Electricity Consumption Prediction With Machine Learning Presentation

Ahmet Furkan Yorulmaz[1], Arda Erturhan[2], Batuhan Gümüş[3], Melis Can[4], Vehbi Ozan Güzel[5]

[1]*Software Engineering Department, Istanbul Atlas University, Istanbul (220504004)*
[2]*Software Engineering Department, Istanbul Atlas University, Istanbul (220504001)*
[3]*Software Engineering Department, Istanbul Atlas University, Istanbul (220504021)*
[4]*Software Engineering Department, Istanbul Atlas University, Istanbul (220504019)*
[5]*Software Engineering Department, Istanbul Atlas University, Istanbul (220504032)*

1. **Import Libraries**:
   Necessary libraries (pandas, numpy, seaborn, and matplotlib) are imported for data manipulation and visualization.

2. **Load Data**:
   The dataset is read from a CSV file located at 'Users/meliscan/machineProject/electricity_data.csv'.

3. **Combine and Convert Columns**:
   The Date and Time columns are merged into a new datetime column, which is converted to datetime format for easier processing.

4. **Filter Hourly Data**:
   Rows where the minute value is 0 are selected to focus on hourly data.

5. **Remove Unnecessary Columns**:
   The original Date and Time columns are deleted as they are no longer needed.

6. **Reset Index**:
   The index of the filtered dataset is reset to make it sequential and clean.

7. **Save the Filtered Dataset**:
   The modified dataset is saved to a new CSV file named 'hourly_electricity_data.csv'.

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

# Loading file
file_path = '/Users/meliscan/machineProject/electricity_data.csv'

data = pd.read_csv(file_path)

# Combining the Date and Time columns and converting them to datetime type
data['datetime'] = pd.to_datetime(data['Date'] + ' ' + data['Time'], format='%d/%m/%Y %H:%M:%S')

# Filtering rows where the minute is 0
hourly_data = data[data['datetime'].dt.minute == 0]

# Removing the original Date and Time columns
del hourly_data['Date']
del hourly_data['Time']

# Saving or inspecting the filtered dataset
hourly_data.reset_index(inplace=True, drop=True)  # To reset the index
print(hourly_data.head())

# Save the filtered data to a new CSV file
hourly_data.to_csv('hourly_electricity_data.csv', index=False)
```

1. **Load Filtered Dataset**:
   The previously saved filtered dataset 'hourly_electricity_data.csv' is read into a DataFrame.

2. **Convert Datetime Column**:
   The datetime column is explicitly converted to datetime format for further processing.

3. **Extract Date-Time Components**:
   New columns are created for year, month, day, hour, and weekday (where weekday is represented numerically: 0 = Monday, 6 = Sunday) by extracting them from the datetime column.

4. **Drop Datetime Column**:
   The datetime column is removed since its components are now extracted.

5. **Remove Missing Data**:
   Any rows with missing values are dropped to ensure a clean dataset.

6. **Preview Data**:
   The first few rows of the processed dataset are displayed using print(hourly_data.head()).

```python
file_path2 = '/Users/meliscan/machineProject/hourly_electricity_data.csv'
hourly_data = pd.read_csv(file_path2)

hourly_data['datetime'] = pd.to_datetime(hourly_data['datetime'])

# Extract year, month, day, hour by adding new columns
hourly_data['year'] = hourly_data['datetime'].dt.year
hourly_data['month'] = hourly_data['datetime'].dt.month
hourly_data['day'] = hourly_data['datetime'].dt.day
hourly_data['hour'] = hourly_data['datetime'].dt.hour
hourly_data['weekday'] = hourly_data['datetime'].dt.weekday  # day of the week (0: monday, 6: sunday)

hourly_data = hourly_data.drop(columns=['datetime'])

# Remove missing data
hourly_data = hourly_data.dropna()

print(hourly_data.head())
```

1.Separate Features and Target:

The dataset is split into input features (X) and the target variable (y), where y is the Global_active_power column, and X contains all other columns except Global_active_power.

2. Initial Split:

The data is divided into training (X_train, y_train) and temporary sets (X_temp, y_temp) using a 70%-30% split.

3. Further Split:

The temporary set (X_temp, y_temp) is split equally into test (X_test, y_test) and validation (X_val, y_val) sets, with a 50%-50% split.

4. Output Data Sizes:

The sizes of the training, test, and validation datasets are printed to verify the splits.

```python
# Separate input (X) and target (y) variables
X = hourly_data.drop(columns=['Global_active_power'])
y = hourly_data['Global_active_power']

# Split into training and temporary set (test + validation)
from sklearn.model_selection import train_test_split
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42)

# Split the test and validation sets
X_test, X_val, y_test, y_val = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)

print("Training Data Size:", X_train.shape)
print("Test Data Size:", X_test.shape)
print("Validation Data Size:", X_val.shape)
```

1.Model Definition

- Sequential defines a simple feedforward model where layers are stacked sequentially
- Input specifies the input shape based on the training data (X_train.shape[1])

2.Layers

- First Dense Layer
  - 64 neurons with ReLU activation
  - Includes L2 regularization (kernel_regularizer=l2(0.01)) to prevent overfitting
- Second Dense Layer
  - 32 neurons with ReLU activation
  - Also includes L2 regularization
- Output Dense Layer
  - Single neuron with linear activation for regression tasks

3.Model Compilation

- Optimizer: adam (efficient and adaptive optimization algorithm)
- Loss: mse (Mean Squared Error, common for regression)
- Metrics: mae (Mean Absolute Error, evaluates model performance)

4.Summary Display

- model.summary() displays the architecture and parameters of the model

```python
# Building the model
import tensorflow as tf
from tensorflow.keras.regularizers import l2
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input

model = Sequential([
    Input(shape=(X_train.shape[1],)),  # Input layer specifying the input shape
    Dense(64, activation='relu', kernel_regularizer=l2(0.01)),  # First layer with L2 regularization
    Dense(32, activation='relu', kernel_regularizer=l2(0.01)),
    Dense(1, activation='linear')  # Output layer for regression
])

# Compile the model
model.compile(optimizer='adam', loss='mse', metrics=['mae'])

# Display the model summary
model.summary()
```

1. Model Summary

- Displays the details of the model architecture, including layer types, output shapes, and the number of trainable parameters
- The model consists of three Dense layers
  - First layer has 768 parameters
  - Second layer has 2,080 parameters
  - Third layer has 33 parameters
- Total parameters: 2,881, all of which are trainable

2. Training the Model

- model.fit trains the model using the training data (X_train, y_train)
- epochs=25: The model iterates 25 times over the dataset during training
- batch_size=32: The dataset is divided into batches of 32 samples for each iteration
- validation_data: Specifies validation data (X_val, y_val) to evaluate the model after each epoch
- verbose=1: Displays detailed training progress information during execution

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 64)	768
dense_1 (Dense)	(None, 32)	2,080
dense_2 (Dense)	(None, 1)	33

Total params: 2,881 (11.25 KB)

Trainable params: 2,881 (11.25 KB)

Non-trainable params: 0 (0.00 B)

```python
# Training the model
history = model.fit(
    X_train, y_train,
    epochs=25,
    batch_size=32,
    validation_data=(X_val, y_val),
    verbose=1
)
```

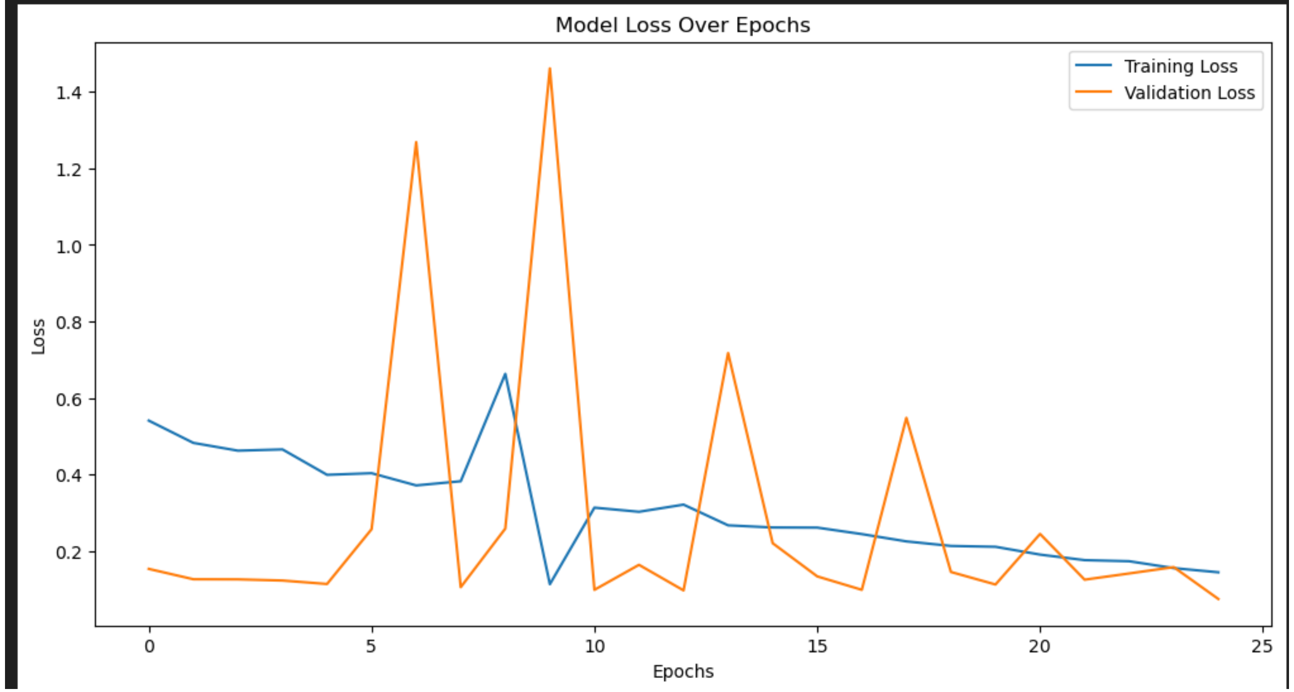
1.Plotting Training History

- A graph is created using Matplotlib to visualize the training and validation loss over epochs
- plt.plot(history.history['loss']): Plots the training loss values for each epoch
- plt.plot(history.history['val_loss']): Plots the validation loss values for each epoch
- plt.xlabel('Epochs') and plt.ylabel('Loss'): Label the axes for better interpretation
- plt.title('Model Loss Over Epochs'): Adds a title to the graph
- plt.legend: Adds a legend to differentiate between training and validation loss

2.Interpretation of the Graph

- The blue line represents the **Training Loss**
- The orange line represents the **Validation Loss**
- The graph shows how the loss values decrease and stabilize as training progresses, indicating the model's performance on the training and validation data over 25 epochs

```python
# Plot training history
plt.figure(figsize=(12, 6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Model Loss Over Epochs')
plt.legend()
plt.show()
```



1.Set Figure Dimensions:

•The plt.figure(figsize=(12, 6)) sets the size of the plot to be 12x6 inches.

2.Plot Training MAE:

•The training Mean Absolute Error (MAE) values over epochs are plotted using plt.plot(history.history['mae'], label='Training MAE’).

3.Plot Validation MAE:

•The validation MAE values over epochs are plotted with plt.plot(history.history['val_mae'], label='Validation MAE’).

4.Add Labels and Title:

•X-axis is labeled "Epochs," and Y-axis is labeled "Mean Absolute Error." The plot is titled "Model MAE Over Epochs."

5.Add Legend:

•The plt.legend() adds a legend to differentiate between Training MAE and Validation MAE.

6.Display the Plot:

•The plt.show() command displays the final graph.

7.Interpretation of the Graph:

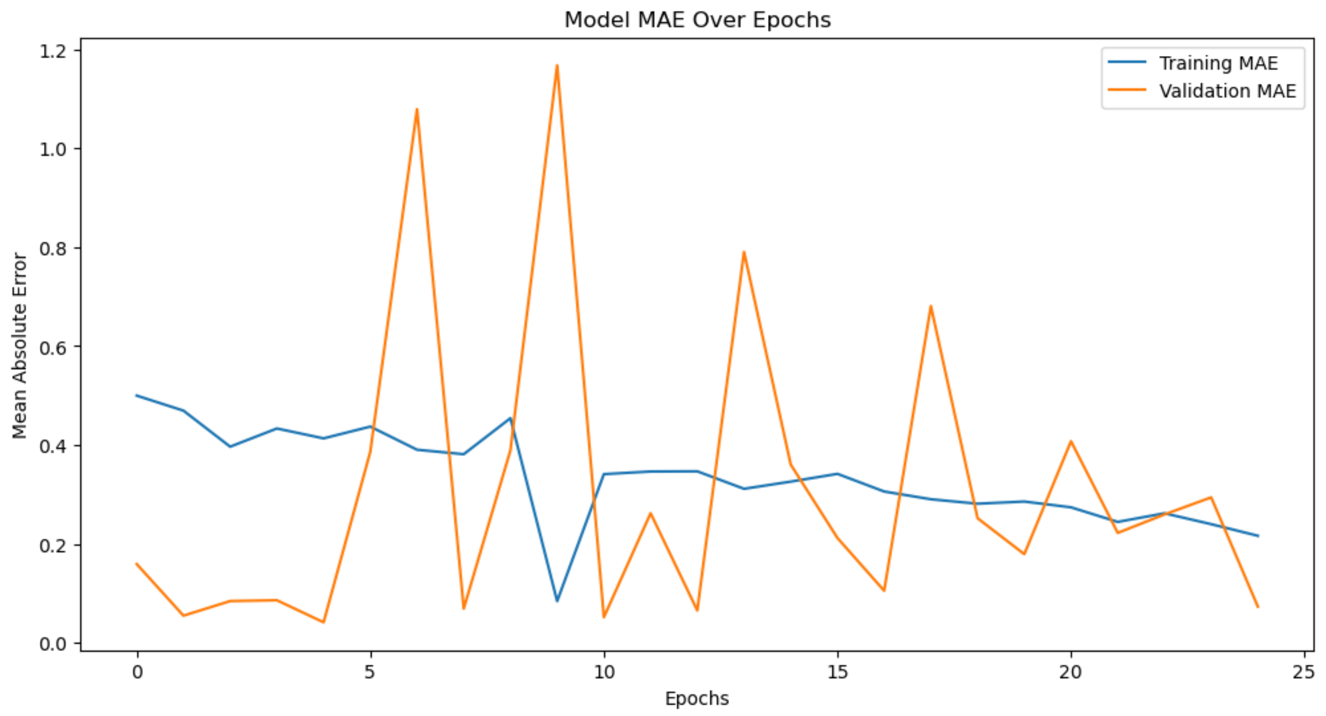
•The graph shows how the training and validation MAE change over training epochs.

•Training MAE is represented by the blue line, and Validation MAE is represented by the orange line.

•Fluctuations in validation MAE may indicate overfitting or instability in training.

This outline should work well for your presentation!

```
plt.figure(figsize=(12, 6))
plt.plot(history.history['mae'], label='Training MAE')
plt.plot(history.history['val_mae'], label='Validation MAE')
plt.xlabel('Epochs')
plt.ylabel('Mean Absolute Error')
plt.title('Model MAE Over Epochs')
plt.legend()
plt.show()
```



1.Generate Predictions:

- y_pred = model.predict(X_val) calculates predictions from the model using the validation data (X_val).

2.Set Figure Dimensions:

- The figure size is set to 10x6 inches using plt.figure(figsize=(10, 6)).

3.Scatter Plot:

- A scatter plot is created using plt.scatter(y_val, y_pred, color='blue', alpha=0.5) to compare actual values (y_val) and predicted values (y_pred).
- alpha=0.5 makes the points slightly transparent for better visualization.

4.Plot Reference Line:

- A red dashed line is added using plt.plot([min(y_val), max(y_val)], [min(y_val), max(y_val)], color='red', linestyle='--') to represent the ideal case where predictions perfectly match actual values.

5.Add Labels and Title:

- The X-axis is labeled "Actual Values," and the Y-axis is labeled "Predicted Values."
- The plot is titled "Actual vs Predicted Values."

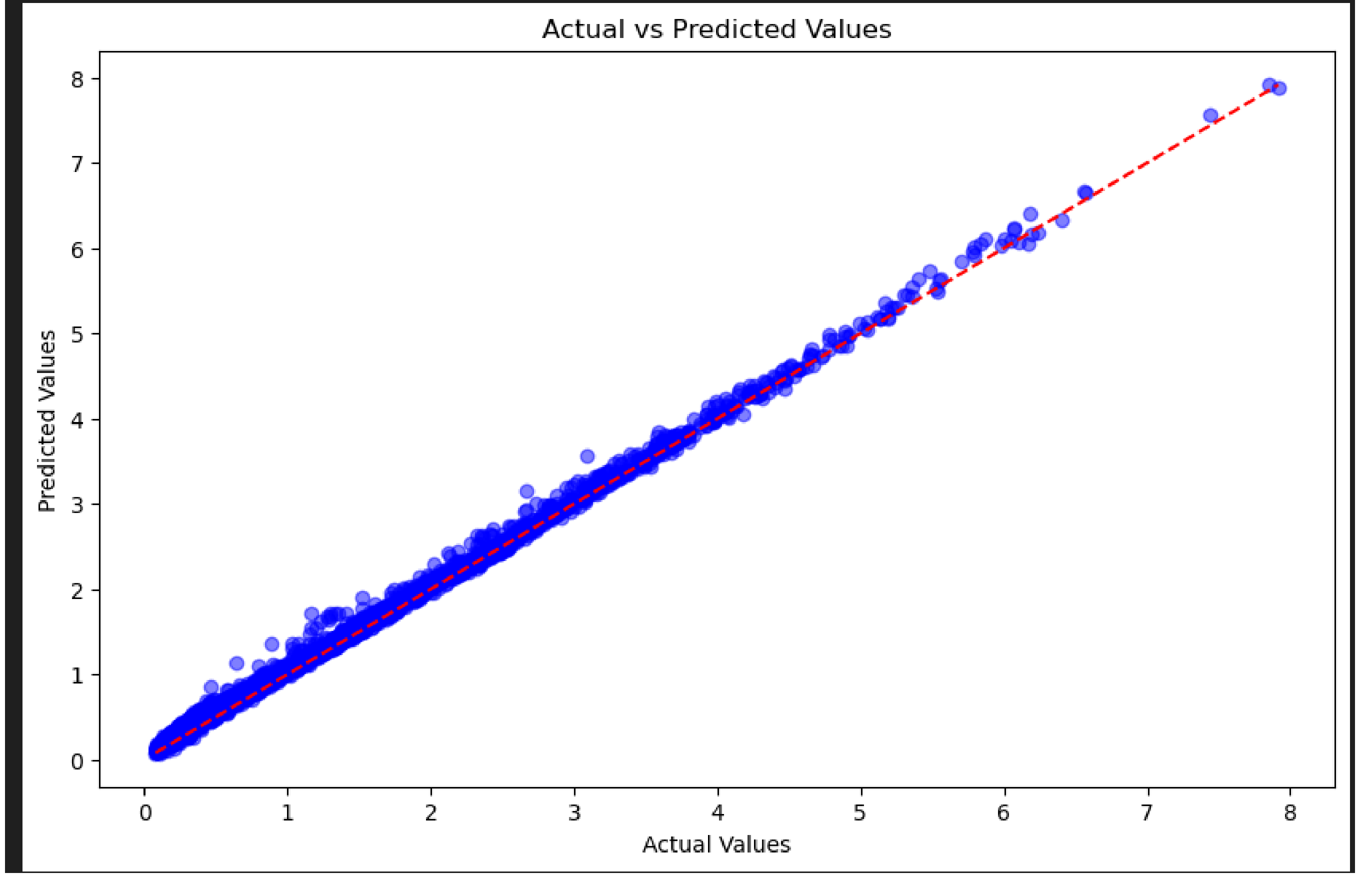
6.Display the Plot:

- The plt.show() command displays the final visualization.

7.Interpretation of the Graph:

- The blue points represent the relationship between actual and predicted values.
- The closer the points are to the red dashed line, the better the model's predictions.
- Significant deviations from the red line indicate prediction errors.

```python
# Real vs Prediction Graph
plt.figure(figsize=(10, 6))
plt.scatter(y_val, y_pred, color='blue', alpha=0.5)
plt.plot([min(y_val), max(y_val)], [min(y_val), max(y_val)], color='red', linestyle='--')
plt.xlabel("Actual Values")
plt.ylabel("Predicted Values")
plt.title("Actual vs Predicted Values")
plt.show()
```



1.Calculate Residuals:

- •residuals = y_val - y_pred.flatten() computes the difference between actual values (y_val) and predicted values (y_pred).

2.Set Figure Dimensions:

- •The figure size is set to 10x6 inches using plt.figure(figsize=(10, 6)).

3.Plot Residuals Distribution:

- •A histogram is plotted using sns.histplot(residuals, kde=True, color='purple', bins=30):
  - •**Histogram**: Shows the frequency distribution of residuals.
  - •**Kernel Density Estimate (KDE)**: The smooth curve over the histogram represents the probability density function.

4.Add Labels and Title:

- •The X-axis is labeled "Residuals," and the Y-axis is labeled "Frequency."
- •The plot is titled "Residuals Distribution."

5.Display the Plot:

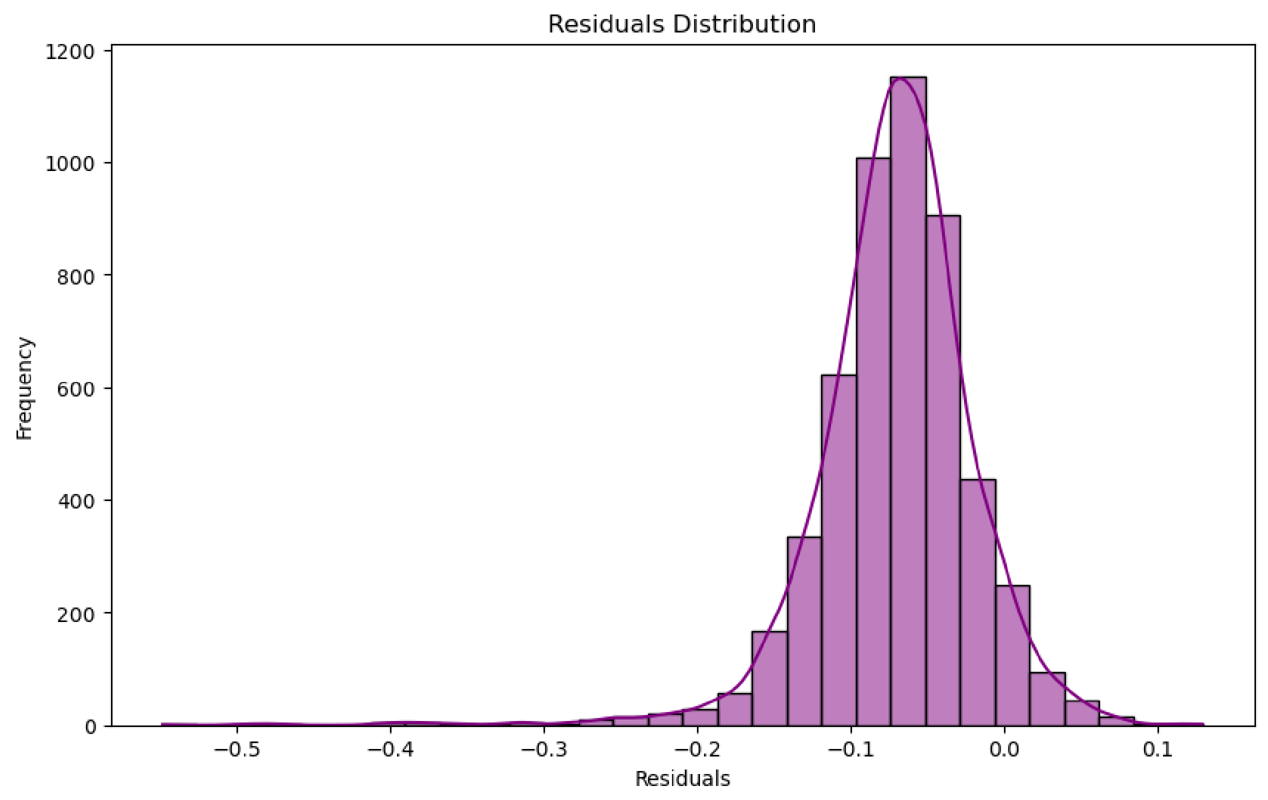
- •The plt.show() command displays the final visualization.

6.Interpretation of the Graph:

- •**Symmetry**: The residuals appear symmetrically distributed around zero, indicating no significant bias in predictions.
- •**Narrow Spread**: Most residuals are close to zero, implying that the model has small errors.
- •**Outliers**: Any values far from zero might represent outliers.

```python
# Residuals Analysis
residuals = y_val - y_pred.flatten()

plt.figure(figsize=(10, 6))
sns.histplot(residuals, kde=True, color='purple', bins=30)
plt.title("Residuals Distribution")
plt.xlabel("Residuals")
plt.ylabel("Frequency")
plt.show()
```



1. **Define Daytime and Nighttime Periods**:
   - Assigns each hour of the day to either "Day" (09:00–18:00) or "Night" (18:00–09:00).
2. **Calculate Average Consumption**:
   - Computes the hourly average power consumption separately for daytime and nighttime.
3. **Determine Thresholds**:
   - Calculates the 75th percentile (upper quartile) thresholds for daytime and nighttime power consumption.
4. **Identify High-Consumption Hours**:
   - Identifies hours where the power consumption exceeds the 75th percentile threshold for both daytime and nighttime.
5. **Generate Warning Messages**:
   - For each period (daytime and nighttime), prints a warning if high-consumption hours are detected, along with the hour and consumption value. Otherwise, prints a congratulatory message if no hours exceed the threshold.

```python
# Define daytime (09:00 - 18:00) and nighttime (18:00 - 09:00) hours
hourly_data['period'] = hourly_data['hour'].apply(lambda x: 'Day' if 9 <= x <= 18 else 'Night')
```

```python
# Calculate hourly average consumption for daytime and nighttime
day_consumption = hourly_data[hourly_data['period'] == 'Day'].groupby('hour')['Global_active_power'].mean()
night_consumption = hourly_data[hourly_data['period'] == 'Night'].groupby('hour')['Global_active_power'].mean()
```

```python
# Calculate the 75th percentile thresholds for daytime and nighttime
day_threshold = day_consumption.quantile(0.75)
night_threshold = night_consumption.quantile(0.75)
```

```python
# Identify the hours where consumption exceeds the threshold for daytime and nighttime
day_high_usage = day_consumption[day_consumption > day_threshold]
night_high_usage = night_consumption[night_consumption > night_threshold]
```

```python
# Warning messages
print("=== Daytime (09:00 - 18:00) ===")
if not day_high_usage.empty:
    print("Warning! High consumption hours during daytime:")
    for hour, consumption in day_high_usage.items():
        print(f"  - Hour {hour}: {consumption:.2f} kW (exceeds threshold!)")
else:
    print("Congratulations! No hours exceed the threshold during daytime.")

print("\n=== Nighttime (18:00 - 09:00) ===")
if not night_high_usage.empty:
    print("Warning! High consumption hours during nighttime:")
    for hour, consumption in night_high_usage.items():
        print(f"  - Hour {hour}: {consumption:.2f} kW (exceeds threshold!)")
else:
    print("Congratulations! No hours exceed the threshold during nighttime.")
```

1.Setup for Plotting:

- Creates a figure with two subplots to display the daytime and nighttime consumption graphs side by side.

2.Daytime Plot:

- Plots the average hourly consumption during the day.
- Adds a horizontal red dashed line indicating the daytime threshold.
- Highlights high-consumption hours with orange markers.

3.Nighttime Plot:

- Similar to the daytime plot, but for nighttime consumption.
- Plots the nighttime average hourly consumption, threshold, and high-consumption hours.

4.Graph Formatting:

- Titles, axes labels, legends, and grid lines are added for clarity.
- plt.tight_layout() ensures proper spacing between plots.
- plt.show() displays the graphs.

5.Left Graph (Daytime):

- Blue line: Shows the average consumption during daytime hours.
- Red dashed line: Indicates the daytime consumption threshold (75th percentile).
- Orange dots: Highlight the high-consumption hours exceeding the threshold.

6.Right Graph (Nighttime):

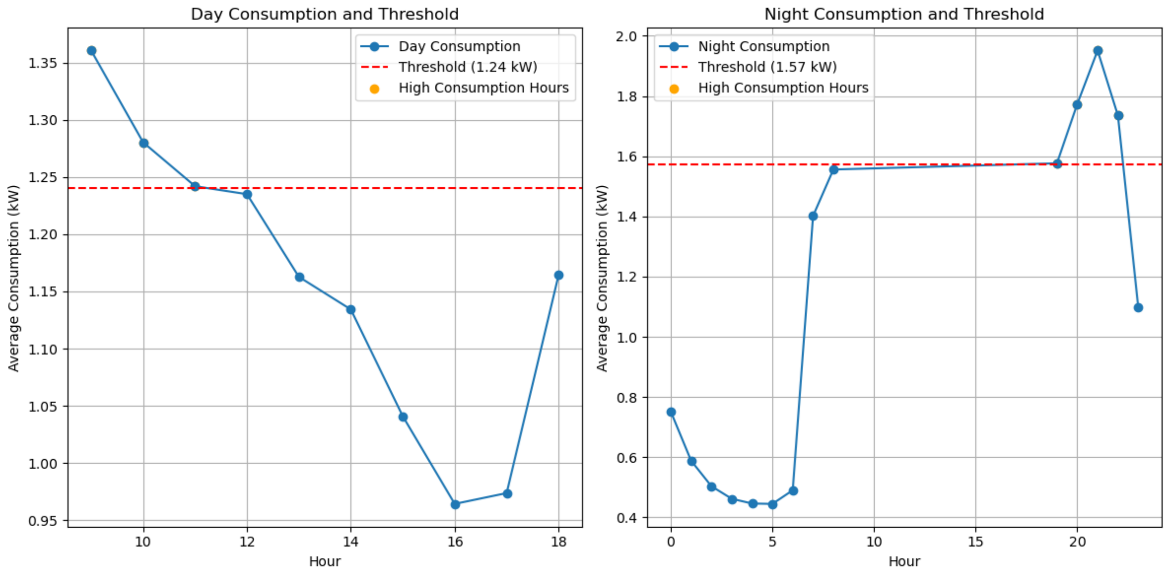
- Blue line: Shows the average consumption during nighttime hours.
- Red dashed line: Indicates the nighttime consumption threshold (75th percentile).
- Orange dots: Highlight the high-consumption hours exceeding the threshold.

```python
# Plotting the daytime and nighttime graphs
plt.figure(figsize=(12, 6))

# Daytime plot
plt.subplot(1, 2, 1)
plt.plot(day_consumption.index, day_consumption.values, marker='o', label='Day Consumption')
plt.axhline(y=day_threshold, color='red', linestyle='--', label=f'Threshold ({day_threshold:.2f} kW)')
plt.scatter(day_high_usage.index, day_high_usage.values, color='orange', label='High Consumption Hours')
plt.title('Day Consumption and Threshold')
plt.xlabel('Hour')
plt.ylabel('Average Consumption (kW)')
plt.legend()
plt.grid()

# Nighttime plot
plt.subplot(1, 2, 2)
plt.plot(night_consumption.index, night_consumption.values, marker='o', label='Night Consumption')
plt.axhline(y=night_threshold, color='red', linestyle='--', label=f'Threshold ({night_threshold:.2f} kW)')
plt.scatter(night_high_usage.index, night_high_usage.values, color='orange', label='High Consumption Hours')
plt.title('Night Consumption and Threshold')
plt.xlabel('Hour')
plt.ylabel('Average Consumption (kW)')
plt.legend()
plt.grid()

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



1.Monthly Average Consumption:

- Calculates the average power consumption for each month.

2.Threshold for Monthly Consumption:

- Determines the 75th percentile threshold for monthly consumption
.

3.Identify High-Consumption Months:

- Finds months where the average consumption exceeds the threshold.

4.Generate Warning Messages:

- Lists months exceeding the threshold with their consumption values.
- If no month exceeds the threshold, displays a congratulatory message.

5.Summer and Winter Analysis:

- Separates summer months (June, July, August) and winter months (December, January, February).
- Calculates and prints the average consumption for summer and winter months.

```python
# Calculate the average consumption for each month
monthly_consumption = hourly_data.groupby('month')['Global_active_power'].mean()
```

```python
# Threshold for monthly consumption (75th percentile)
monthly_threshold = monthly_consumption.quantile(0.75)
```

```python
# Identify the months where consumption exceeds the threshold
high_usage_months = monthly_consumption[monthly_consumption > monthly_threshold]
```

```python
# Warning messages
print("=== Monthly Analysis ===")
if not high_usage_months.empty:
    print("Warning! Months exceeding the threshold:")
    for month, consumption in high_usage_months.items():
        print(f"  - Month {month}: {consumption:.2f} kW (exceeds threshold!)")
else:
    print("Congratulations! No months exceed the threshold.")
```

```
=== Monthly Analysis ===
Warning! Months exceeding the threshold:
  - Month 1: 1.46 kW (exceeds threshold!)
  - Month 11: 1.30 kW (exceeds threshold!)
  - Month 12: 1.51 kW (exceeds threshold!)
```

```python
# Summer and Winter months analysis
summer_months = hourly_data[hourly_data['month'].isin([6, 7, 8])]
winter_months = hourly_data[hourly_data['month'].isin([12, 1, 2])]

summer_consumption = summer_months['Global_active_power'].mean()
winter_consumption = winter_months['Global_active_power'].mean()

print("Average Consumption in Summer Months:", summer_consumption)
print("Average Consumption in Winter Months:", winter_consumption)
```

```
Average Consumption in Summer Months: 0.732735656372411
Average Consumption in Winter Months: 1.421891677775047
```

1.Customize X-Axis:

- The x-axis tick labels are set to display month names with a 45° rotation for clarity.

2.Bar Plot of Monthly Average Consumption:

- A bar chart (plt.bar) is used to display monthly average consumption data.
- Bars are colored "skyblue" and labeled accordingly.

3.Draw Threshold Line:

- A horizontal red dashed line (plt.axhline) represents the consumption threshold.
- The threshold value is dynamically labeled.

4.Highlight High Consumption Months:

- High consumption months are highlighted with orange bars to differentiate them from regular months.

5.Customize X-Axis:

- The x-axis tick labels are set to display month names with a 45° rotation for clarity.

6.Add Title and Labels:

- A title, x-axis, and y-axis labels are added to describe the chart.

7.Legend for Clarity:

- A legend is included to explain the plot components: threshold, average consumption, and high consumption.

8.Add Gridlines:

- Horizontal gridlines are added to enhance readability.

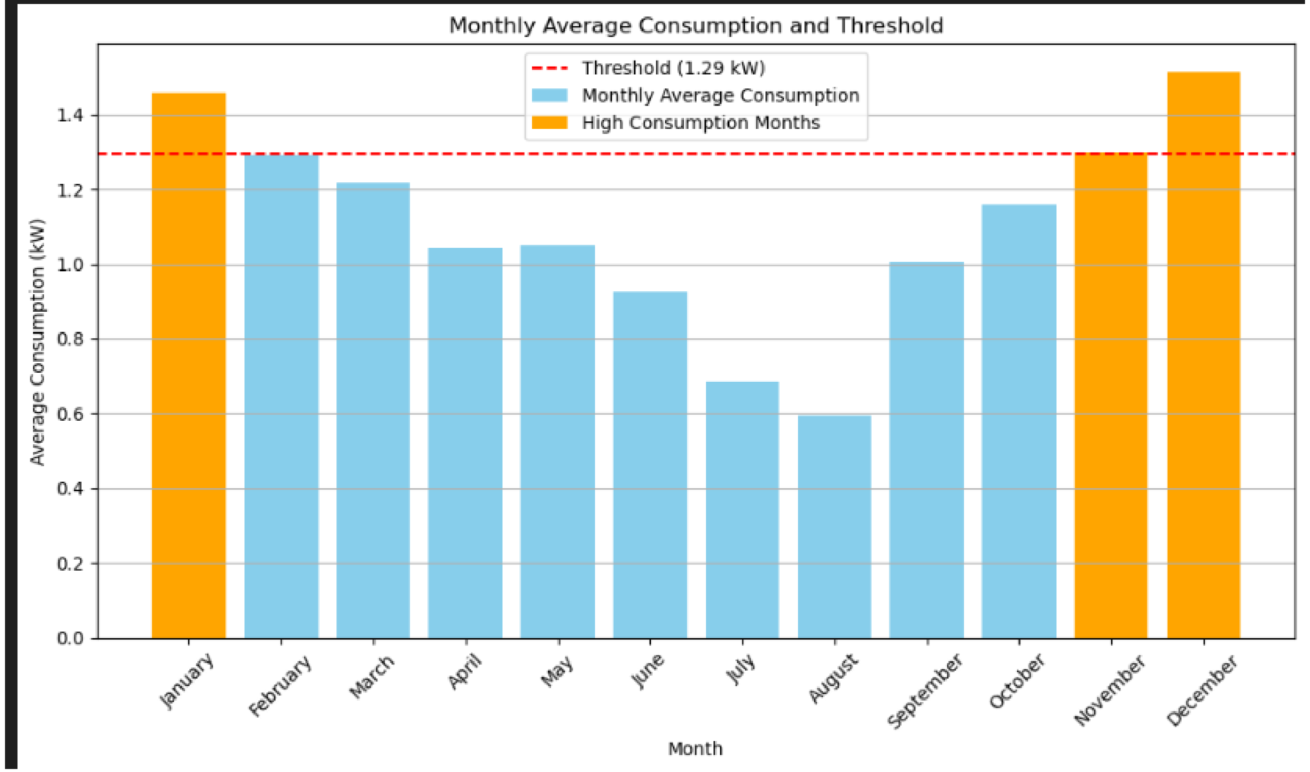
9.Tight Layout:

- The layout is adjusted (plt.tight_layout) to prevent overlapping of elements.

10.Display the Plot:

- The final plot is shown using plt.show().

```python
# Create a plot
plt.figure(figsize=(10, 6))
plt.bar(monthly_consumption.index, monthly_consumption.values, color='skyblue', label='Monthly Average Consumption')
plt.axhline(y=monthly_threshold, color='red', linestyle='--', label=f'Threshold ({monthly_threshold:.2f} kW)')
plt.bar(high_usage_months.index, high_usage_months.values, color='orange', label='High Consumption Months')
plt.xticks(monthly_consumption.index, ['January', 'February', 'March', 'April', 'May', 'June',
                                       'July', 'August', 'September', 'October', 'November', 'December'], rotation=45)
plt.title('Monthly Average Consumption and Threshold')
plt.xlabel('Month')
plt.ylabel('Average Consumption (kW)')
plt.legend()
plt.grid(axis='y')
plt.tight_layout()
plt.show()
```



1.Calculate Average Consumption by Day:

- Data is grouped by weekdays using hourly_data.groupby('weekday').
- The mean of the Global_active_power column is calculated to get daily average consumption.

2.Define Days of the Week:

- A list of days (Monday to Sunday) is explicitly defined to ensure correct order on the x-axis.

3.Set Figure Size:

- A plot with dimensions of 10x6 inches is created for better visualization.

4.Create Bar Chart:

- A bar chart (plt.bar) is used to display average consumption for each day of the week.
- Bars are colored green for visual distinction.

5.Add Title and Labels:

- The plot includes a title describing the data (Average Consumption by Day of the Week).
- X-axis and Y-axis are labeled to indicate days and average consumption (kW), respectively.

6.Display the Plot:

- The final plot is displayed using plt.show().

```python
# Daily total consumption
weekday_consumption = hourly_data.groupby('weekday')['Global_active_power'].mean()

days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
plt.figure(figsize=(10, 6))
plt.bar(days, weekday_consumption.values, color='green')
plt.title('Average Consumption by Day of the Week')
plt.xlabel('Day')
plt.ylabel('Average Consumption (kW)')
plt.show()
```