XGBoost for Prediciting Energy Usage in Buildings

Notebook Overview

This notebook demonstrates:

1. **Data Preprocessing**:
    - Cleaning, aligning, and preparing building-specific data.
2. **Model Evaluation**:
    - Using a trained XGBoost model to predict energy consumption.
3. **Visualization**:
    - Comparing actual and predicted values with interactive plots.

Use cases:

- **Model Validation**: Ensure predictions align with ground truth data.
- **Error Analysis**: Identify where and why predictions deviate from reality.
- **Presentation**: Generate clear visuals for stakeholder communication.

```
import pandas as pd
import numpy as np
import os
import matplotlib.pyplot as plt
import re
import json
import gc
import psutil
import xgboost as xgb
import random
import joblib
import plotly.express as px

from multiprocessing import Pool, cpu_count
from concurrent.futures import ProcessPoolExecutor, as_completed
from functools import partial

from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from sklearn.model_selection import RandomizedSearchCV
```

```
# if you are using Google Colab, you can easily mount the drive and access
# the data.
from google.colab import drive
drive.mount('/content/drive', force_remount=True)
```

```
Mounted at /content/drive
```

Load the datasets

```
# In order for you to use the data, you need to update these paths.
PROCESSED = f'/content/drive/MyDrive/Team-Fermata-Energy/processed_data/' # https://drive.google.com/drive/folders/
BUILDINGS = f'{PROCESSED}processed_weather_load_w_timestamp/' # https://drive.google.com/drive/folders/1kW3Ip5_xm6
```

```
with open(f'{PROCESSED}subset20_data.json', 'r') as test_train_file:
    test_train_ids = json.load(test_train_file)

train_ids = [int(bldg_id.replace('.csv', '')) for bldg_id in test_train_ids['train_bldg_ids']]
test_ids = [int(bldg_id.replace('.csv', '')) for bldg_id in test_train_ids['test_bldg_ids']]
```

```
df_metadata = pd.read_csv(f"{PROCESSED}md_one_hot_encoded_subset20.csv")
df_metadata.head(20)
```

	bldg_id	in.state	in.vintage	in.sqft	in.building_america_climate_zone_Cold	in.building_america_climate_zo
0	105885	10	3	750000.0	0	
1	305819	40	2	150000.0	0	
2	305934	40	4	350000.0	0	
3	317044	40	3	350000.0	0	
4	32	1	6	37500.0	0	
5	64	1	0	37500.0	0	
6	103	1	4	75000.0	0	
7	112	1	3	7500.0	0	
8	277	1	0	37500.0	0	
9	355	1	7	17500.0	0	
10	363	1	5	37500.0	0	
11	379	1	1	17500.0	0	
12	417	1	0	17500.0	0	
13	530	1	2	17500.0	0	
14	575	1	3	3000.0	0	
15	611	1	5	7500.0	0	
16	633	1	6	37500.0	0	
17	864	1	4	37500.0	0	
18	1025	1	4	75000.0	0	
19	1215	1	3	3000.0	0	

20 rows × 40 columns

If you want to load in the model and just evaluate, you can do so here!

```
model = joblib.load(f'{PROCESSED}xgb_model2.pkl')
```

Functions for Training the Model

Training the Model

This function prepares data and trains the XGBoost model for energy consumption prediction.

Steps:

1. Train the model: Set hyperparameters, use early stopping, and monitor performance with metrics like RMSE or MAE.
2. Analyze feature importance: Understand key drivers of predictions.

Outputs:

- A trained model

```
def preprocess_bldg_optimized(bldg_id, df_metadata):
    """
    Preprocesses data for a single building.

    Parameters:
    - bldg_id (int): Building ID.
    - df_metadata (DataFrame): Metadata DataFrame.

    Returns:
    - X (DataFrame): Feature DataFrame.
    - y (Series): Target variable.
    """
    try:
        # Load CSV with optimized data types
        df_bldg = pd.read_csv(
            f"{BUILDINGS}{bldg_id}.csv",
            dtype={
                'bldg_id': 'int32',
                'minute': 'int8',
                'out_electricity_total_energy_consumption': 'float32',
                # Add other columns with appropriate types if known
                # Example:
                # 'temperature': 'float32',
                # 'humidity': 'float32',
                # 'heat_index': 'float32',
                # 'location': 'category',
            }
        )

        # Clean column names
        df_bldg.columns = [re.sub(r"[^A-Za-z0-9_]+", "_", col) for col in df_bldg.columns]

        # Filter rows where 'minute' == 0
        df_bldg = df_bldg[df_bldg['minute'] == 0]

        # Merge with metadata
        bldg_metadata = df_metadata[df_metadata['bldg_id'] == bldg_id]
        df_bldg = df_bldg.merge(bldg_metadata, on='bldg_id', how='left')

        # Prepare features and target
        y = df_bldg['out_electricity_total_energy_consumption']
        X = df_bldg.drop(columns=['out_electricity_total_energy_consumption', 'timestamp', 'bldg_id'])

        return X, y
    except Exception as e:
        print(f"Error processing building ID {bldg_id}: {e}")
        return pd.DataFrame(), pd.Series()

def train_in_chunks_optimized(df_metadata, train_bldg_ids, best_param):
    """
    Trains an XGBoost model on data from multiple buildings using GPU acceleration.

    Parameters:
    - df_metadata (DataFrame): Metadata DataFrame.
    - train_bldg_ids (list): List of building IDs for training.
    - best_param (dict): Best hyperparameters for XGBoost.

    Returns:
```

```
    Returns:
    - model (XGBRegressor): Trained XGBoost model.
    """
    # Determine number of parallel processes
    n_jobs = max(cpu_count() - 1, 1)  # Reserve one core for the system
    print(f"Using {n_jobs} parallel processes for data preprocessing.")

    # Initialize multiprocessing Pool
    with Pool(processes=n_jobs) as pool:
        # Partial function to pass df_metadata
        func = partial(preprocess_bldg_optimized, df_metadata=df_metadata)

        # Map the preprocessing function to building IDs
        results = pool.map(func, train_bldg_ids)

    # Filter out any failed preprocessing results
    results = [res for res in results if not res[0].empty]

    if not results:
        raise ValueError("No data was successfully preprocessed.")

    # Concatenate all preprocessed data
    X_all, y_all = zip(*results)
    X_all = pd.concat(X_all, ignore_index=True)
    y_all = pd.concat(y_all, ignore_index=True)

    # Clean up intermediate results
    del results
    gc.collect()

    print("Starting model training on GPU...")

    # Initialize and train the XGBoost model with GPU support
    model = xgb.XGBRegressor(
        tree_method='hist',          # Use 'hist' for faster training with GPU via 'device'
        device='cuda',               # Specify to use GPU
        n_jobs=-1,                   # Utilize all available CPU cores for data preprocessing
        enable_categorical=True,     # Enable categorical feature support
        **best_param,                # Additional hyperparameters
        reg_alpha=1.0,
        reg_lambda=1.0,
        random_state=42,
        verbosity=1                  # Set to 1 for basic logging
    )

    # Fit the model
    model.fit(X_all, y_all, verbose=True)

    # Monitor memory usage
    process = psutil.Process()
    memory_usage = process.memory_info().rss
    print(f"Memory Usage After Training: {memory_usage / (1024 ** 2):.2f} MB")

    # Clean up to free memory
    del X_all
    del y_all
    gc.collect()

    return model

```

```
model = xgb.XGBRegressor(
    tree_method='hist',          # Use 'hist' for faster training with GPU via 'device'
    device='cuda',               # Specify to use GPU
    n_jobs=-1,                   # Utilize all available CPU cores for data preprocessing
```

```
    enable_categorical=True,     # Enable categorical feature support
    reg_alpha=1.0,
    reg_lambda=1.0,
    random_state=42,
    verbosity=1                  # Set to 1 for basic logging
)
```

```
best_param = {'subsample': 0.8, 'n_estimators': 300, 'max_depth': 6, 'learning_rate': 0.01, 'colsample_bytree': 0.8
model = train_in_chunks_optimized(df_metadata, train_ids, best_param)
```

```
Using 11 parallel processes for data preprocessing.
Starting model training on GPU...
Memory Usage After Training: 37627.39 MB
```

Find best hyperparameters for the model.

I've already run this code so you don't have to.

[ ] ↳ *2 cells hidden*

Save and Test Model

Evaluating the Model

This function tests the trained model on unseen data and calculates performance metrics.

Steps:

1. Load and preprocess test data: Align features with the trained model.
2. Predict and compare: Generate predictions and calculate metrics like SMAPE.
3. Monitor performance: Assess memory usage and model accuracy.

Outputs:

- Evaluation metrics and insights into prediction accuracy.

```
# Save the model
# joblib.dump(model, PATHGOESHERE)
```

```
['/content/drive/MyDrive/Team-Fermata-Energy/processed_data/xgb_model2.pkl']
```

```
df_test_bldg = pd.read_csv(f"{BUILDINGS}32.csv")
df_test_bldg.columns = [re.sub(r"[^A-Za-z0-9_]+", "_", col) for col in df_test_bldg.columns]
df_test_bldg.columns
```

```
Index(['timestamp', 'out_electricity_total_energy_consumption',
       'Dry_Bulb_Temperature_C_', 'Relative_Humidity_', 'heat_index', 'minute',
       'hour', 'day', 'month', 'is_weekday', 'is_holiday', 'max_load_hourly',
       'min_load_hourly', 'max_temp_hourly', 'min_temp_hourly', 'bldg_id'],
      dtype='object')
```

```
# Pre-create numpy arrays to store the metrics
# num_test_ids = len(test_ids)  # Replace test_ids with your list of test building IDs
smape_values = []

def calculate_smape(y_true, y_pred):
    """
    Calculate Symmetric Mean Absolute Percentage Error (SMAPE).

```

```
    Parameters:
        y_true: Actual values.
        y_pred: Predicted values.

    Returns:
        SMAPE value as a percentage.
    """
    denominator = (np.abs(y_true) + np.abs(y_pred)) / 2
    diff = np.abs(y_true - y_pred)
    smape = np.mean(diff / denominator) * 100  # Percentage
    return smape

def evaluate_model(model, df_metadata, test_ids):
    """
    Loops through each test building ID, evaluates the model, and stores metrics.

    Parameters:
        model: Trained XGBoost model.
        df_metadata: DataFrame containing building metadata.
        test_ids: List of test building IDs.
    """
    for idx, bldg_id in enumerate(test_ids):
        # Load and preprocess test building data
        df_test_bldg = pd.read_csv(f"{BUILDINGS}{bldg_id}.csv")
        df_test_bldg.columns = [re.sub(r"[^A-Za-z0-9_]+", "_", col) for col in df_test_bldg.columns]

        df_test_bldg = df_test_bldg[df_test_bldg['minute'] == 0]

        # Merge metadata
        test_bldg_metadata = df_metadata[df_metadata['bldg_id'] == bldg_id]
        df_test_bldg = df_test_bldg.merge(test_bldg_metadata, on='bldg_id', how='left')

        # Prepare features (X_test) and target (y_test)
        y_test = df_test_bldg['out_electricity_total_energy_consumption']
        X_test = df_test_bldg.drop(columns=['out_electricity_total_energy_consumption', 'timestamp', 'bldg_id'])

        # print(y_test.describe())
        # Predict and evaluate metrics
        pred = model.predict(X_test)
        smape = calculate_smape(y_test, pred)
        smape_values.append(smape)

        # Monitor memory usage
        process = psutil.Process()
        memory_usage = process.memory_info().rss
        print(f"Memory Usage: {memory_usage / (1024 ** 2):.2f} MB")

        # Clean up memory
        del df_test_bldg, X_test, y_test, pred, process, memory_usage
        gc.collect()
```

```
# Example usage
random.seed(521)
evaluate_model(model, df_metadata, random.choices(test_ids, k = 5))

# performance metrics
smape_array = np.array(smape_values)

print(f'Mean SMAPE: {mean_smape}')
```

```
Memory Usage: 19700.38 MB
Memory Usage: 19700.38 MB
Memory Usage: 19700.38 MB
Memory Usage: 19700.38 MB
Memory Usage: 19700.38 MB
```

```
Mean SMAPE: 21.48865254136774
```

Visualizations and Misc

Align Features for Model Compatibility

The `align_features` function ensures that the test data matches the trained model's expected input format. This is critical for avoiding feature name mismatches, which occur when:

- The test data contains columns that were not part of the training data.
- The test data is missing columns present during model training.

Key steps:

1. **Add Missing Columns**: Columns that are in the model's feature list but not in the test data are added with default values (e.g., `0` for one-hot encoded features).
2. **Drop Extra Columns**: Any columns in the test data but not required by the model are removed.
3. **Order Matching**: Ensures that the columns are in the same order as the model's training data.

This function ensures smooth prediction and prevents runtime errors due to feature mismatch.

```python
def align_features(X_test, model):
    """
    Aligns test data features with the features expected by the model.

    Parameters:
        X_test (pd.DataFrame): Test data features.
        model: Trained model (XGBoost or similar).

    Returns:
        pd.DataFrame: Aligned test data.
    """
    # Get the feature names from the model
    model_features = model.get_booster().feature_names

    # Add missing columns to X_test
    for col in model_features:
        if col not in X_test.columns:
            X_test[col] = 0  # Default value for missing features

    # Drop extra columns not in the model
    X_test = X_test[model_features]

    return X_test


def visualize_time_series(df_metadata, bldg_id, model):
    """
    Load and visualize time series data for energy consumption and model predictions.

    Parameters:
        df_metadata (pd.DataFrame): DataFrame containing building metadata.
        bldg_id (str): Building ID for which the data is visualized.
        model: Trained XGBoost model for predictions.
    """
    # Load the specific building data
    file_path = f"{BUILDINGS}{bldg_id}.csv"
    df_bldg = pd.read_csv(file_path)

    # Clean column names
```

```
    df_bldg.columns = [re.sub(r"[^A-Za-z0-9_]+", "_", col) for col in df_bldg.columns]
    print(df_bldg.columns)

    # Convert timestamp to datetime
    df_bldg['timestamp'] = pd.to_datetime(df_bldg['timestamp'])

    # Filter rows where minute == 0
    df_bldg = df_bldg[df_bldg['minute'] == 0]

    # Prepare features for prediction
    X_test = df_bldg.drop(columns=['out_electricity_total_energy_consumption', 'timestamp', 'bldg_id'])
    X_test = align_features(X_test, model)

    # Predict using the model
    df_bldg['Predicted_Energy_Consumption'] = model.predict(X_test)

    # Melt the DataFrame for easier plotting
    df_long = df_bldg.melt(
        id_vars='timestamp',
        value_vars=[
            'out_electricity_total_energy_consumption',
            'Predicted_Energy_Consumption'
        ],
        var_name='Measurement',
        value_name='Value'
    )

    # Replace specific measurement names for better legend labels
    df_long['Measurement'] = df_long['Measurement'].replace({
        'out_electricity_total_energy_consumption': 'Actual Energy Consumption',
        'Predicted_Energy_Consumption': 'Predicted Energy Consumption'
    })

    # Create the time series plot
    fig = px.line(
        df_long,
        x='timestamp',
        y='Value',
        color='Measurement',
        labels={'Value': 'Measurement Value', 'timestamp': 'Time'},
        title=f"Time Series Data for Building ID: {bldg_id} (Actual vs Predicted)"
    )

    # Show the plot
    fig.show()
```

Visualize Time Series Data (Actual vs Predicted)

The `visualize_time_series` function provides a comprehensive view of the model's performance by comparing actual energy consumption against the predicted values.

Key components:

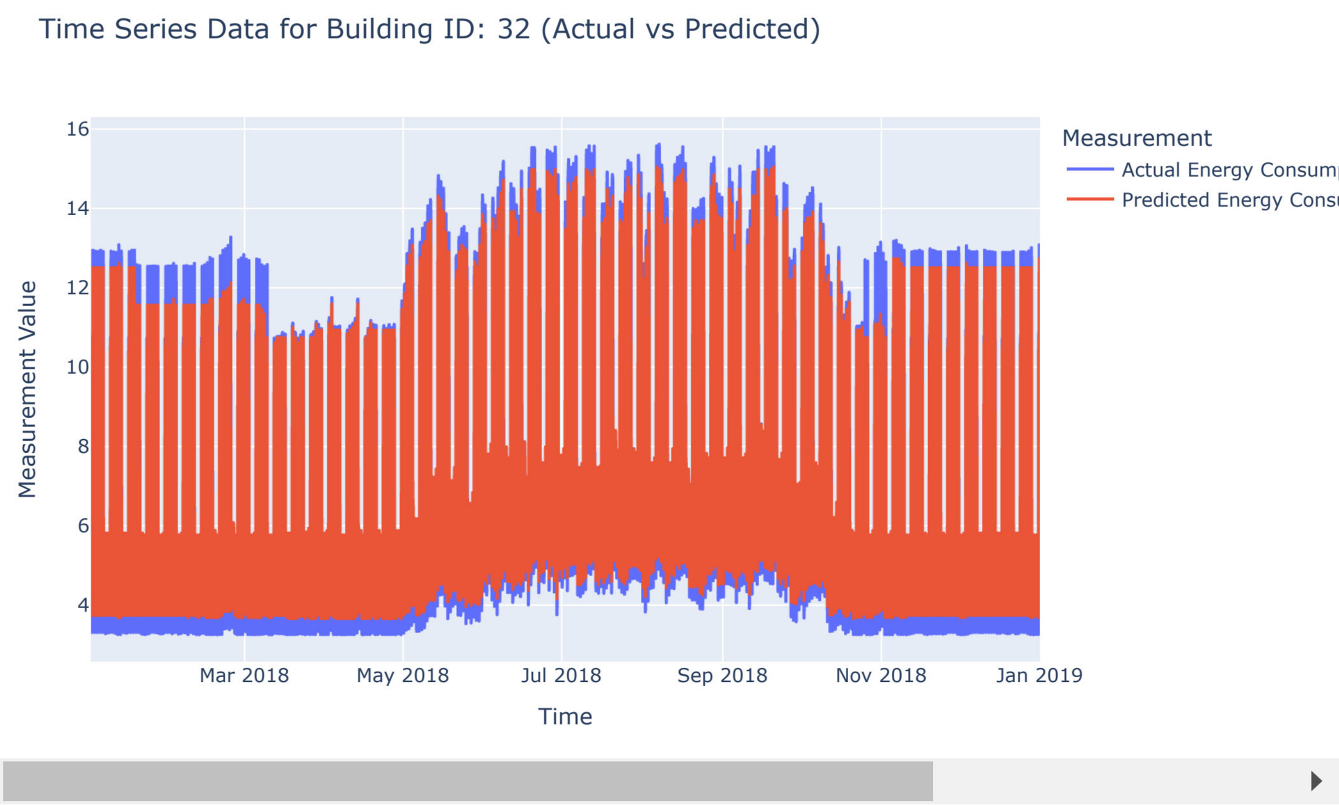
1. **Data Preparation**:
    - Load the building-specific dataset and filter rows where `minute == 0` for consistent granularity.
    - Align features with the trained model using the `align_features` function.
2. **Predictions**:
    - The model predicts energy consumption using preprocessed test data.
    - Predicted values are added as a new column to the dataset.
3. **Visualization**:

- A line plot compares actual and predicted energy consumption over time.
- Interactive legends allow users to focus on specific measurements.

This visualization helps identify patterns, trends, and areas where the model's predictions deviate from the actual values.

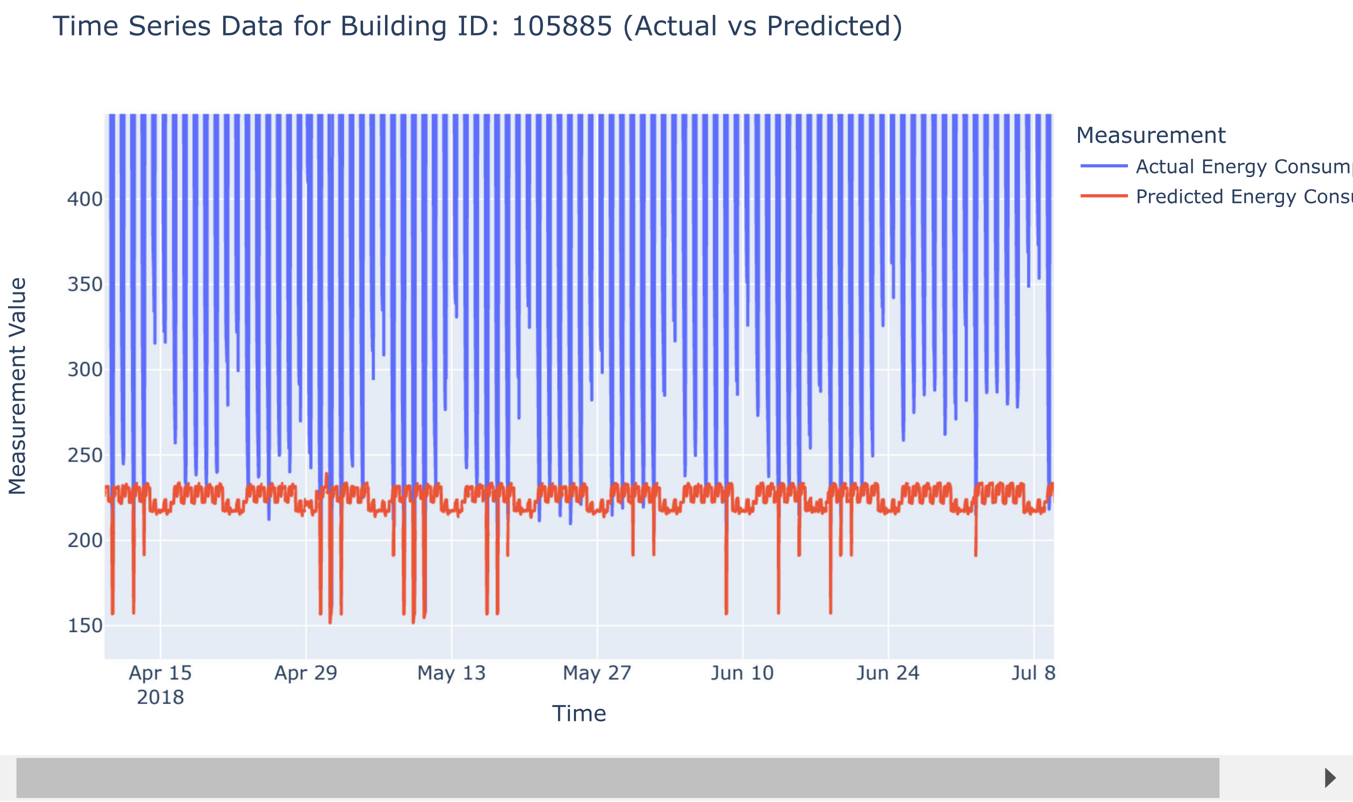
```
1 visualize_time_series(df_metadata, '32', model)
```

```
Index(['timestamp', 'out_electricity_total_energy_consumption',
       'Dry_Bulb_Temperature_C_', 'Relative_Humidity_', 'heat_index', 'minute',
       'hour', 'day', 'month', 'is_weekday', 'is_holiday', 'max_load_hourly',
       'min_load_hourly', 'max_temp_hourly', 'min_temp_hourly', 'bldg_id'],
      dtype='object')
```



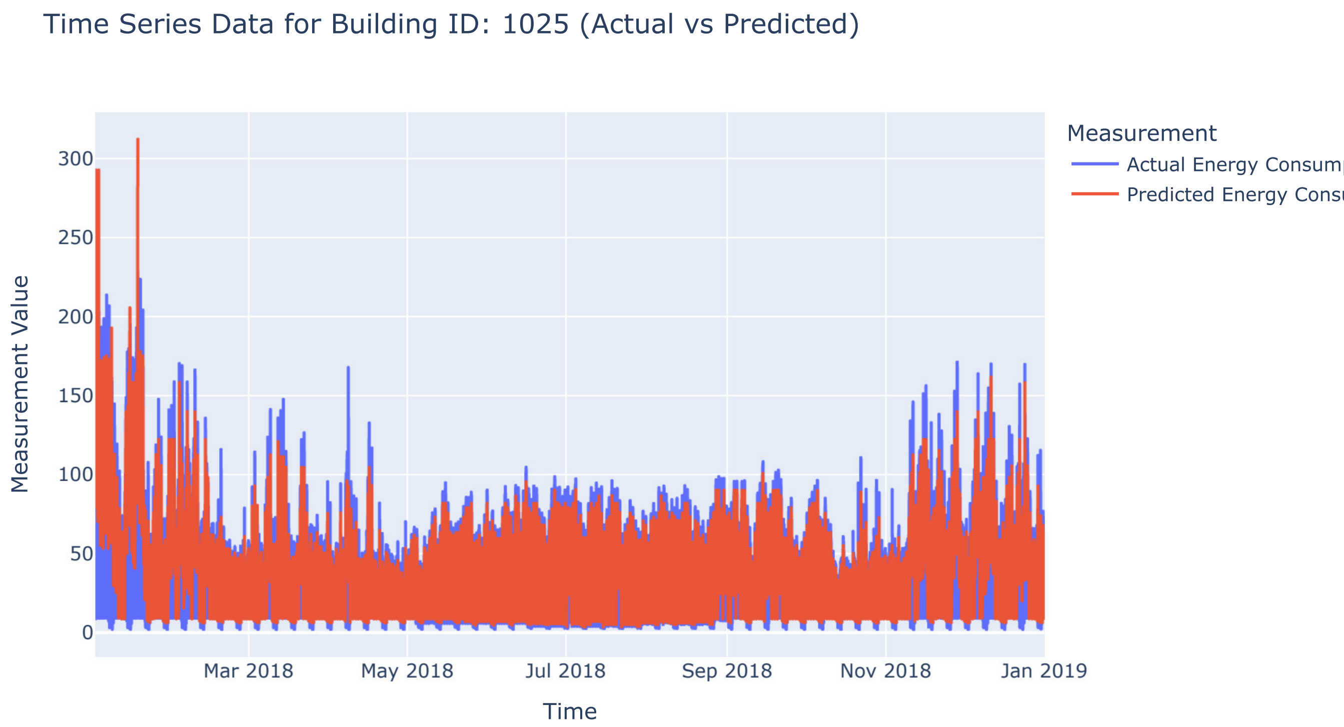

```
1 visualize_time_series(df_metadata, '105885', model)
```

```
Index(['timestamp', 'out_electricity_total_energy_consumption',
       'Dry_Bulb_Temperature_C_', 'Relative_Humidity_', 'heat_index', 'minute',
       'hour', 'day', 'month', 'is_weekday', 'is_holiday', 'max_load_hourly',
       'min_load_hourly', 'max_temp_hourly', 'min_temp_hourly', 'bldg_id'],
      dtype='object')
```




```
1 visualize_time_series(df_metadata, '1025', model)
```

```
Index(['timestamp', 'out_electricity_total_energy_consumption',
       'Dry_Bulb_Temperature_C_', 'Relative_Humidity_', 'heat_index', 'minute',
       'hour', 'day', 'month', 'is_weekday', 'is_holiday', 'max_load_hourly',
       'min_load_hourly', 'max_temp_hourly', 'min_temp_hourly', 'bldg_id'],
      dtype='object')
```




```python
feature_importances = model.feature_importances_
features = []
features = model.get_booster().feature_names

# Create a DataFrame for better handling of the data
data = pd.DataFrame({
    'Feature': features,
    'Importance': feature_importances
})

# Sort features by importance for better visualization
data = data.sort_values(by='Importance', ascending=True)

# Plot the feature importances using Plotly
fig = px.bar(data, x='Importance', y='Feature', orientation='h',
             title='Feature Importances for Load Forecasting',
             labels={'Importance': 'Feature Importance', 'Feature': 'Feature'}
             text='Importance')

# Improve layout for better readability
fig.update_layout(
    xaxis_title="Feature Importance",
    yaxis_title="Feature",
    title_x=0.5,  # Center the title
    font=dict(size=12),
    showlegend=False,
    margin=dict(l=150, r=20, t=50, b=50),  # Adjust left margin for long label
    height=400 + 20 * len(features)  # Dynamically adjust height for label siz
)

# Add better formatting for the text
```

```
fig.update_traces(texttemplate='%{text:.2f}', textposition='outside')

# Display the plot
fig.show()
```