosting:  ['RH_3', 'Press_mm_hg', 'T3', 'T8', 'hour']
```

```
# ── Permutation Importance (Best Model) ──────────────────────────────
if best_model_name in ['Linear Regression', 'Polynomial Ridge', 'SVR (RBF)']:
    X_perm, y_perm = X_test_scaled, y_test
else:
    X_perm, y_perm = X_test_unscaled, y_test

perm_result = permutation_importance(
    best_res['model'], X_perm, y_perm,
    n_repeats=10, random_state=RANDOM_STATE, scoring='neg_mean_squared_error'
)

perm_imp = pd.Series(perm_result.importances_mean, index=feature_names).sort_values(ascending=True)

fig, ax = plt.subplots(figsize=(10, 6))
perm_imp.tail(15).plot.barh(ax=ax, color='forestgreen', alpha=0.8,
                            xerr=pd.Series(perm_result.importances_std, index=feature_names).sort_values(ascending=Tr
ax.set_title(f'Permutation Importance – {best_model_name} (Top 15)', fontweight='bold')
ax.set_xlabel('Mean Decrease in MSE')
plt.tight_layout()
```

```
plt.savefig('figures/11_permutation_importance.png', dpi=150, bbox_inches='tight')
plt.show()
```




```
# ── Linear Model Coefficients ──────────────────────────────────────
lr_model = results['Linear Regression']['model']
coefs = pd.Series(lr_model.coef_, index=feature_names).sort_values()

fig, ax = plt.subplots(figsize=(10, 6))
colors = ['coral' if c < 0 else 'steelblue' for c in coefs]
coefs.plot.barh(ax=ax, color=colors, alpha=0.8)
ax.set_title('Linear Regression – Feature Coefficients', fontweight='bold')
ax.set_xlabel('Coefficient Value (standardized)')
ax.axvline(0, color='black', linewidth=0.5)
plt.tight_layout()
plt.savefig('figures/12_linear_coefficients.png', dpi=150, bbox_inches='tight')
plt.show()

print("Top 5 positive coefficients:")
print(coefs.tail(5).round(4).to_string())
print("\nTop 5 negative coefficients:")
print(coefs.head(5).round(4).to_string())
```




```
# ── Error Analysis: Large-Error Cases ──────────────────────────────
y_pred_best = best_res['y_test_pred']
errors = y_test.values - y_pred_best
error_std = errors.std()

# Cases where |error| > 2 standard deviations
large_error_mask = np.abs(errors) > 2 * error_std
n_large = large_error_mask.sum()

print(f"Error standard deviation: {error_std:.2f}")
print(f"Large-error cases (|error| > 2σ = {2*error_std:.2f}):")
print(f"  Count: {n_large} ({n_large/len(errors)*100:.1f}% of test set)")
print(f"  Mean actual value:    {y_test.values[large_error_mask].mean():.1f} Wh")
print(f"  Mean predicted value: {y_pred_best[large_error_mask].mean():.1f} Wh")
print(f"  Mean absolute error:  {np.abs(errors[large_error_mask]).mean():.1f} Wh")

fig, ax = plt.subplots(figsize=(12, 5))
ax.scatter(range(len(errors)), errors, s=3, alpha=0.3, color='steelblue', label='Normal errors')
ax.scatter(np.where(large_error_mask)[0], errors[large_error_mask], s=8, alpha=0.6,
           color='red', label=f'Large errors (>{2*error_std:.0f} Wh)')
ax.axhline(0, color='black', linewidth=0.5)
ax.axhline(2*error_std, color='red', linestyle='--', alpha=0.5)
ax.axhline(-2*error_std, color='red', linestyle='--', alpha=0.5)
ax.set_xlabel('Test Sample Index')
ax.set_ylabel('Prediction Error (Wh)')
ax.set_title(f'Error Distribution – {best_model_name}', fontweight='bold')
ax.legend()
plt.tight_layout()
plt.savefig('figures/13_error_analysis.png', dpi=150, bbox_inches='tight')
plt.show()

print("\nThe model struggles most with high-consumption events, where usage spikes")
print("due to occupancy changes or simultaneous appliance usage. This is expected")
print("since these events are inherently harder to predict from environmental features alone.")
```

```
Error standard deviation: 22.11
Large-error cases (|error| > 2σ = 44.21):
  Count: 272 (6.9% of test set)
  Mean actual value:    135.8 Wh
  Mean predicted value: 112.3 Wh
  Mean absolute error:  62.6 Wh
```