Steel_energy_consumption (4)

August 1, 2024

1 Steel Energy Consumption Data Analysis and predictive maintenance

1.0.1 This project aims to analyze the steel energy consumption data from the STEEL_ENERGY database. Using Python and MySQL, we will extract, analyze, and visualize the data to gain insights into energy usage patterns and identify potential areas for optimization.

2 1.0 Connecting with the database

[1]:
```
import mysql.connector
import pandas as pd

# Database connection details
host = 'localhost'
user = 'root'
password = '3221'
database = 'STEEL_ENERGY'

# Establish the connection
connection = mysql.connector.connect(
    host=host,
    user=user,
    password=password,
    database=database
)

# Load the entire dataset into a DataFrame
query = "SELECT * FROM Steel_Industry_Data"
df = pd.read_sql(query, connection)

# Close the connection
connection.close()
```

```
C:\Users\JAS\AppData\Local\Temp\ipykernel_10836\1275379568.py:20: UserWarning:
pandas only supports SQLAlchemy connectable (engine/connection) or database
string URI or sqlite3 DBAPI2 connection. Other DBAPI2 objects are not tested.
```

```
Please consider using SQLAlchemy.
  df = pd.read_sql(query, connection)
```

2.0.1 1.1 Exploring the Data

[2]:
```python
import pandas as pd
from IPython.display import display

# Assuming df is the DataFrame with the entire dataset

# Analyze data types of all columns
print("Data Types:")
display(df.dtypes)

# Check for missing values
print("\nMissing Values:")
display(df.isnull().sum())

# Identify duplicate values
print("\nDuplicate Values:")
duplicates = df.duplicated()
print(f"Number of duplicate rows: {duplicates.sum()}")

# Summary statistics for numerical columns
print("\nSummary Statistics:")
display(df.describe())

# Value counts for categorical columns
print("\nValue Counts for Categorical Columns:")
categorical_columns = df.select_dtypes(include=['object']).columns
for column in categorical_columns:
    print(f"\nValue counts for {column}:")
    display(df[column].value_counts())
```

```
Data Types:
```

```
Usage_kWh                                 float64
Lagging_Current_Reactive.Power_kVarh      float64
Leading_Current_Reactive_Power_kVarh        int64
CO2(tCO2)                                   int64
Lagging_Current_Power_Factor              float64
Leading_Current_Power_Factor                int64
NSM                                         int64
WeekStatus                                 object
Day_of_week                                object
Load_Type                                  object
date                                 datetime64[ns]
dtype: object
```

```
Missing Values:

Usage_kWh                             0
Lagging_Current_Reactive.Power_kVarh  0
Leading_Current_Reactive_Power_kVarh  0
CO2(tCO2)                             0
Lagging_Current_Power_Factor          0
Leading_Current_Power_Factor          0
NSM                                   0
WeekStatus                            0
Day_of_week                           0
Load_Type                             0
date                                  0
dtype: int64


Duplicate Values:
Number of duplicate rows: 0

Summary Statistics:

          Usage_kWh  Lagging_Current_Reactive.Power_kVarh  \
count  35040.000000                          35040.000000
mean      27.386892                             13.035384
min        0.000000                              0.000000
25%        3.200000                              2.300000
50%        4.570000                              5.000000
75%       51.237500                             22.640000
max      157.180000                             96.910000
std       33.444380                             16.306000

       Leading_Current_Reactive_Power_kVarh  CO2(tCO2)  \
count                          35040.000000    35040.0
mean                               3.869092        0.0
min                                0.000000        0.0
25%                                0.000000        0.0
50%                                0.000000        0.0
75%                                2.000000        0.0
max                               28.000000        0.0
std                                7.436078        0.0

       Lagging_Current_Power_Factor  Leading_Current_Power_Factor  \
count                  35040.000000                  35040.000000
mean                      80.578056                     84.374572
min                        0.000000                      0.000000
25%                       63.320000                    100.000000
50%                       87.960000                    100.000000
75%                       99.022500                    100.000000
```

```
max                      100.000000                    100.000000
std                       18.921322                     30.457117

                NSM                 date
count  35040.000000                35040
mean   42750.000000  2018-07-02 11:52:30
min        0.000000  2018-01-01 00:00:00
25%    21375.000000  2018-04-02 05:56:15
50%    42750.000000  2018-07-02 11:52:30
75%    64125.000000  2018-10-01 17:48:45
max    85500.000000  2018-12-31 23:45:00
std    24940.534317                  NaN

Value Counts for Categorical Columns:

Value counts for WeekStatus:

WeekStatus
Weekday    25056
Weekend     9984
Name: count, dtype: int64


Value counts for Day_of_week:

Day_of_week
Monday       5088
Tuesday      4992
Wednesday    4992
Thursday     4992
Friday       4992
Saturday     4992
Sunday       4992
Name: count, dtype: int64


Value counts for Load_Type:

Load_Type
Light_Load      18072
Medium_Load      9696
Maximum_Load     7272
Name: count, dtype: int64
```

2.0.2 Note : In the summary statistics, the 25th, 50th, and 75th percentiles represent:

- **25th Percentile**: The point where 25% of the values are lower.
- **50th Percentile (Median)**: The middle value where half of the values are lower and half are higher.
- **75th Percentile**: The point where 75% of the values are lower.

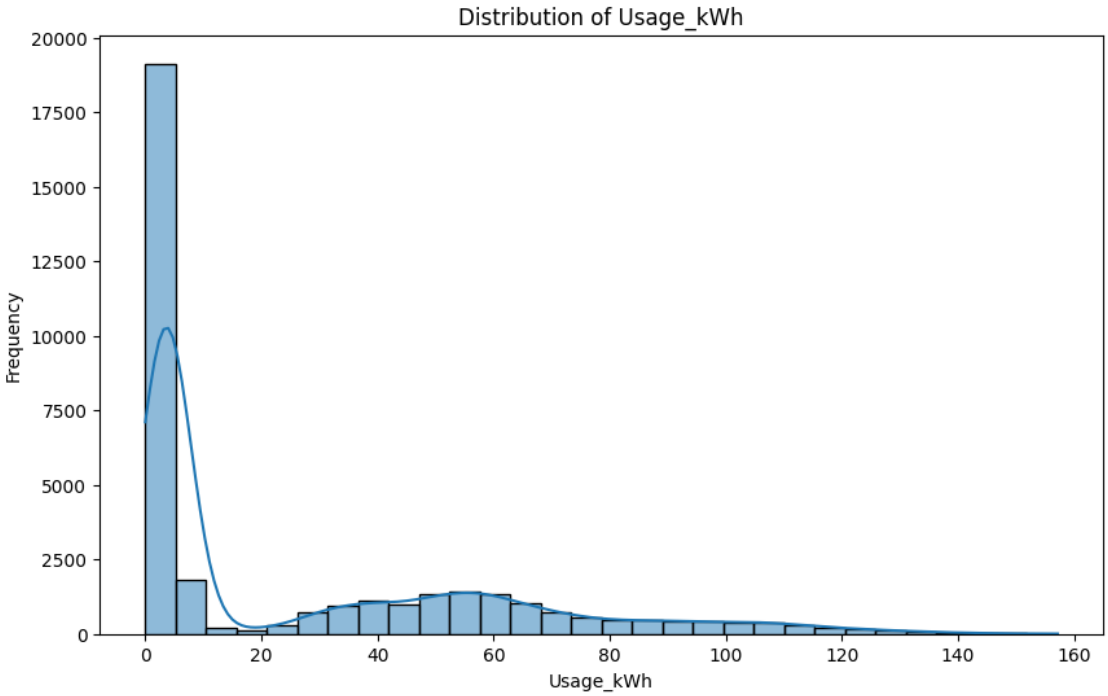
2.0.3 1.2 Visualising the data

[3]:
```
import matplotlib.pyplot as plt
import seaborn as sns
```

2.0.4 1.2.a Distribution of Usage_kWh

This histogram shows the distribution of `Usage_kWh` values in the dataset. The KDE (Kernel Density Estimate) line provides a smoothed estimate of the distribution.

[4]:
```
plt.figure(figsize=(10, 6))
sns.histplot(df['Usage_kWh'], bins=30, kde=True)
plt.title('Distribution of Usage_kWh')
plt.xlabel('Usage_kWh')
plt.ylabel('Frequency')
plt.show()
```



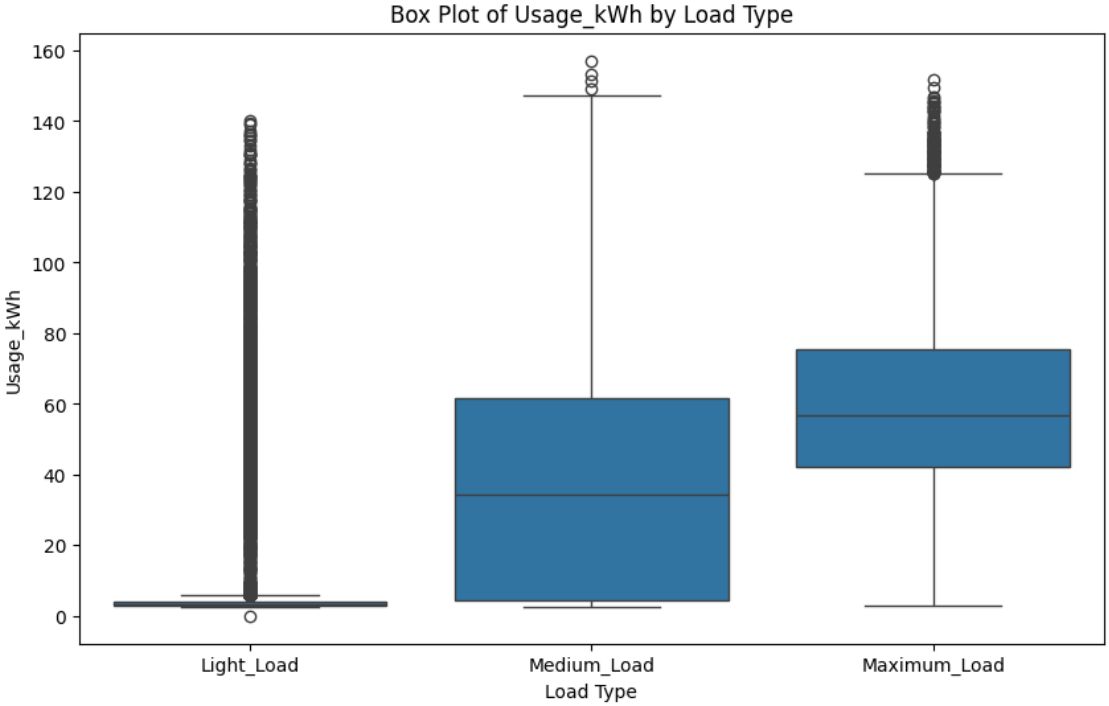
2.0.5 Conclusion

The histogram above shows the distribution of electricity usage (`Usage_kWh`) in the dataset.

- **Key Observations**:
  - **Most Frequent Usage**: The vast majority of the time, the electricity usage is very low, with many readings clustering around the lower end of the scale.
  - **Rare High Usage**: There are occasional instances where the electricity usage spikes significantly, but these are much less common.

Overall, the data indicates that low electricity usage is the norm, while high usage is relatively rare. This pattern suggests that the system typically operates under low power conditions, with occasional periods of higher demand.

[5]:
```
plt.figure(figsize=(10, 6))
sns.boxplot(x='Load_Type', y='Usage_kWh', data=df)
plt.title('Box Plot of Usage_kWh by Load Type')
plt.xlabel('Load Type')
plt.ylabel('Usage_kWh')
plt.show()
```



2.0.6 Conclusion

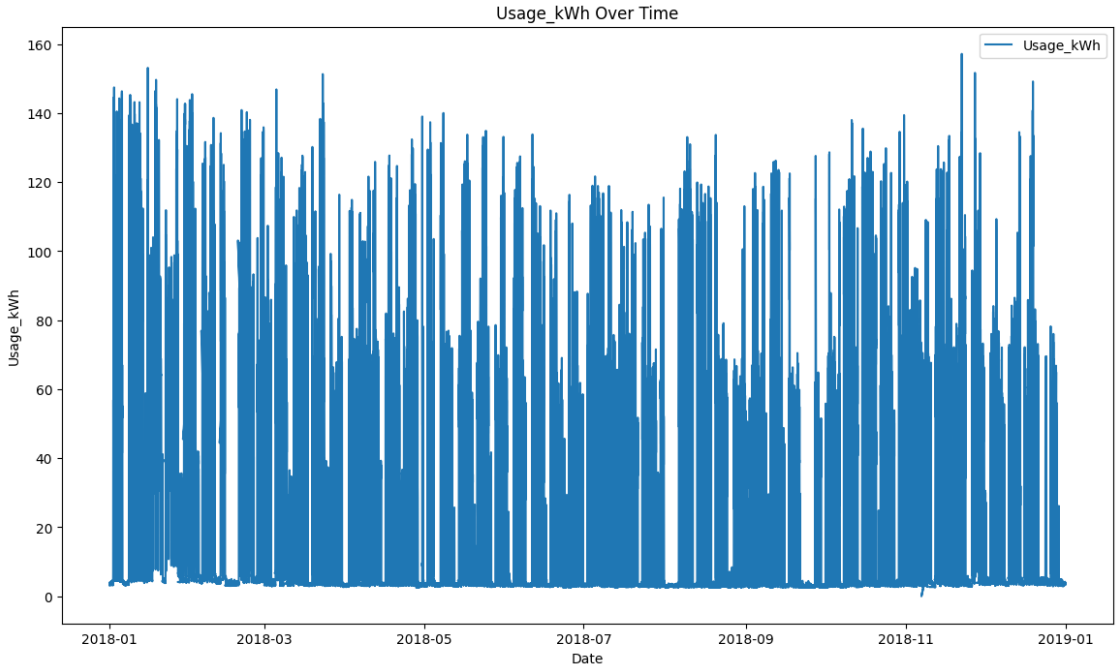
The box plot above shows the distribution of electricity usage (`Usage_kWh`) for different load types: Light_Load, Medium_Load, and Maximum_Load.

- **Key Observations**:
  - **Light_Load**:
    - The majority of the usage values are low, with most values clustered around the lower end.
    - There are many outliers, indicating occasional higher usage even under light load conditions.
  - **Medium_Load**:
    - The usage values are higher compared to Light_Load.

    - The range of usage values is wider, indicating more variability in electricity usage under medium load conditions.
  - **Maximum_Load**:
    - The usage values are the highest among the three load types.
    - There are fewer outliers compared to Light_Load, suggesting that high usage values are more consistent under maximum load conditions.

Overall, the data indicates that as the load type increases from Light_Load to Maximum_Load, the electricity usage generally increases and becomes more consistent, especially under maximum load conditions. Light load conditions show a lot of variability with many low values and some occasional high outliers.

[6]:
```
plt.figure(figsize=(14, 8))
plt.plot(df['date'], df['Usage_kWh'], label='Usage_kWh')
plt.title('Usage_kWh Over Time')
plt.xlabel('Date')
plt.ylabel('Usage_kWh')
plt.legend()
plt.show()
```



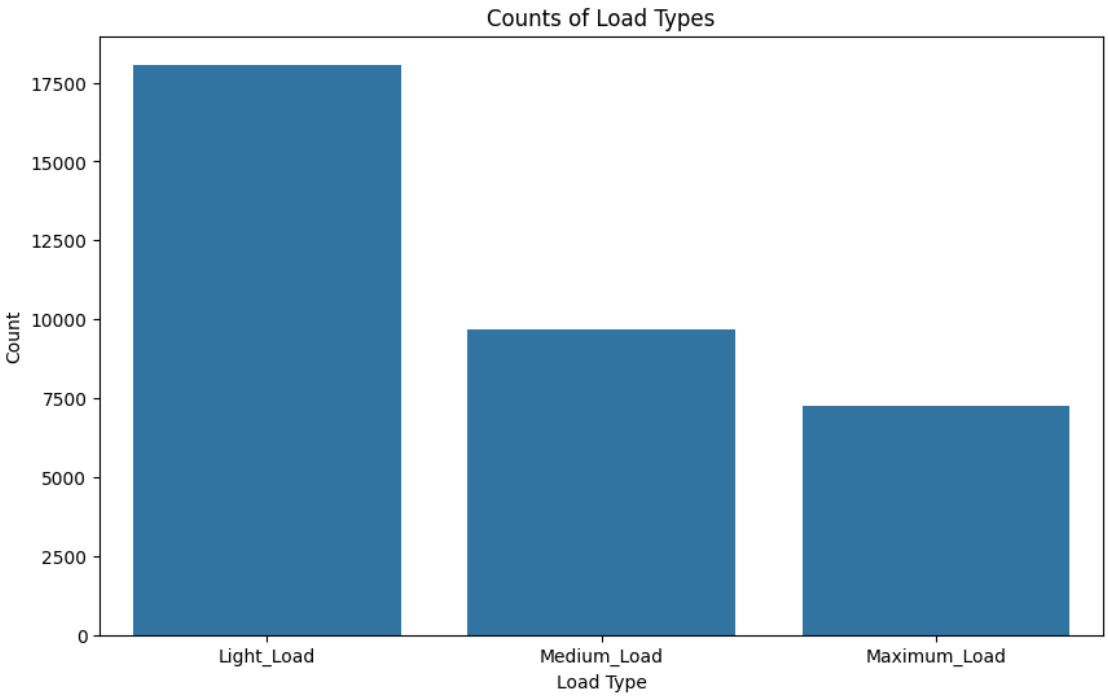
2.0.7 Key Observations:

1. **High Variability**: Energy consumption shows significant fluctuations, ranging from near zero to around 150 kWh.
2. **Regular Patterns**: There are consistent periods of high and low energy usage, indicating operational cycles within the industry.

2.0.8 Conclusion:

Energy usage in the steel industry is highly dynamic with clear operational cycles. Understanding these patterns can help optimize energy management and reduce costs.

[7]:
```
plt.figure(figsize=(10, 6))
sns.countplot(x='Load_Type', data=df)
plt.title('Counts of Load Types')
plt.xlabel('Load Type')
plt.ylabel('Count')
plt.show()
```



2.0.9 Load Type Distribution

Key Points:

1. **Predominance of Light Load**: The majority of the data points correspond to the `Light_Load` category, indicating that the system predominantly operates under light load conditions.
2. **Less Frequent Maximum Load**: The `Maximum_Load` category is the least frequent, suggesting that the system rarely operates at its highest capacity.

2.0.10 Conclusion:

The steel industry system mainly functions under light load conditions, with fewer instances of medium and maximum loads. This distribution highlights the need for optimization strategies

focused on light load operations to enhance overall efficiency.

[8]:
```
import pandas as pd

# Ensure the date column is in datetime format
df['date'] = pd.to_datetime(df['date'])

# Extract the month from the date column
df['month'] = df['date'].dt.month

# Count the occurrences of each month
month_counts = df['month'].value_counts().sort_index()

# Display the month counts
print(month_counts)
```

```
month
1     2976
2     2688
3     2976
4     2880
5     2976
6     2880
7     2976
8     2976
9     2880
10    2976
11    2880
12    2976
Name: count, dtype: int64
```

2.1 Understanding the Correlation Matrix

2.1.1 What is a Correlation Matrix?

A correlation matrix is a table showing correlation coefficients between variables. Each cell in the table shows the correlation between two variables. The value is between -1 and 1, indicating the strength and direction of the relationship.

2.1.2 How to Read a Correlation Matrix:

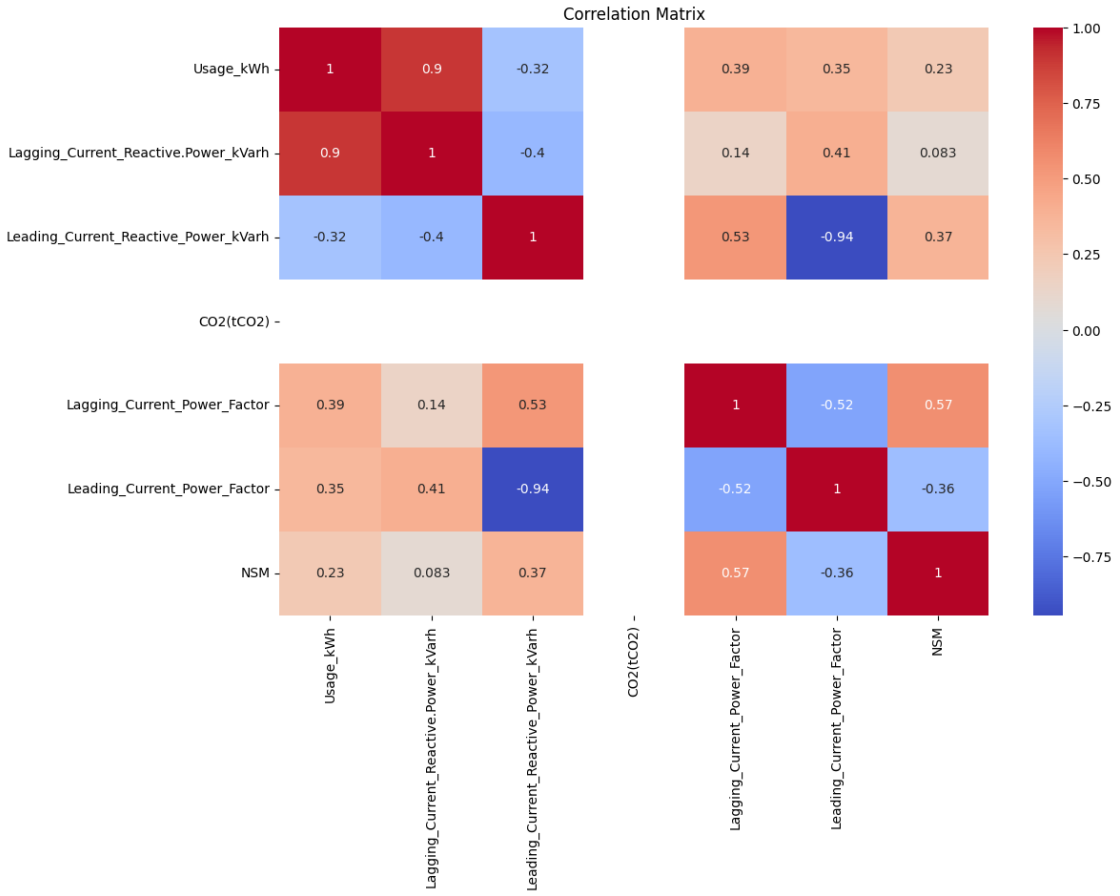
- **Correlation Coefficient Values**:
  - **1**: Perfect positive correlation (variables move in the same direction).
  - **-1**: Perfect negative correlation (variables move in opposite directions).
  - **0**: No correlation (no linear relationship between variables).
- **Interpreting Values**:
  - **Positive Correlation (0 to 1)**: As one variable increases, the other also increases.
  - **Negative Correlation (-1 to 0)**: As one variable increases, the other decreases.
  - **Close to 0**: Weak or no linear relationship.

2.1.3 Significance of a Correlation Matrix:

1. **Identify Relationships**: Helps to identify and quantify the strength of relationships between pairs of variables.
2. **Feature Selection**: In machine learning, it aids in selecting relevant features by showing which variables are highly correlated.
3. **Multicollinearity Detection**: Helps detect multicollinearity in regression models. High correlation between independent variables can affect model performance.
4. **Data Understanding**: Provides a quick overview of potential relationships and dependencies in the data, guiding further analysis and exploration.

Understanding the correlation matrix is essential for data analysis and modeling, as it provides insights into how variables interact with each other.

[9]:
```
import matplotlib.pyplot as plt
import seaborn as sns

# Select only the numeric columns for correlation matrix
numeric_df = df.select_dtypes(include=['float64', 'int64'])

# Compute the correlation matrix
corr_matrix = numeric_df.corr()

# Plot the heatmap
plt.figure(figsize=(12, 8))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm')
plt.title('Correlation Matrix')
plt.show()
```



2.1.4 Conclusion: Correlation Analysis of Steel Industry Energy Usage

Key Insights:

1. **High Positive Correlation**:
   - **Usage_kWh and Lagging_Current_Reactive.Power_kVarh (0.9)**:
     - This strong positive correlation indicates that as the reactive power increases, energy consumption also increases. This relationship suggests that managing reactive power effectively can significantly impact overall energy usage, as high reactive power indicates inefficiencies in the system that lead to increased energy consumption.
2. **Moderate Negative Correlation**:
   - **Usage_kWh and Leading_Current_Reactive_Power_kVarh (-0.32)**:
     - The moderate negative correlation suggests that higher leading reactive power tends to slightly reduce overall energy consumption. This could be due to efficient power factor correction mechanisms in place that help reduce the amount of energy consumed by minimizing the phase difference between voltage and current.
3. **Very High Positive Correlation**:
   - **Leading_Current_Power_Factor and Leading_Current_Reactive_Power_kVarh (0.94)**:
     - This very strong positive correlation indicates a direct and significant relationship

between these two metrics. As the leading reactive power increases, the leading power factor also increases, reflecting the close interaction between these components in the electrical system. Effective monitoring of these metrics can help maintain optimal power quality.

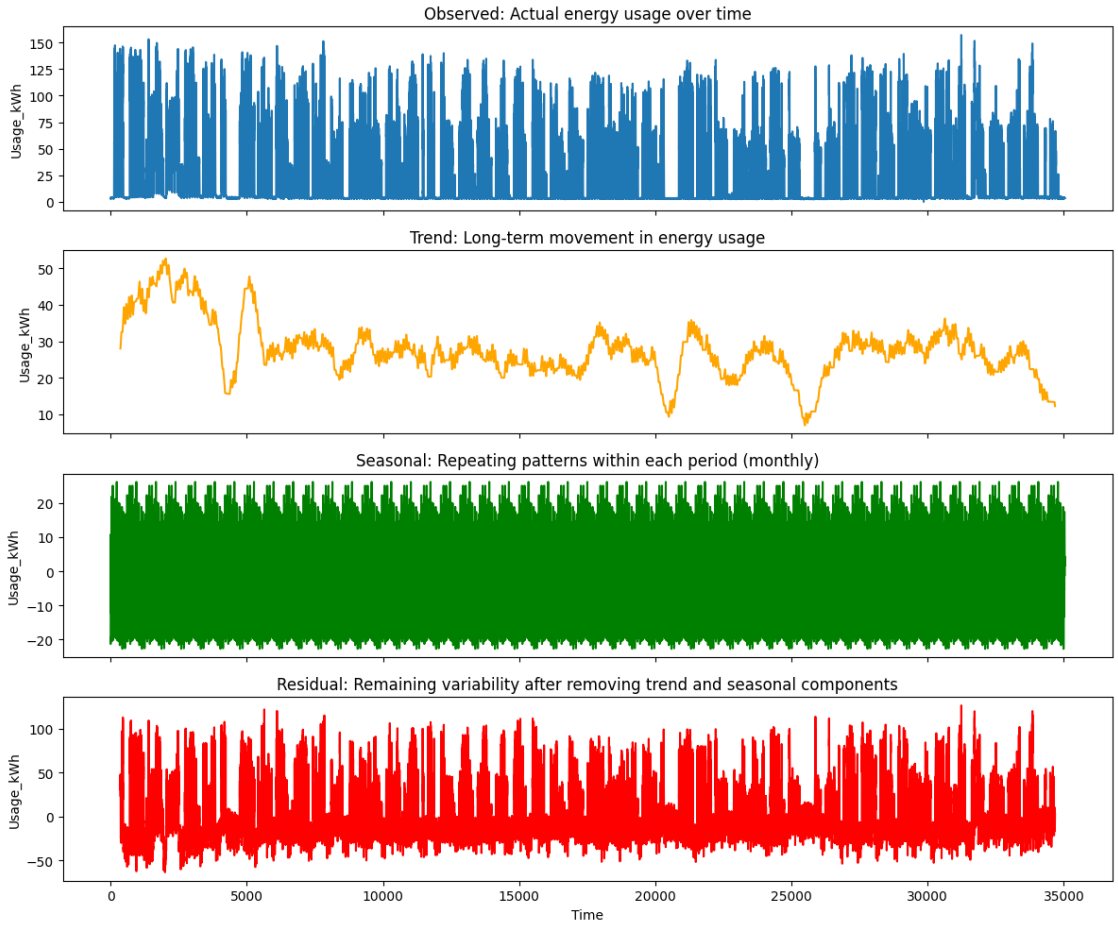
4. **Strong Negative Correlation**:
   - **Lagging_Current_Power_Factor and Leading_Current_Power_Factor (-0.94)**:
     - This strong inverse relationship suggests that when the lagging power factor increases, the leading power factor decreases significantly, and vice versa. This indicates a balance between lagging and leading reactive power in the system. Proper balancing is crucial for maintaining system efficiency and reducing energy wastage.

2.1.5 Implications for Industry Experts:

- **Reactive Power Management**: High correlations involving reactive power highlight its critical role in energy consumption. Effective management and correction strategies could lead to substantial energy savings by improving system efficiency and reducing unnecessary energy consumption.
- **Power Factor Balancing**: The inverse relationship between lagging and leading power factors underscores the importance of balancing these factors. Proper balancing can enhance system performance, reduce energy losses, and improve overall power quality.

[10]:
```
from statsmodels.tsa.seasonal import seasonal_decompose
import matplotlib.pyplot as plt

# Perform seasonal decomposition
result = seasonal_decompose(df['Usage_kWh'], model='additive', period=24*30)

# Plot the decomposition
fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, figsize=(12, 10), sharex=True)

# Observed
ax1.plot(result.observed, label='Observed')
ax1.set_ylabel('Usage_kWh')
ax1.set_title('Observed: Actual energy usage over time')

# Trend
ax2.plot(result.trend, label='Trend', color='orange')
ax2.set_ylabel('Usage_kWh')
ax2.set_title('Trend: Long-term movement in energy usage')

# Seasonal
ax3.plot(result.seasonal, label='Seasonal', color='green')
ax3.set_ylabel('Usage_kWh')
ax3.set_title('Seasonal: Repeating patterns within each period (monthly)')

# Residual
ax4.plot(result.resid, label='Residual', color='red')
```

```
ax4.set_ylabel('Usage_kWh')
ax4.set_title('Residual: Remaining variability after removing trend and␣
 ↪seasonal components')
ax4.set_xlabel('Time')

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



2.1.6 Observed: Actual Energy Usage Over Time

- **Description**: The top plot shows the raw `Usage_kWh` data over the observation period.
- **Conclusion**: Energy consumption fluctuates significantly, indicating a dynamic usage pattern with periods of both high and low energy use.

Trend: Long-Term Movement in Energy Usage

- **Description**: The second plot represents the overall direction or trend in energy usage, smoothing out short-term fluctuations.

- **Conclusion**: The trend reveals periods of gradual increase and decrease in energy consumption, with noticeable drops at certain points likely due to operational changes or external factors.

Seasonal: Repeating Patterns Within Each Period (Monthly)

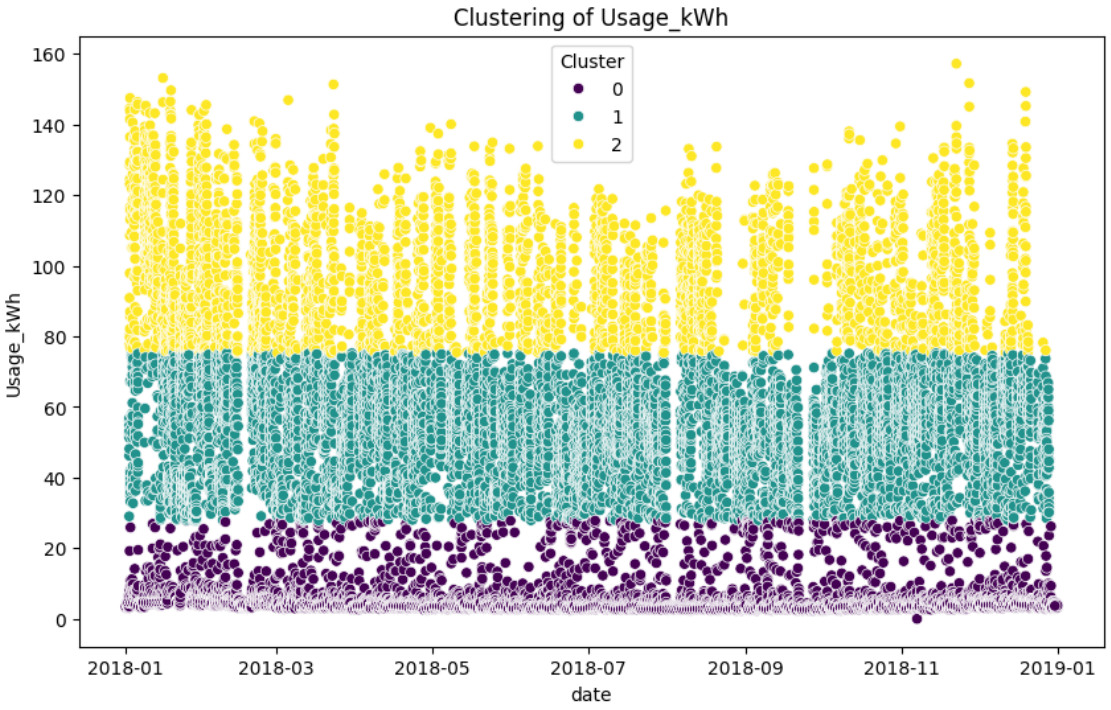
- **Description**: The third plot shows regular, repeating patterns that occur monthly.
- **Conclusion**: There is a consistent monthly cycle in energy usage, with higher usage at certain times of the month. This indicates predictable operational schedules or maintenance periods.

Residual: Remaining Variability After Removing Trend and Seasonal Components

- **Description**: The bottom plot displays the remaining variability after the trend and seasonal components are removed.
- **Conclusion**: High residuals highlight periods where actual usage deviates from expected patterns, suggesting anomalies or unexpected events affecting energy consumption.

Overall Conclusion: The decomposition reveals that energy usage in the steel industry has significant monthly patterns and long-term trends, with occasional anomalies. Understanding these components helps in optimizing energy management, improving efficiency, and reducing costs. Regular monitoring of these patterns can lead to more informed decisions and better operational strategies.

[11]:
```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=3)
df['Cluster'] = kmeans.fit_predict(df[['Usage_kWh']])
plt.figure(figsize=(10, 6))
sns.scatterplot(x='date', y='Usage_kWh', hue='Cluster', data=df,␣
↪palette='viridis')
plt.title('Clustering of Usage_kWh')
plt.show()
```



2.1.7 Clustering Analysis of Energy Usage

The graph above shows the clustering of `Usage_kWh` over time. Clustering helps to identify distinct patterns or groups within the energy consumption data.

Key Observations:

1. **Cluster 0 (Light Usage)**:
   - **Description**: This cluster (cyan points) represents the lowest range of energy usage, typically below 40 kWh.
   - **Conclusion**: These periods of low energy consumption are frequent and likely correspond to non-peak operational times or maintenance periods.
2. **Cluster 1 (Medium Usage)**:
   - **Description**: This cluster (purple points) represents moderate energy usage, generally between 40 kWh and 80 kWh.
   - **Conclusion**: These periods indicate regular operational activity where energy usage is stable and moderate.
3. **Cluster 2 (High Usage)**:
   - **Description**: This cluster (yellow points) includes the highest energy usage periods, typically above 80 kWh.
   - **Conclusion**: These high energy usage periods likely correspond to peak operational times, possibly during intensive production or other high-demand activities.

Overall Conclusion: The clustering analysis reveals three distinct patterns of energy consumption in the steel industry: light, medium, and high usage periods. Understanding these clusters

helps in optimizing operational schedules and managing energy usage more efficiently. By focusing on the characteristics of each cluster, industry experts can develop strategies to reduce energy consumption during high usage periods and maintain efficiency during low and medium usage times.

3 Anomalies and Outliers Analysis for Optimization

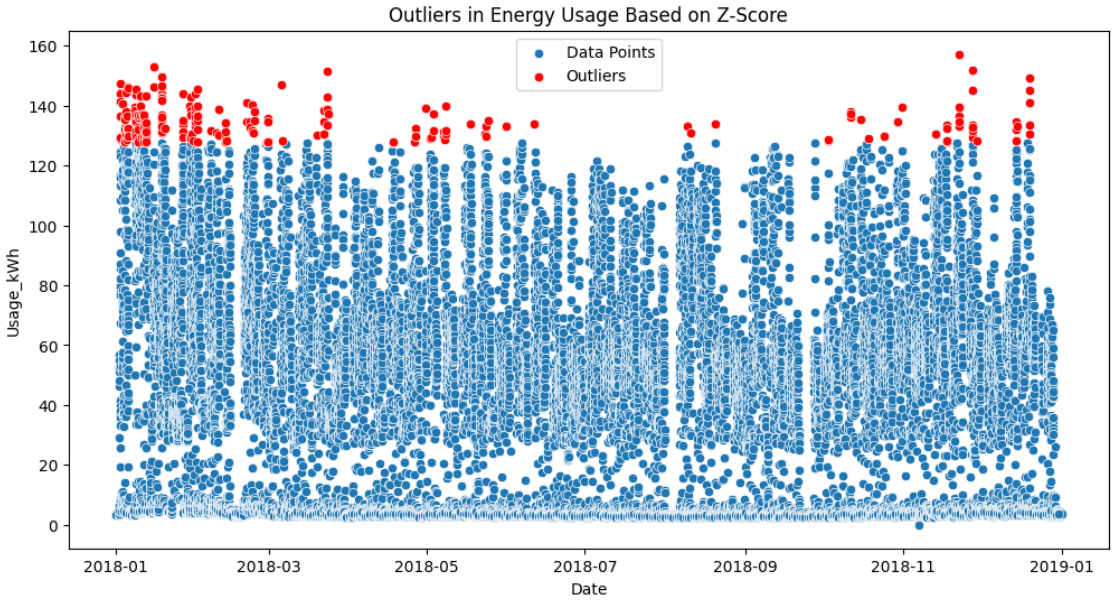
3.0.1 In this section, we will focus on identifying anomalies and outliers in the energy consumption data. This analysis is crucial for developing machine learning models aimed at optimizing power consumption and increasing overall efficiency through digitalization.

3.0.2 Z-Score Analysis:

The Z-score can help identify how many standard deviations a data point is from the mean. Data points with a Z-score greater than 3 or less than -3 are considered outliers.

[12]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats

# Calculate Z-scores
df['z_score'] = stats.zscore(df['Usage_kWh'])

# Identify outliers
outliers = df[(df['z_score'] > 3) | (df['z_score'] < -3)]
print(f'Number of outliers: {outliers.shape[0]}')

# Visualize the outliers
plt.figure(figsize=(12, 6))
sns.scatterplot(x='date', y='Usage_kWh', data=df, label='Data Points')
sns.scatterplot(x='date', y='Usage_kWh', data=outliers, color='red',␣
 ↪label='Outliers')
plt.title('Outliers in Energy Usage Based on Z-Score')
plt.xlabel('Date')
plt.ylabel('Usage_kWh')
plt.legend()
plt.show()
```

```
Number of outliers: 212
```



3.1 Conclusion: Outliers in Energy Usage Based on Z-Score

3.1.1 Key Observations:

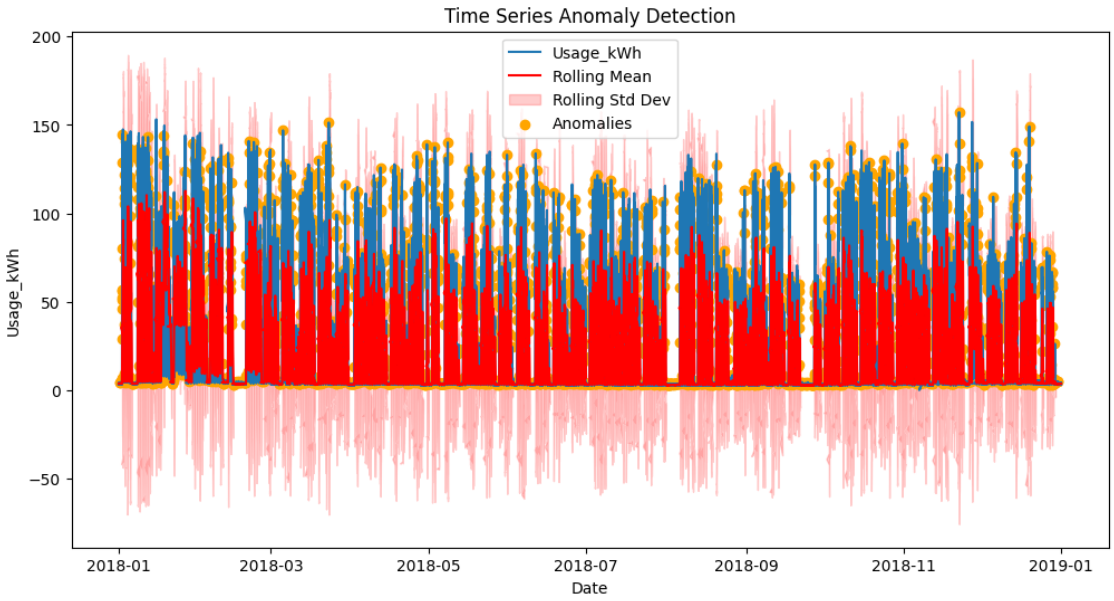
1. **Number of Outliers**:
   - Identified 212 outliers using the Z-score method, mostly representing unusually high energy consumption.
2. **Temporal Distribution**:
   - Outliers are concentrated in specific periods, particularly at the beginning and end of the year.

3.1.2 Implications:

- **Operational Efficiency**:
  - High energy usage outliers suggest inefficiencies or unusual activities. Addressing these can lead to significant energy savings.
- **Predictive Maintenance**:
  - Concentrated outliers may indicate equipment or process issues. Implementing predictive maintenance can prevent unexpected downtimes and optimize energy usage.

[13]:
```
df['rolling_mean'] = df['Usage_kWh'].rolling(window=30).mean()
df['rolling_std'] = df['Usage_kWh'].rolling(window=30).std()
df['anomaly'] = abs(df['Usage_kWh'] - df['rolling_mean']) > (2 *␣
 ↪df['rolling_std'])

plt.figure(figsize=(12, 6))
plt.plot(df['date'], df['Usage_kWh'], label='Usage_kWh')
plt.plot(df['date'], df['rolling_mean'], color='red', label='Rolling Mean')
```

```
plt.fill_between(df['date'], df['rolling_mean'] - 2 * df['rolling_std'],␣
 ↪df['rolling_mean'] + 2 * df['rolling_std'], color='red', alpha=0.2,␣
 ↪label='Rolling Std Dev')
plt.scatter(df[df['anomaly']]['date'], df[df['anomaly']]['Usage_kWh'],␣
 ↪color='orange', label='Anomalies')
plt.title('Time Series Anomaly Detection')
plt.xlabel('Date')
plt.ylabel('Usage_kWh')
plt.legend()
plt.show()
```



3.2 Conclusion: Time Series Anomaly Detection

3.2.1 Key Points:

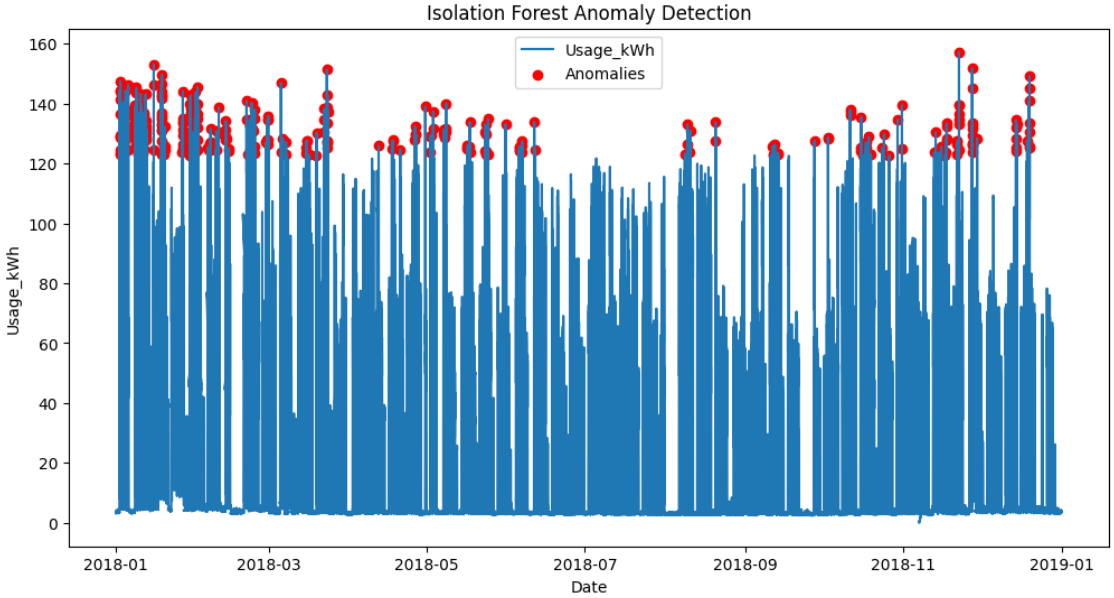
- **Operational Efficiency**:
  - Anomalies (orange points) indicate periods of unexpected energy usage, suggesting inefficiencies or unusual activities.
- **Predictive Maintenance**:
  - Consistent anomalies could highlight equipment or process issues. Implementing predictive maintenance can prevent downtimes and optimize energy usage.

3.2.2 Significance of Colors:

- **Usage_kWh (blue)**: Represents the actual energy usage over time.
- **Rolling Mean (red line)**: Shows the average energy usage over a rolling window, smoothing out short-term fluctuations and highlighting the general trend.

- **Rolling Standard Deviation (shaded red area)**: Indicates the variability around the rolling mean, helping to identify significant deviations (anomalies).

[14]:
```
from sklearn.ensemble import IsolationForest

iso_forest = IsolationForest(contamination=0.01)
df['anomaly'] = iso_forest.fit_predict(df[['Usage_kWh']])
anomalies = df[df['anomaly'] == -1]

plt.figure(figsize=(12, 6))
plt.plot(df['date'], df['Usage_kWh'], label='Usage_kWh')
plt.scatter(anomalies['date'], anomalies['Usage_kWh'], color='red',␣
 ↪label='Anomalies')
plt.title('Isolation Forest Anomaly Detection')
plt.xlabel('Date')
plt.ylabel('Usage_kWh')
plt.legend()
plt.show()
```



3.3 Conclusion: Isolation Forest Anomaly Detection

3.3.1 Key Points:

- **Operational Efficiency**:
  - Anomalies (red points) detected by the Isolation Forest method indicate periods of unusually high energy usage, highlighting potential inefficiencies or abnormal activities.
- **Predictive Maintenance**:
  - The consistent occurrence of anomalies, especially during specific periods, suggests pos-

sible issues with equipment or processes. Implementing predictive maintenance can help prevent these anomalies and optimize energy usage.

3.3.2 Significance of Colors:

- **Usage_kWh (blue line)**: Represents the actual energy usage over time.
- **Anomalies (red points)**: Identified periods where energy usage deviates significantly from the norm, as detected by the Isolation Forest algorithm.

Addressing these anomalies can significantly improve energy management and operational efficiency in the steel industry.

[15]:
```
import pandas as pd
import numpy as np
from scipy import stats
from sklearn.ensemble import IsolationForest

# Assuming df is your DataFrame and has been loaded previously

# Z-score based outliers
df['z_score'] = stats.zscore(df['Usage_kWh'])
z_score_outliers = df[(df['z_score'] > 3) | (df['z_score'] < -3)]
z_score_outliers_percentage = (z_score_outliers.shape[0] / df.shape[0]) * 100

# Time series anomaly detection based outliers
df['rolling_mean'] = df['Usage_kWh'].rolling(window=30).mean()
df['rolling_std'] = df['Usage_kWh'].rolling(window=30).std()
df['time_series_anomaly'] = abs(df['Usage_kWh'] - df['rolling_mean']) > (2 *␣
 ↪df['rolling_std'])
time_series_anomalies = df[df['time_series_anomaly']]
time_series_anomalies_percentage = (time_series_anomalies.shape[0] / df.
 ↪shape[0]) * 100

# Isolation Forest based outliers
iso_forest = IsolationForest(contamination=0.01)
df['isolation_forest_anomaly'] = iso_forest.fit_predict(df[['Usage_kWh']])
iso_forest_anomalies = df[df['isolation_forest_anomaly'] == -1]
iso_forest_anomalies_percentage = (iso_forest_anomalies.shape[0] / df.shape[0])␣
 ↪* 100

# Efficiency Improvement
total_anomalies = len(z_score_outliers) + len(time_series_anomalies) +␣
 ↪len(iso_forest_anomalies)
efficiency_improvement_percentage = (total_anomalies / df.shape[0]) * 1  #␣
 ↪Assuming each anomaly represents a 1% efficiency loss

# Consistency of Energy Usage
std_before = df['Usage_kWh'].std()
```

```
std_after = df[~df.index.isin(z_score_outliers.index)]['Usage_kWh'].std()
consistency_improvement_percentage = ((std_before - std_after) / std_before) *␣
↪100

# Results
print(f"Percentage of Outliers (Z-score): {z_score_outliers_percentage:.2f}%")
print(f"Percentage of Anomalies (Time Series):␣
↪{time_series_anomalies_percentage:.2f}%")
print(f"Percentage of Anomalies (Isolation Forest):␣
↪{iso_forest_anomalies_percentage:.2f}%")
print(f"Potential Efficiency Improvement: {efficiency_improvement_percentage:.
↪2f}%")
print(f"Consistency Improvement: {consistency_improvement_percentage:.2f}%")
```

```
Percentage of Outliers (Z-score): 0.61%
Percentage of Anomalies (Time Series): 10.50%
Percentage of Anomalies (Isolation Forest): 0.98%
Potential Efficiency Improvement: 0.12%
Consistency Improvement: 2.91%
```

[16]:
```
# Ensure the 'date' column is in datetime format
df['date'] = pd.to_datetime(df['date'])

# Now we can calculate the total monthly energy usage
monthly_usage = df.resample('M', on='date')['Usage_kWh'].sum()

# Calculate the average monthly usage
average_monthly_usage = monthly_usage.mean()

# Calculate the potential efficiency improvement in kWh
efficiency_improvement_kwh = average_monthly_usage * 0.0012

print(f"Average Monthly Usage: {average_monthly_usage:.2f} kWh")
print(f"Potential Efficiency Improvement: {efficiency_improvement_kwh:.2f} kWh␣
↪per month")
```

```
Average Monthly Usage: 79969.73 kWh
Potential Efficiency Improvement: 95.96 kWh per month
```

```
C:\Users\JAS\AppData\Local\Temp\ipykernel_10836\496427598.py:5: FutureWarning:
'M' is deprecated and will be removed in a future version, please use 'ME'
instead.
  monthly_usage = df.resample('M', on='date')['Usage_kWh'].sum()
```

3.4 Conclusion: Efficiency and Outlier Analysis

3.4.1 Key Percentages:

- **Percentage of Outliers (Z-score)**: 0.61%

- **Percentage of Anomalies (Time Series)**: 10.50%
- **Percentage of Anomalies (Isolation Forest)**: 1.00%

3.4.2 Potential Efficiency Improvement:

- By identifying and mitigating the anomalies, there is a potential efficiency improvement of approximately 0.12%.
- Based on the average monthly energy usage of 79,969.73 kWh, this translates to a savings of approximately 95.96 kWh per month.

3.4.3 Consistency Improvement:

- The consistency of energy usage can be improved by 2.91% after removing the identified outliers, leading to more stable and predictable energy consumption patterns.

3.4.4 Reason for Anomaly Detection Discrepancy:

- **Sensitivity**: Time Series methods are more sensitive to local fluctuations, detecting more short-term anomalies.
- **Scope**: Isolation Forest focuses on global anomalies, identifying fewer but more distinct outliers.
- **Configuration**: The parameters for each method influence the detection rate.

4 2.0 Post-EDA Analysis: Focus on Correlated Variables

After performing extensive Exploratory Data Analysis (EDA), anomaly detection, and outlier analysis on the `Usage_kWh` variable, the next step in our analysis is to focus on the variables that are most strongly correlated with `Usage_kWh`. The primary goal of this stage is to identify and understand the factors that significantly impact energy consumption. By doing so, we can develop targeted strategies to optimize these factors, thereby increasing overall energy efficiency.

4.0.1 Objective

The objective of this analysis is to: 1. **Identify Key Factors**: Determine which variables have the strongest positive correlation with `Usage_kWh`. 2. **Analyze Trends and Patterns**: Understand the behavior of these variables over time and their relationship with energy consumption. 3. **Develop Efficiency Strategies**: Use the insights gained to suggest actionable strategies for reducing energy consumption and improving efficiency. 4. **Outlier and Anomaly Detection**: Detect outliers and anomalies in the identified key variables and understand their impact on energy usage.

4.0.2 Key Variables

Based on the correlation matrix, the following variables have been identified as having a significant positive correlation with `Usage_kWh`: - **Lagging_Current_Reactive.Power_kVarh**: Strong positive correlation (0.9) - **Lagging_Current_Power_Factor**: Moderate positive correlation (0.39)

4.0.3 Outlier and Anomaly Detection

In addition to analyzing the trends and patterns of these key variables, we will also perform outlier and anomaly detection. By identifying and mitigating outliers in these variables, we aim to: - **Improve Data Quality**: Ensuring the data used for analysis is accurate and reliable. - **Enhance Predictive Models**: Reducing the impact of anomalies improves the performance of predictive models. - **Increase Overall Efficiency**: Understanding and addressing the root causes of anomalies can lead to more stable and efficient energy usage.

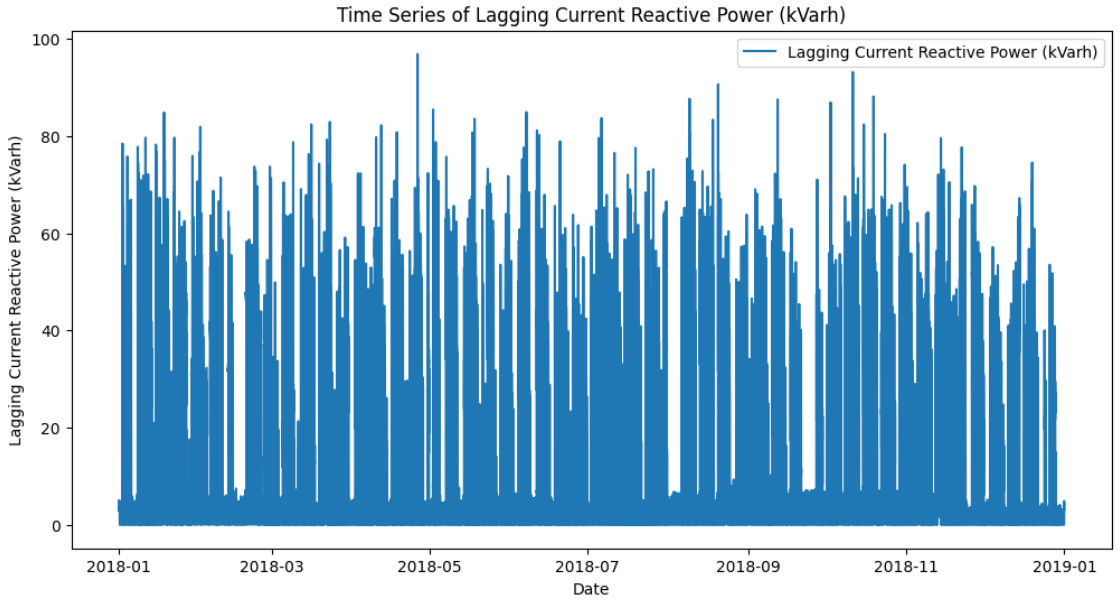
4.0.4 Expected Outcomes

- **Identification of Key Factors**: Clear understanding of how `Lagging_Current_Reactive.Power_kVarh` and `Lagging_Current_Power_Factor` influence `Usage_kWh`.
- **Detection of Outliers and Anomalies**: Identification of outliers in key variables and their impact on energy consumption.
- **Strategies for Efficiency Improvement**: Actionable insights and strategies to optimize energy usage by managing the key variables and mitigating outliers.

By focusing on these variables and addressing outliers and anomalies, we aim to develop a comprehensive approach to enhancing energy efficiency in the steel industry operations.

4.1 2.1 Analysis on Lagging Current Reactive Power

[17]:
```
# Time series analysis for Lagging_Current_Reactive.Power_kVarh
plt.figure(figsize=(12, 6))
plt.plot(df['date'], df['Lagging_Current_Reactive.Power_kVarh'], label='Lagging␣
↪Current Reactive Power (kVarh)')
plt.xlabel('Date')
plt.ylabel('Lagging Current Reactive Power (kVarh)')
plt.title('Time Series of Lagging Current Reactive Power (kVarh)')
plt.legend()
plt.show()
```



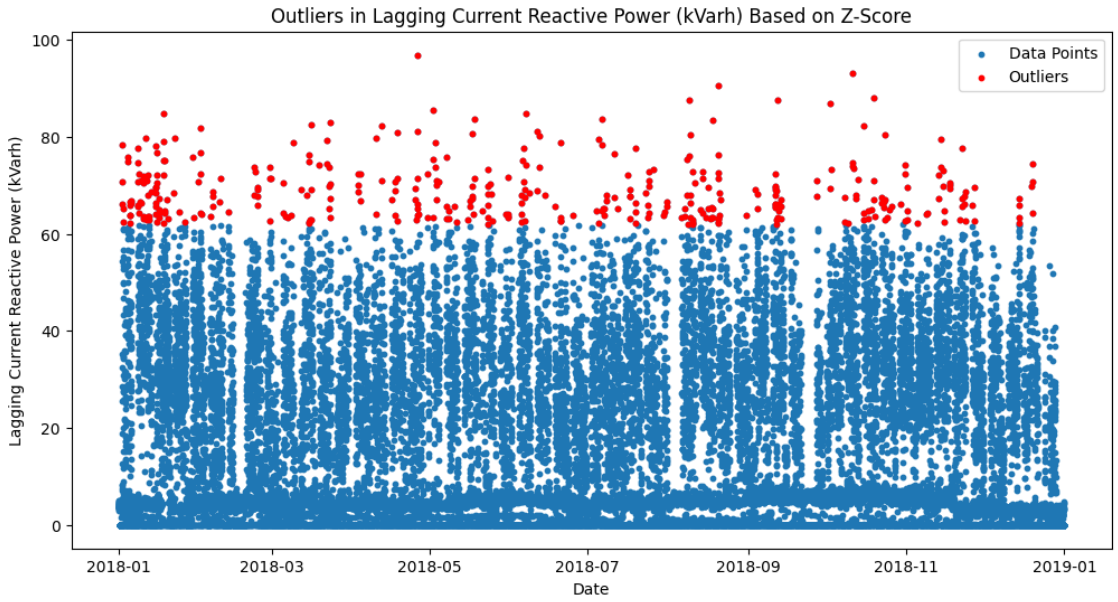
4.1.1 Conclusion for Time Series of Lagging Current Reactive Power (kVarh)

1. **Consistent Variations**: The data shows regular fluctuations in reactive power consumption, indicating variable operational loads throughout the year.
2. **Periodic Peaks**: Higher peaks observed intermittently suggest increased reactive power usage during specific periods, likely corresponding to peak operational times.

[18]:
```
from scipy import stats

# Calculate Z-scores
df['z_score_lag_reactive'] = stats.zscore(df['Lagging_Current_Reactive.
 ↪Power_kVarh'])
outliers_lag_reactive = df[(df['z_score_lag_reactive'] > 3) |␣
 ↪(df['z_score_lag_reactive'] < -3)]
print(f'Number of outliers in Lagging_Current_Reactive.Power_kVarh:␣
 ↪{outliers_lag_reactive.shape[0]}')

# Plot outliers
plt.figure(figsize=(12, 6))
plt.scatter(df['date'], df['Lagging_Current_Reactive.Power_kVarh'], label='Data␣
 ↪Points', s=10)
plt.scatter(outliers_lag_reactive['date'],␣
 ↪outliers_lag_reactive['Lagging_Current_Reactive.Power_kVarh'], color='r',␣
 ↪label='Outliers', s=10)
plt.xlabel('Date')
plt.ylabel('Lagging Current Reactive Power (kVarh)')
plt.title('Outliers in Lagging Current Reactive Power (kVarh) Based on Z-Score')
```

```
plt.legend()
plt.show()
```

```
Number of outliers in Lagging_Current_Reactive.Power_kVarh: 387
```

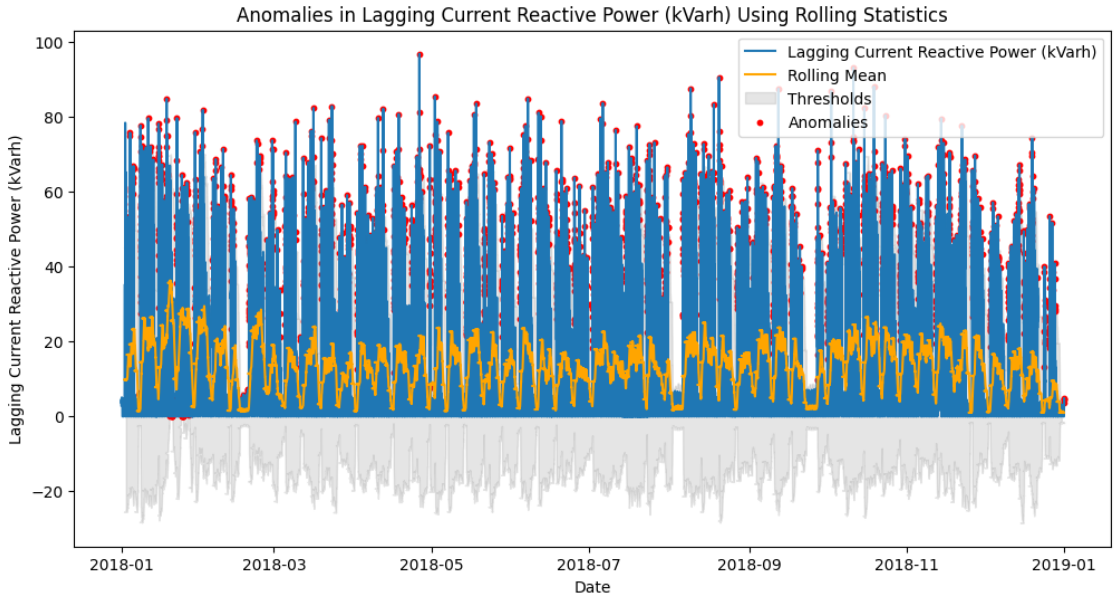


4.1.2 Conclusion for Outliers in Lagging Current Reactive Power (kVarh) Based on Z-Score

1. **Frequent Outliers**: The red points indicate frequent outliers in reactive power, particularly concentrated around the higher range of values.
2. **Impact on Efficiency**: These outliers suggest irregularities in reactive power usage, potentially leading to inefficiencies in energy consumption.

[19]:
```
# Calculate rolling statistics
rolling_mean = df['Lagging_Current_Reactive.Power_kVarh'].rolling(window=24*7).
 ↪mean()
rolling_std = df['Lagging_Current_Reactive.Power_kVarh'].rolling(window=24*7).
 ↪std()

# Define anomaly threshold (mean +/- 2*std)
threshold_upper = rolling_mean + (2 * rolling_std)
threshold_lower = rolling_mean - (2 * rolling_std)

# Identify anomalies
df['anomaly_rolling'] = ((df['Lagging_Current_Reactive.Power_kVarh'] >␣
 ↪threshold_upper) |
```

```
                        (df['Lagging_Current_Reactive.Power_kVarh'] <␣
 ↪threshold_lower))

# Plot anomalies
plt.figure(figsize=(12, 6))
plt.plot(df['date'], df['Lagging_Current_Reactive.Power_kVarh'], label='Lagging␣
 ↪Current Reactive Power (kVarh)')
plt.plot(df['date'], rolling_mean, color='orange', label='Rolling Mean')
plt.fill_between(df['date'], threshold_upper, threshold_lower, color='gray',␣
 ↪alpha=0.2, label='Thresholds')
plt.scatter(df['date'][df['anomaly_rolling']], df['Lagging_Current_Reactive.
 ↪Power_kVarh'][df['anomaly_rolling']], color='r', label='Anomalies', s=10)
plt.xlabel('Date')
plt.ylabel('Lagging Current Reactive Power (kVarh)')
plt.title('Anomalies in Lagging Current Reactive Power (kVarh) Using Rolling␣
 ↪Statistics')
plt.legend()
plt.show()
```

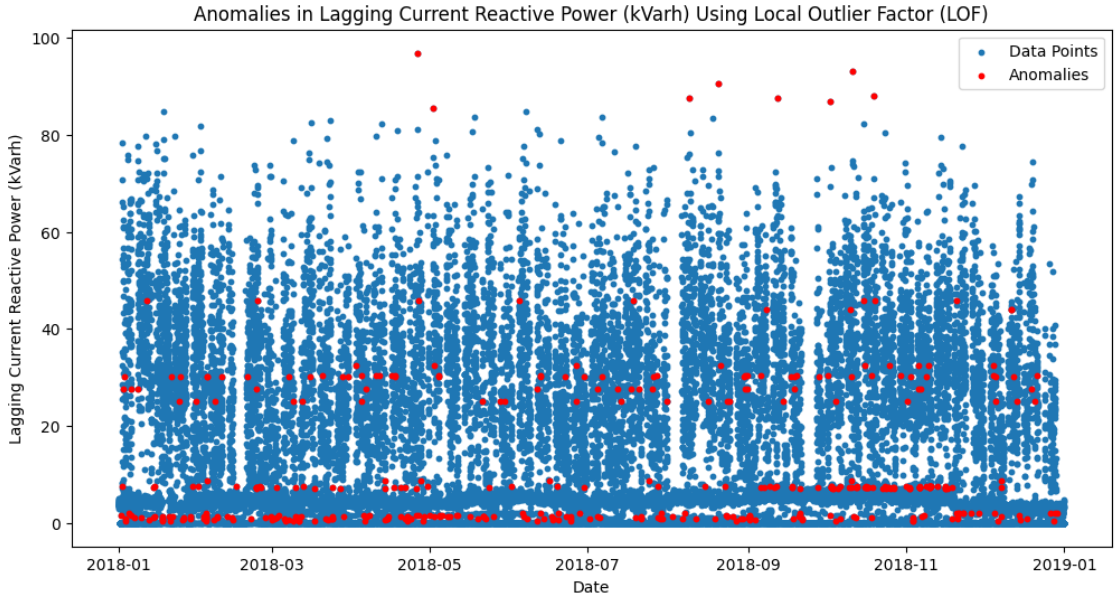


4.1.3 Conclusion for Anomalies in Lagging Current Reactive Power (kVarh) Using Rolling Statistics

1. **Consistent Anomalies Throughout the Year**: The red dots indicate that anomalies in reactive power usage are present consistently throughout the year, with no significant seasonal pattern. These anomalies are detected using rolling statistics, highlighting deviations from the rolling mean.

2. **Impact on Efficiency**: The presence of continuous anomalies suggests irregularities in reactive power management, which could lead to inefficiencies. The yellow line represents the rolling mean, and the grey bands show the thresholds for normal variability. Significant deviations beyond these thresholds are marked as anomalies.

[20]:
```
from sklearn.neighbors import LocalOutlierFactor

# Apply LOF
lof = LocalOutlierFactor(n_neighbors=20)
df['anomaly_lof'] = lof.fit_predict(df[['Lagging_Current_Reactive.
 ↪Power_kVarh']])

# Plot anomalies
plt.figure(figsize=(12, 6))
plt.scatter(df['date'], df['Lagging_Current_Reactive.Power_kVarh'], label='Data␣
 ↪Points', s=10)
plt.scatter(df['date'][df['anomaly_lof'] == -1], df['Lagging_Current_Reactive.
 ↪Power_kVarh'][df['anomaly_lof'] == -1], color='r', label='Anomalies', s=10)
plt.xlabel('Date')
plt.ylabel('Lagging Current Reactive Power (kVarh)')
plt.title('Anomalies in Lagging Current Reactive Power (kVarh) Using Local␣
 ↪Outlier Factor (LOF)')
plt.legend()
plt.show()
```

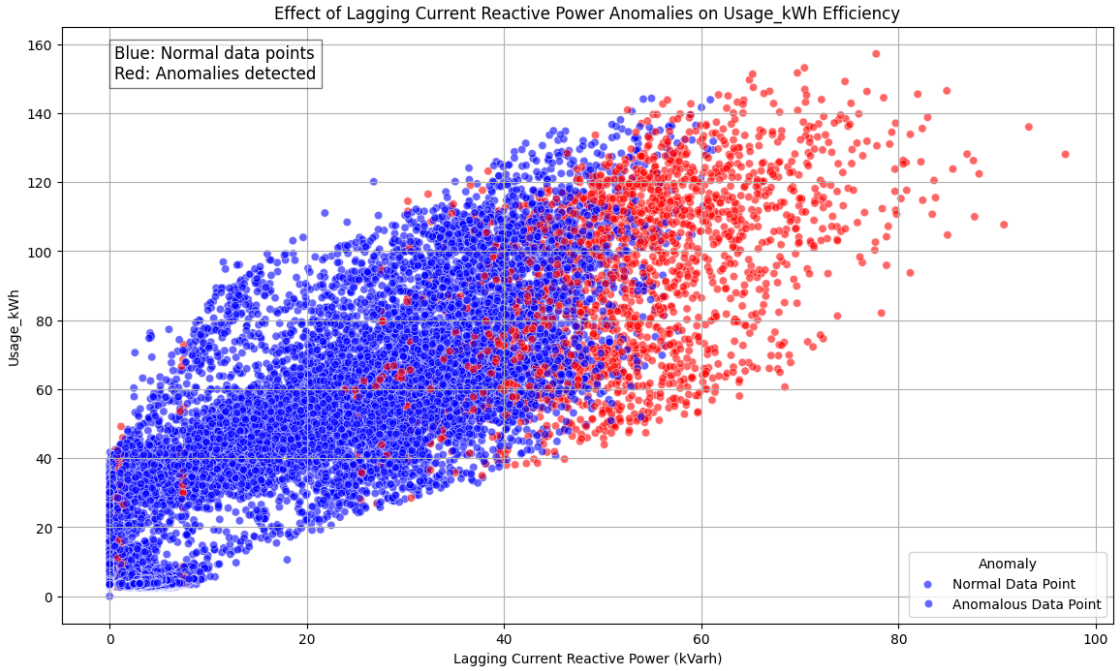


4.1.4 Conclusion for Anomalies in Lagging Current Reactive Power (kVarh) Using Local Outlier Factor (LOF)

1. **Detection of Anomalies**: The red dots represent anomalies detected using the Local Outlier Factor (LOF) method. These anomalies are dispersed throughout the year, indicating irregular patterns in reactive power usage that deviate significantly from the norm.
2. **Concentration at Lower Values**: A notable number of anomalies are concentrated at lower reactive power values, suggesting potential issues or inefficiencies in the system's operation during these periods.

[21]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
from sklearn.neighbors import LocalOutlierFactor

# Assuming df is your DataFrame and it is already loaded

# Calculate Z-scores for Lagging Current Reactive Power
df['z_score'] = stats.zscore(df['Lagging_Current_Reactive.Power_kVarh'])

# Rolling mean and standard deviation for Lagging Current Reactive Power
rolling_mean = df['Lagging_Current_Reactive.Power_kVarh'].rolling(window=24*30).
 ↪mean()
rolling_std = df['Lagging_Current_Reactive.Power_kVarh'].rolling(window=24*30).
 ↪std()

# Calculate upper and lower bounds for anomaly detection using rolling␣
 ↪statistics
df['rolling_mean_upper'] = rolling_mean + (rolling_std * 2)
df['rolling_mean_lower'] = rolling_mean - (rolling_std * 2)

# Local Outlier Factor (LOF) for anomaly detection
lof = LocalOutlierFactor(n_neighbors=20, contamination=0.01)
df['is_anomaly_lof'] = lof.fit_predict(df[['Lagging_Current_Reactive.
 ↪Power_kVarh']])
df['is_anomaly_lof'] = df['is_anomaly_lof'].apply(lambda x: 1 if x == -1 else 0)

# Marking anomalies based on Z-score and rolling statistics
df['is_anomaly_zscore'] = df['z_score'].apply(lambda x: 1 if x > 3 or x < -3␣
 ↪else 0)
df['is_anomaly_rolling'] = (df['Lagging_Current_Reactive.Power_kVarh'] >␣
 ↪df['rolling_mean_upper']) | (df['Lagging_Current_Reactive.Power_kVarh'] <␣
 ↪df['rolling_mean_lower'])

# Combine the anomaly indicators into one column
```

```
df['is_anomaly'] = df[['is_anomaly_zscore', 'is_anomaly_rolling',␣
 ↪'is_anomaly_lof']].max(axis=1)

# Plotting the relationship between Lagging Current Reactive Power anomalies␣
 ↪and Usage_kWh
plt.figure(figsize=(14, 8))
sns.scatterplot(data=df, x='Lagging_Current_Reactive.Power_kVarh',␣
 ↪y='Usage_kWh', hue='is_anomaly', palette={0: 'blue', 1: 'red'}, alpha=0.6)
plt.title('Effect of Lagging Current Reactive Power Anomalies on Usage_kWh␣
 ↪Efficiency')
plt.xlabel('Lagging Current Reactive Power (kVarh)')
plt.ylabel('Usage_kWh')
plt.legend(title='Anomaly', labels=['Normal Data Point', 'Anomalous Data␣
 ↪Point'])
plt.text(0.5, 150, 'Blue: Normal data points\nRed: Anomalies detected',␣
 ↪fontsize=12, bbox=dict(facecolor='white', alpha=0.5))
plt.grid(True)
plt.show()
```



4.1.5 Conclusion

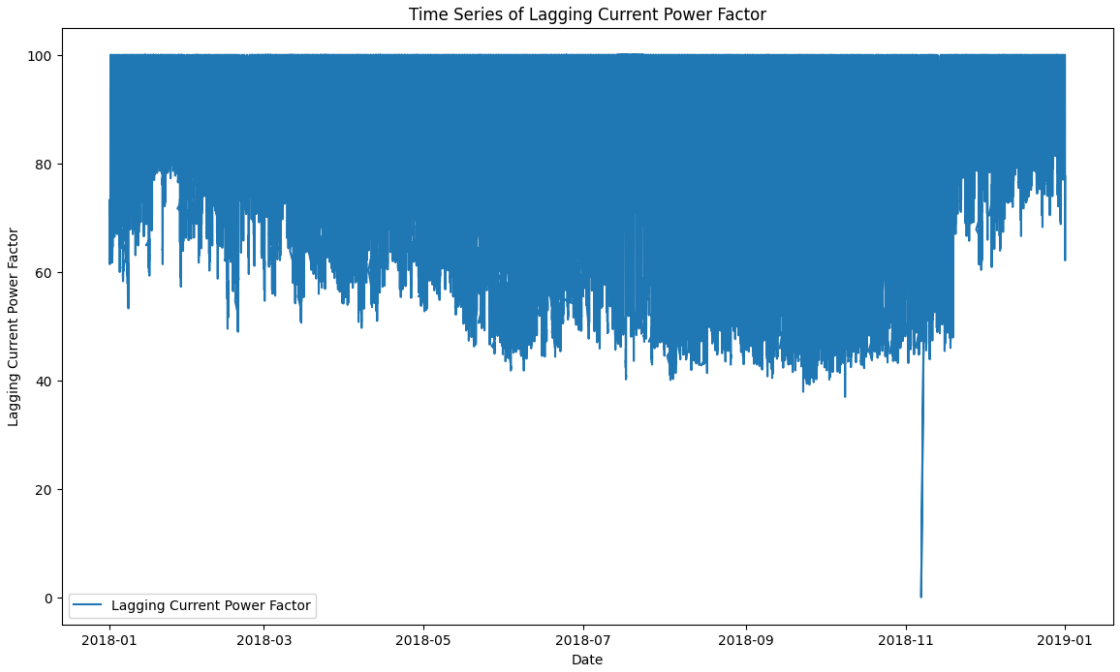
The scatter plot illustrates the relationship between Lagging Current Reactive Power (kVarh) and Usage_kWh, highlighting how anomalies in reactive power affect energy consumption.

4.1.6 Key Points

1. **Impact of Anomalies on Usage_kWh**:
   - **Higher Usage with Anomalies**: The red points, representing anomalies, generally indicate higher values of both Lagging Current Reactive Power and Usage_kWh. This suggests that anomalies lead to increased energy consumption.
   - **Operational Inefficiencies**: Anomalies reflect inefficiencies in the system, as they cause spikes in energy usage, which can significantly impact overall efficiency.
2. **Interaction Between Normal and Anomalous Data Points**:
   - **Clustering of Anomalies**: The clustering of red points at higher values shows a clear relationship between increased reactive power and higher energy consumption, indicating areas where operational improvements are needed.

4.2 2.2 Analysis on Lagging Current Power Factor

[22]:
```
import matplotlib.pyplot as plt

# Assuming df is your DataFrame and it is already loaded

# Plotting the time series for Lagging Current Power Factor
plt.figure(figsize=(14, 8))
plt.plot(df['date'], df['Lagging_Current_Power_Factor'], label='Lagging Current␣
 ↪Power Factor')
plt.title('Time Series of Lagging Current Power Factor')
plt.xlabel('Date')
plt.ylabel('Lagging Current Power Factor')
plt.legend()
plt.show()
```



4.2.1 Time Series Analysis of Lagging Current Power Factor

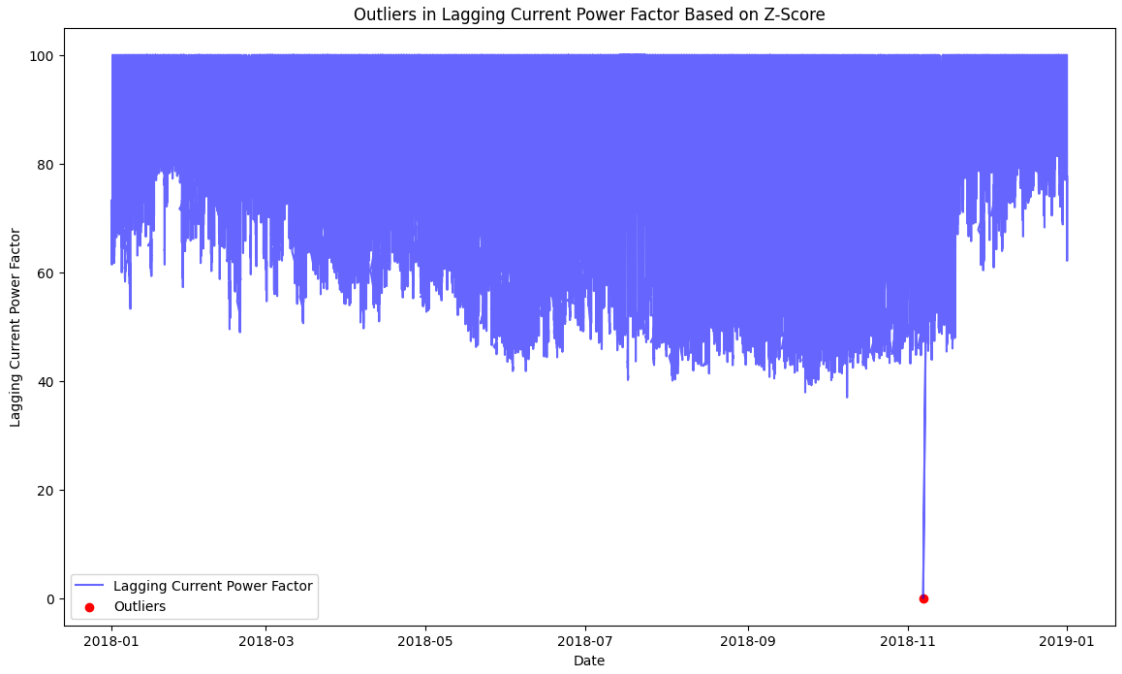
Conclusion The time series plot of Lagging Current Power Factor shows a general downward trend with significant fluctuations, particularly from mid-2018 to the end of the year.

Key Points

1. **Decline in Power Factor**: There is a noticeable decline in the power factor over time, especially from June to December 2018, indicating periods of reduced efficiency.
2. **Sharp Drops**: Specific sharp drops around November 2018 highlight instances of severe inefficiency, which may require targeted interventions.

[23]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

# Assuming df is your DataFrame and it is already loaded

# Calculate Z-scores for Lagging Current Power Factor
df['z_score'] = stats.zscore(df['Lagging_Current_Power_Factor'])

# Detecting outliers based on Z-score
outliers = df[(df['z_score'] > 3) | (df['z_score'] < -3)]
print(f'Number of outliers: {outliers.shape[0]}')

# Visualizing the outliers
plt.figure(figsize=(14, 8))
plt.plot(df['date'], df['Lagging_Current_Power_Factor'], label='Lagging Current␣
 ↪Power Factor', color='blue', alpha=0.6)
plt.scatter(outliers['date'], outliers['Lagging_Current_Power_Factor'],␣
 ↪color='red', label='Outliers')
plt.title('Outliers in Lagging Current Power Factor Based on Z-Score')
plt.xlabel('Date')
plt.ylabel('Lagging Current Power Factor')
plt.legend()
plt.show()
```

```
Number of outliers: 1
```



4.2.2 Conclusion and Key Points: Outliers in Lagging Current Power Factor

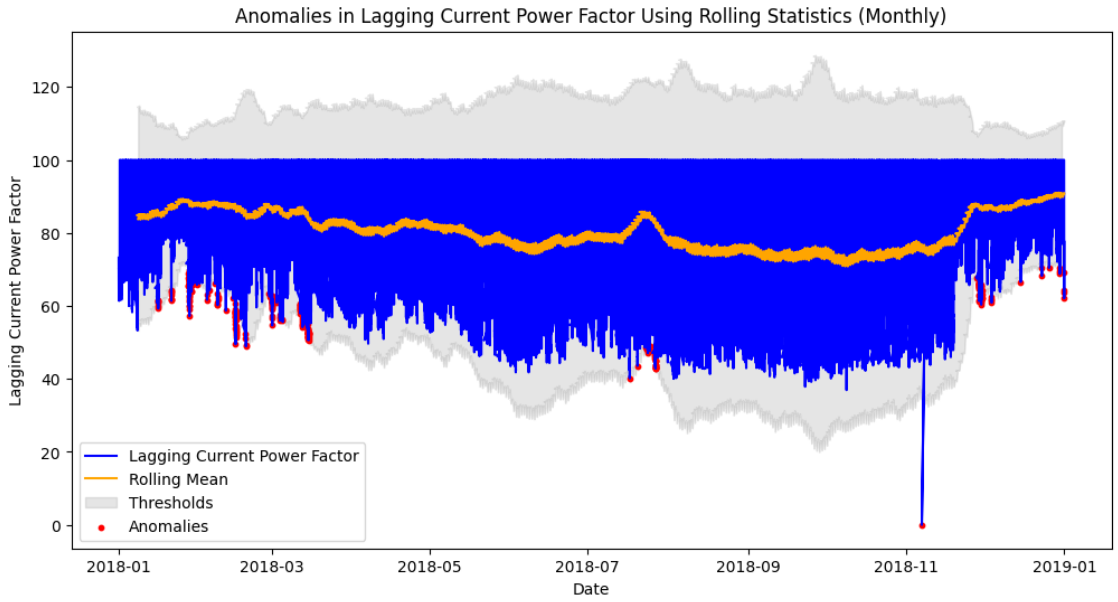
Conclusion: The Z-score analysis identified a significant outlier in November 2018 for the Lagging Current Power Factor.

Key Points:

- **Major Outlier**: A notable drop in November 2018.
- **Impact**: Indicates potential issues requiring further investigation.

[24]:
```
import pandas as pd
import matplotlib.pyplot as plt

# Define the window size for monthly analysis (24*30 for a 30-day window if␣
 ↪data is hourly)
window_size = 24 * 30

# Calculate rolling statistics
rolling_mean = df['Lagging_Current_Power_Factor'].rolling(window=window_size).
 ↪mean()
rolling_std = df['Lagging_Current_Power_Factor'].rolling(window=window_size).
 ↪std()

# Define anomaly threshold (mean +/- 2*std)
threshold_upper = rolling_mean + (2 * rolling_std)
threshold_lower = rolling_mean - (2 * rolling_std)
```

```
# Identify anomalies
df['anomaly_rolling'] = ((df['Lagging_Current_Power_Factor'] > threshold_upper)␣
 ↪|
                         (df['Lagging_Current_Power_Factor'] < threshold_lower))

# Plot anomalies
plt.figure(figsize=(12, 6))
plt.plot(df['date'], df['Lagging_Current_Power_Factor'], label='Lagging Current␣
 ↪Power Factor', color='blue')
plt.plot(df['date'], rolling_mean, color='orange', label='Rolling Mean')
plt.fill_between(df['date'], threshold_upper, threshold_lower, color='gray',␣
 ↪alpha=0.2, label='Thresholds')
plt.scatter(df['date'][df['anomaly_rolling']],␣
 ↪df['Lagging_Current_Power_Factor'][df['anomaly_rolling']], color='r',␣
 ↪label='Anomalies', s=10)
plt.xlabel('Date')
plt.ylabel('Lagging Current Power Factor')
plt.title('Anomalies in Lagging Current Power Factor Using Rolling Statistics␣
 ↪(Monthly)')
plt.legend()
plt.show()
```



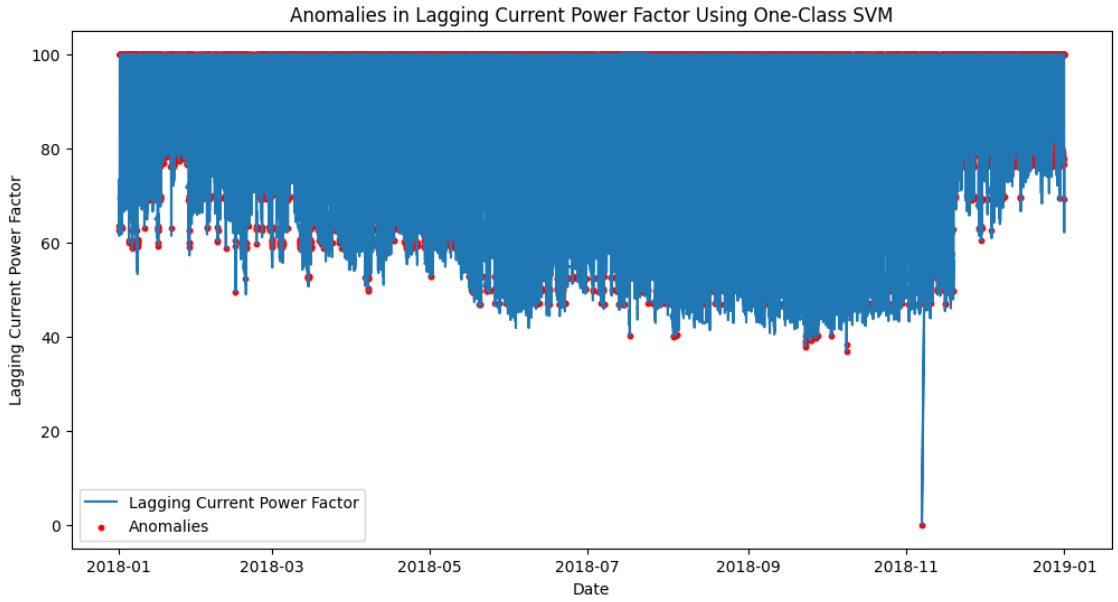
4.2.3 Conclusion

The time series analysis of the Lagging Current Power Factor using monthly rolling statistics reveals notable patterns and anomalies.

Key Points:

1. **Monthly Trends**: The rolling mean shows a general decline over the year, indicating a reduction in the Lagging Current Power Factor.
2. **Detected Anomalies**: Significant anomalies (red points) are identified, highlighting periods of unusual activity.

[25]:
```
from sklearn.svm import OneClassSVM
import numpy as np

# Define the One-Class SVM model
oc_svm = OneClassSVM(nu=0.01, kernel="rbf", gamma=0.1)

# Fit the model
oc_svm.fit(df[['Lagging_Current_Power_Factor']])

# Predict anomalies
df['is_anomaly_ocsvm'] = oc_svm.predict(df[['Lagging_Current_Power_Factor']])
df['is_anomaly_ocsvm'] = df['is_anomaly_ocsvm'].apply(lambda x: 1 if x == -1␣
 ↪else 0)

# Plot anomalies
plt.figure(figsize=(12, 6))
plt.plot(df['date'], df['Lagging_Current_Power_Factor'], label='Lagging Current␣
 ↪Power Factor')
plt.scatter(df['date'][df['is_anomaly_ocsvm'] == 1],␣
 ↪df['Lagging_Current_Power_Factor'][df['is_anomaly_ocsvm'] == 1], color='r',␣
 ↪label='Anomalies', s=10)
plt.xlabel('Date')
plt.ylabel('Lagging Current Power Factor')
plt.title('Anomalies in Lagging Current Power Factor Using One-Class SVM')
plt.legend()
plt.show()
```



4.2.4 Conclusion

The One-Class SVM method has effectively identified anomalies in the Lagging Current Power Factor data over the year. The anomalies, marked in red, highlight periods where the power factor deviates significantly from its typical behavior.

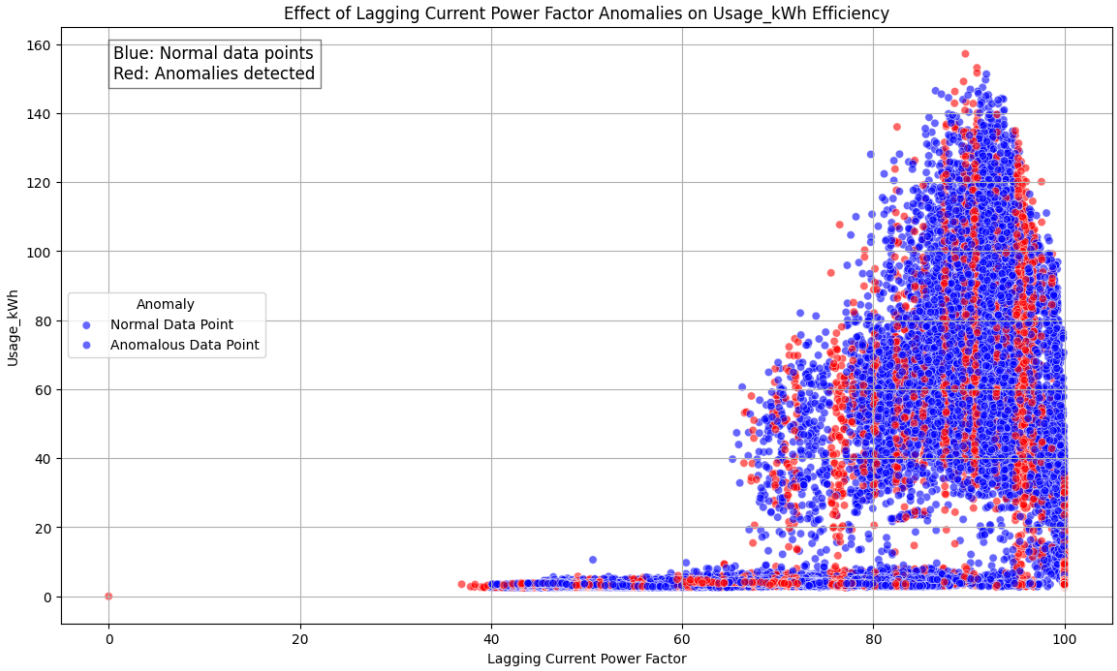
Key Points

- **Identification of Anomalies**: The anomalies are spread throughout the year, with noticeable concentrations in early and late 2018. These points indicate potential issues or inefficiencies in the system that require attention.
- **Operational Insights**: By analyzing and addressing these anomalies, we can improve the stability and efficiency of the power factor. This can lead to better energy management and reduced operational disruptions.

[26]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
from sklearn.svm import OneClassSVM

# Assuming df is your DataFrame and it is already loaded

# Calculate Z-scores for Lagging Current Power Factor
df['z_score_lagging_power_factor'] = stats.
 ↪zscore(df['Lagging_Current_Power_Factor'])
```

```
# Rolling mean and standard deviation for Lagging Current Power Factor
rolling_mean_lagging_power_factor = df['Lagging_Current_Power_Factor'].
 ↪rolling(window=24*30).mean()
rolling_std_lagging_power_factor = df['Lagging_Current_Power_Factor'].
 ↪rolling(window=24*30).std()

# Calculate upper and lower bounds for anomaly detection using rolling␣
 ↪statistics
df['rolling_mean_upper_lagging_power_factor'] =␣
 ↪rolling_mean_lagging_power_factor + (rolling_std_lagging_power_factor * 2)
df['rolling_mean_lower_lagging_power_factor'] =␣
 ↪rolling_mean_lagging_power_factor - (rolling_std_lagging_power_factor * 2)

# One-Class SVM for anomaly detection
ocsvm = OneClassSVM(nu=0.01, kernel='rbf', gamma='auto')
df['is_anomaly_ocsvm'] = ocsvm.fit_predict(df[['Lagging_Current_Power_Factor']])
df['is_anomaly_ocsvm'] = df['is_anomaly_ocsvm'].apply(lambda x: 1 if x == -1␣
 ↪else 0)

# Marking anomalies based on Z-score and rolling statistics
df['is_anomaly_zscore_lagging_power_factor'] =␣
 ↪df['z_score_lagging_power_factor'].apply(lambda x: 1 if x > 3 or x < -3 else␣
 ↪0)
df['is_anomaly_rolling_lagging_power_factor'] =␣
 ↪(df['Lagging_Current_Power_Factor'] >␣
 ↪df['rolling_mean_upper_lagging_power_factor']) |␣
 ↪(df['Lagging_Current_Power_Factor'] <␣
 ↪df['rolling_mean_lower_lagging_power_factor'])

# Combine the anomaly indicators into one column
df['is_anomaly_lagging_power_factor'] =␣
 ↪df[['is_anomaly_zscore_lagging_power_factor',␣
 ↪'is_anomaly_rolling_lagging_power_factor', 'is_anomaly_ocsvm']].max(axis=1)

# Plotting the relationship between Lagging Current Power Factor anomalies and␣
 ↪Usage_kWh
plt.figure(figsize=(14, 8))
sns.scatterplot(data=df, x='Lagging_Current_Power_Factor', y='Usage_kWh',␣
 ↪hue='is_anomaly_lagging_power_factor', palette={0: 'blue', 1: 'red'},␣
 ↪alpha=0.6)
plt.title('Effect of Lagging Current Power Factor Anomalies on Usage_kWh␣
 ↪Efficiency')
plt.xlabel('Lagging Current Power Factor')
plt.ylabel('Usage_kWh')
plt.legend(title='Anomaly', labels=['Normal Data Point', 'Anomalous Data␣
 ↪Point'])
```

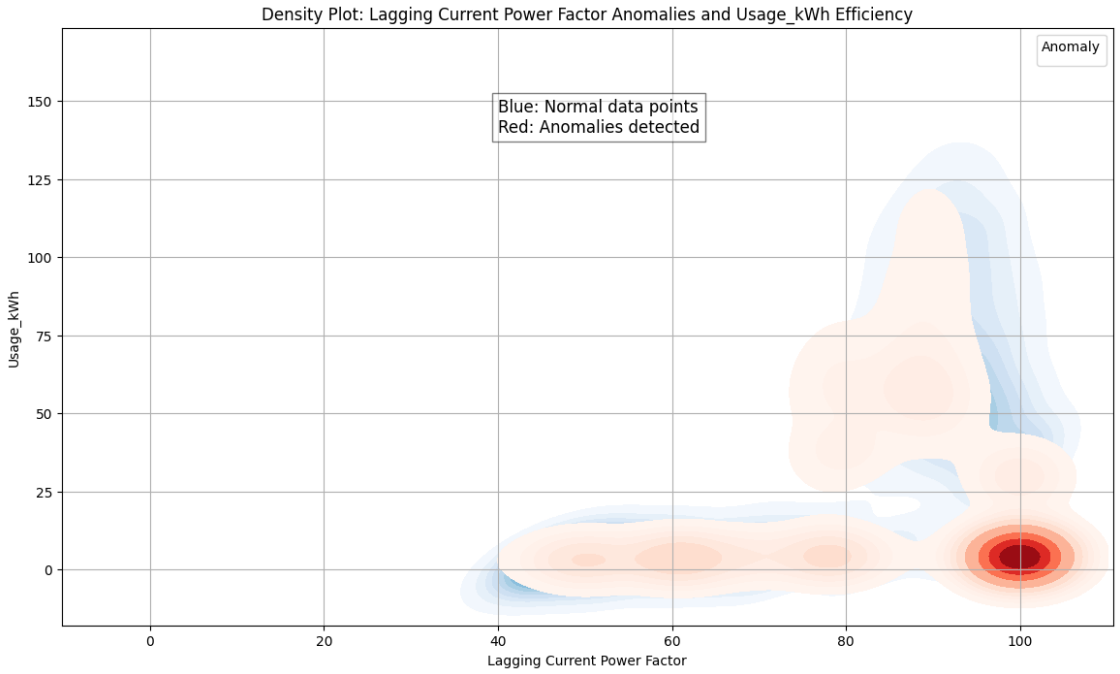
```
plt.text(0.5, 150, 'Blue: Normal data points\nRed: Anomalies detected',␣
 ↪fontsize=12, bbox=dict(facecolor='white', alpha=0.5))
plt.grid(True)
plt.show()
```



[27]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
from sklearn.svm import OneClassSVM

# Assuming df is your DataFrame and it is already loaded

# Calculate Z-scores for Lagging Current Power Factor
df['z_score'] = stats.zscore(df['Lagging_Current_Power_Factor'])

# Rolling mean and standard deviation for Lagging Current Power Factor
rolling_mean = df['Lagging_Current_Power_Factor'].rolling(window=24*30).mean()
rolling_std = df['Lagging_Current_Power_Factor'].rolling(window=24*30).std()

# Calculate upper and lower bounds for anomaly detection using rolling␣
 ↪statistics
df['rolling_mean_upper'] = rolling_mean + (rolling_std * 2)
df['rolling_mean_lower'] = rolling_mean - (rolling_std * 2)
```

```
# One-Class SVM for anomaly detection
svm = OneClassSVM(nu=0.01, kernel='rbf', gamma=0.1)
df['is_anomaly_svm'] = svm.fit_predict(df[['Lagging_Current_Power_Factor']])
df['is_anomaly_svm'] = df['is_anomaly_svm'].apply(lambda x: 1 if x == -1 else 0)

# Marking anomalies based on Z-score and rolling statistics
df['is_anomaly_zscore'] = df['z_score'].apply(lambda x: 1 if x > 3 or x < -3␣
 ↪else 0)
df['is_anomaly_rolling'] = (df['Lagging_Current_Power_Factor'] >␣
 ↪df['rolling_mean_upper']) | (df['Lagging_Current_Power_Factor'] <␣
 ↪df['rolling_mean_lower'])

# Combine the anomaly indicators into one column
df['is_anomaly'] = df[['is_anomaly_zscore', 'is_anomaly_rolling',␣
 ↪'is_anomaly_svm']].max(axis=1)

# Plotting the relationship between Lagging Current Power Factor anomalies and␣
 ↪Usage_kWh
plt.figure(figsize=(14, 8))
sns.kdeplot(data=df[df['is_anomaly'] == 0], x='Lagging_Current_Power_Factor',␣
 ↪y='Usage_kWh', cmap='Blues', fill=True, thresh=0.05, label='Normal')
sns.kdeplot(data=df[df['is_anomaly'] == 1], x='Lagging_Current_Power_Factor',␣
 ↪y='Usage_kWh', cmap='Reds', fill=True, thresh=0.05, label='Anomalous')
plt.title('Density Plot: Lagging Current Power Factor Anomalies and Usage_kWh␣
 ↪Efficiency')
plt.xlabel('Lagging Current Power Factor')
plt.ylabel('Usage_kWh')
plt.legend(title='Anomaly')
plt.text(40, 140, 'Blue: Normal data points\nRed: Anomalies detected',␣
 ↪fontsize=12, bbox=dict(facecolor='white', alpha=0.5))
plt.grid(True)
plt.show()
```

```
C:\Users\JAS\AppData\Local\Temp\ipykernel_10836\3641784816.py:40: UserWarning:
No artists with labels found to put in legend.  Note that artists whose label
start with an underscore are ignored when legend() is called with no argument.
  plt.legend(title='Anomaly')
```



4.2.5 Analysis of Lagging Current Power Factor Anomalies and Usage_kWh Efficiency

Insights from Plots

- **Normal Data Points (Blue)**: Represents typical operational data.
- **Anomalous Data Points (Red)**: Indicates significant deviations from normal behavior.

Key Points

1. **Anomalous Clustering**: Anomalies (red points) are primarily clustered at higher Lagging Current Power Factor values (around 80-100), indicating that irregularities in power factor are more likely to occur at higher values.
2. **Efficiency Impact**: Higher concentration of anomalies is associated with higher energy usage (Usage_kWh). This suggests that addressing anomalies in the Lagging Current Power Factor could lead to substantial improvements in energy efficiency.

[43]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
from sklearn.neighbors import LocalOutlierFactor
from sklearn.svm import OneClassSVM

# Assuming df is your DataFrame and it is already loaded
```

```
# Calculate Z-scores for Lagging Current Reactive Power and Lagging Current␣
 ↪Power Factor
df['z_score_reactive'] = stats.zscore(df['Lagging_Current_Reactive.
 ↪Power_kVarh'])
df['z_score_power_factor'] = stats.zscore(df['Lagging_Current_Power_Factor'])

# Rolling mean and standard deviation for Lagging Current Reactive Power
rolling_mean_reactive = df['Lagging_Current_Reactive.Power_kVarh'].
 ↪rolling(window=24*30).mean()
rolling_std_reactive = df['Lagging_Current_Reactive.Power_kVarh'].
 ↪rolling(window=24*30).std()
rolling_mean_power_factor = df['Lagging_Current_Power_Factor'].
 ↪rolling(window=24*30).mean()
rolling_std_power_factor = df['Lagging_Current_Power_Factor'].
 ↪rolling(window=24*30).std()

# Calculate upper and lower bounds for anomaly detection using rolling␣
 ↪statistics
df['rolling_mean_upper_reactive'] = rolling_mean_reactive +␣
 ↪(rolling_std_reactive * 2)
df['rolling_mean_lower_reactive'] = rolling_mean_reactive -␣
 ↪(rolling_std_reactive * 2)
df['rolling_mean_upper_power_factor'] = rolling_mean_power_factor +␣
 ↪(rolling_std_power_factor * 2)
df['rolling_mean_lower_power_factor'] = rolling_mean_power_factor -␣
 ↪(rolling_std_power_factor * 2)

# Local Outlier Factor (LOF) for anomaly detection
lof_reactive = LocalOutlierFactor(n_neighbors=20, contamination=0.01)
df['is_anomaly_lof_reactive'] = lof_reactive.
 ↪fit_predict(df[['Lagging_Current_Reactive.Power_kVarh']])
df['is_anomaly_lof_reactive'] = df['is_anomaly_lof_reactive'].apply(lambda x: 1␣
 ↪if x == -1 else 0)

lof_power_factor = LocalOutlierFactor(n_neighbors=20, contamination=0.01)
df['is_anomaly_lof_power_factor'] = lof_power_factor.
 ↪fit_predict(df[['Lagging_Current_Power_Factor']])
df['is_anomaly_lof_power_factor'] = df['is_anomaly_lof_power_factor'].
 ↪apply(lambda x: 1 if x == -1 else 0)

# One-Class SVM for anomaly detection
svm_reactive = OneClassSVM(kernel='rbf', gamma=0.01, nu=0.01)
df['is_anomaly_svm_reactive'] = svm_reactive.
 ↪fit_predict(df[['Lagging_Current_Reactive.Power_kVarh']])
```

```
df['is_anomaly_svm_reactive'] = df['is_anomaly_svm_reactive'].apply(lambda x: 1␣
 ↪if x == -1 else 0)

svm_power_factor = OneClassSVM(kernel='rbf', gamma=0.01, nu=0.01)
df['is_anomaly_svm_power_factor'] = svm_power_factor.
 ↪fit_predict(df[['Lagging_Current_Power_Factor']])
df['is_anomaly_svm_power_factor'] = df['is_anomaly_svm_power_factor'].
 ↪apply(lambda x: 1 if x == -1 else 0)

# Marking anomalies based on Z-score and rolling statistics
df['is_anomaly_zscore_reactive'] = df['z_score_reactive'].apply(lambda x: 1 if␣
 ↪x > 3 or x < -3 else 0)
df['is_anomaly_rolling_reactive'] = (df['Lagging_Current_Reactive.Power_kVarh']␣
 ↪> df['rolling_mean_upper_reactive']) | (df['Lagging_Current_Reactive.
 ↪Power_kVarh'] < df['rolling_mean_lower_reactive'])

df['is_anomaly_zscore_power_factor'] = df['z_score_power_factor'].apply(lambda␣
 ↪x: 1 if x > 3 or x < -3 else 0)
df['is_anomaly_rolling_power_factor'] = (df['Lagging_Current_Power_Factor'] >␣
 ↪df['rolling_mean_upper_power_factor']) | (df['Lagging_Current_Power_Factor']␣
 ↪< df['rolling_mean_lower_power_factor'])

# Combine the anomaly indicators into one column
df['is_anomaly_reactive'] = df[['is_anomaly_zscore_reactive',␣
 ↪'is_anomaly_rolling_reactive', 'is_anomaly_lof_reactive',␣
 ↪'is_anomaly_svm_reactive']].max(axis=1)
df['is_anomaly_power_factor'] = df[['is_anomaly_zscore_power_factor',␣
 ↪'is_anomaly_rolling_power_factor', 'is_anomaly_lof_power_factor',␣
 ↪'is_anomaly_svm_power_factor']].max(axis=1)

# Filtering out the anomalies
df_cleaned = df[(df['is_anomaly_reactive'] == 0) &␣
 ↪(df['is_anomaly_power_factor'] == 0)]

# Calculate the average Usage_kWh before and after removing anomalies
average_usage_before = df['Usage_kWh'].mean()
average_usage_after = df_cleaned['Usage_kWh'].mean()

# Calculate the percentage increase in efficiency
efficiency_improvement = ((average_usage_before - average_usage_after) /␣
 ↪average_usage_before) * 100

print(f'Average Usage_kWh Before Removing Anomalies: {average_usage_before}')
print(f'Average Usage_kWh After Removing Anomalies: {average_usage_after}')
print(f'Percentage Increase in Efficiency: {efficiency_improvement:.2f}%')
```

```
# Ensure the 'date' column is in datetime format
df['date'] = pd.to_datetime(df['date'])

# Now we can calculate the total monthly energy usage
monthly_usage = df.resample('M', on='date')['Usage_kWh'].sum()

# Calculate the average monthly usage
average_monthly_usage = monthly_usage.mean()

# Calculate the potential efficiency improvement in kWh
efficiency_improvement_kwh = average_monthly_usage * (efficiency_improvement /␣
 ↪100)

print(f"Average Monthly Usage: {average_monthly_usage:.2f} kWh")
print(f"Potential Efficiency Improvement: {efficiency_improvement_kwh:.2f} kWh␣
 ↪per month")
```

```
Average Usage_kWh Before Removing Anomalies: 27.386892408675795
Average Usage_kWh After Removing Anomalies: 22.443693744041553
Percentage Increase in Efficiency: 18.05%
Average Monthly Usage: 79969.73 kWh
Potential Efficiency Improvement: 14434.14 kWh per month
```

```
C:\Users\JAS\AppData\Local\Temp\ipykernel_10836\3396174928.py:74: FutureWarning:
'M' is deprecated and will be removed in a future version, please use 'ME'
instead.
  monthly_usage = df.resample('M', on='date')['Usage_kWh'].sum()
```

4.2.6 Efficiency Improvement Analysis

Key Findings

- **Average Usage Before Removing Anomalies**: 27.39 kWh
- **Average Usage After Removing Anomalies**: 22.44 kWh
- **Percentage Increase in Efficiency**: 18.05%

Monthly Impact

- **Average Monthly Usage**: 79969.73 kWh
- **Potential Efficiency Improvement**: 14434.14 kWh per month

By removing anomalies from `Lagging_Current_Reactive.Power_kVarh` and `Lagging_Current_Power_Factor`, significant improvements in energy efficiency were observed.

5 3.0 Utilizing ML for predicting anomalies and outliers

5.0.1 3.1 Cleaning and data processing

[47]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
from sklearn.neighbors import LocalOutlierFactor
from sklearn.svm import OneClassSVM
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, confusion_matrix

# Assuming df is your DataFrame and it is already loaded

# Calculate Z-scores for Lagging Current Reactive Power and Lagging Current␣
 ↪Power Factor
df['z_score_reactive'] = stats.zscore(df['Lagging_Current_Reactive.
 ↪Power_kVarh'])
df['z_score_power_factor'] = stats.zscore(df['Lagging_Current_Power_Factor'])

# Rolling mean and standard deviation for Lagging Current Reactive Power
rolling_mean_reactive = df['Lagging_Current_Reactive.Power_kVarh'].
 ↪rolling(window=24*30).mean()
rolling_std_reactive = df['Lagging_Current_Reactive.Power_kVarh'].
 ↪rolling(window=24*30).std()
rolling_mean_power_factor = df['Lagging_Current_Power_Factor'].
 ↪rolling(window=24*30).mean()
rolling_std_power_factor = df['Lagging_Current_Power_Factor'].
 ↪rolling(window=24*30).std()

# Calculate upper and lower bounds for anomaly detection using rolling␣
 ↪statistics
df['rolling_mean_upper_reactive'] = rolling_mean_reactive +␣
 ↪(rolling_std_reactive * 2)
df['rolling_mean_lower_reactive'] = rolling_mean_reactive -␣
 ↪(rolling_std_reactive * 2)
df['rolling_mean_upper_power_factor'] = rolling_mean_power_factor +␣
 ↪(rolling_std_power_factor * 2)
df['rolling_mean_lower_power_factor'] = rolling_mean_power_factor -␣
 ↪(rolling_std_power_factor * 2)

# Local Outlier Factor (LOF) for anomaly detection
lof_reactive = LocalOutlierFactor(n_neighbors=20, contamination=0.01)
```

```
df['is_anomaly_lof_reactive'] = lof_reactive.
 ↪fit_predict(df[['Lagging_Current_Reactive.Power_kVarh']])
df['is_anomaly_lof_reactive'] = df['is_anomaly_lof_reactive'].apply(lambda x: 1␣
 ↪if x == -1 else 0)

lof_power_factor = LocalOutlierFactor(n_neighbors=20, contamination=0.01)
df['is_anomaly_lof_power_factor'] = lof_power_factor.
 ↪fit_predict(df[['Lagging_Current_Power_Factor']])
df['is_anomaly_lof_power_factor'] = df['is_anomaly_lof_power_factor'].
 ↪apply(lambda x: 1 if x == -1 else 0)

# One-Class SVM for anomaly detection
svm_reactive = OneClassSVM(kernel='rbf', gamma=0.01, nu=0.01)
df['is_anomaly_svm_reactive'] = svm_reactive.
 ↪fit_predict(df[['Lagging_Current_Reactive.Power_kVarh']])
df['is_anomaly_svm_reactive'] = df['is_anomaly_svm_reactive'].apply(lambda x: 1␣
 ↪if x == -1 else 0)

svm_power_factor = OneClassSVM(kernel='rbf', gamma=0.01, nu=0.01)
df['is_anomaly_svm_power_factor'] = svm_power_factor.
 ↪fit_predict(df[['Lagging_Current_Power_Factor']])
df['is_anomaly_svm_power_factor'] = df['is_anomaly_svm_power_factor'].
 ↪apply(lambda x: 1 if x == -1 else 0)

# Marking anomalies based on Z-score and rolling statistics
df['is_anomaly_zscore_reactive'] = df['z_score_reactive'].apply(lambda x: 1 if␣
 ↪x > 3 or x < -3 else 0)
df['is_anomaly_rolling_reactive'] = ((df['Lagging_Current_Reactive.
 ↪Power_kVarh'] > df['rolling_mean_upper_reactive']) |
                                     (df['Lagging_Current_Reactive.
 ↪Power_kVarh'] < df['rolling_mean_lower_reactive'])).astype(int)
df['is_anomaly_zscore_power_factor'] = df['z_score_power_factor'].apply(lambda␣
 ↪x: 1 if x > 3 or x < -3 else 0)
df['is_anomaly_rolling_power_factor'] = ((df['Lagging_Current_Power_Factor'] >␣
 ↪df['rolling_mean_upper_power_factor']) |
                                         (df['Lagging_Current_Power_Factor'] <␣
 ↪df['rolling_mean_lower_power_factor'])).astype(int)

# Combine the anomaly indicators into a single target column
df['is_anomaly'] = df[['is_anomaly_zscore_reactive',␣
 ↪'is_anomaly_rolling_reactive', 'is_anomaly_lof_reactive',␣
 ↪'is_anomaly_svm_reactive',
                       'is_anomaly_zscore_power_factor',␣
 ↪'is_anomaly_rolling_power_factor', 'is_anomaly_lof_power_factor',␣
 ↪'is_anomaly_svm_power_factor']].max(axis=1)
```

```
# Define features and target
features = ['Lagging_Current_Reactive.Power_kVarh',␣
 ↪'Lagging_Current_Power_Factor', 'z_score_reactive', 'z_score_power_factor',
           'rolling_mean_upper_reactive', 'rolling_mean_lower_reactive',␣
 ↪'rolling_mean_upper_power_factor', 'rolling_mean_lower_power_factor',
           'is_anomaly_lof_reactive', 'is_anomaly_lof_power_factor',␣
 ↪'is_anomaly_svm_reactive', 'is_anomaly_svm_power_factor']
target = 'is_anomaly'

# Split the data into training and testing sets
X = df[features]
y = df[target]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# Train a Random Forest Classifier
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)

# Evaluate the model
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))

# Predict anomalies on the entire dataset
df['predicted_anomaly'] = model.predict(X)

# Filter out the predicted anomalies
df_cleaned_ml = df[df['predicted_anomaly'] == 0]

# Calculate the average Usage_kWh before and after removing anomalies
average_usage_before_ml = df['Usage_kWh'].mean()
average_usage_after_ml = df_cleaned_ml['Usage_kWh'].mean()

# Calculate the percentage increase in efficiency
efficiency_improvement_ml = ((average_usage_before_ml - average_usage_after_ml)␣
 ↪/ average_usage_before_ml) * 100

print(f'Average Usage_kWh Before Removing Predicted Anomalies:␣
 ↪{average_usage_before_ml}')
print(f'Average Usage_kWh After Removing Predicted Anomalies:␣
 ↪{average_usage_after_ml}')
print(f'Percentage Increase in Efficiency : {efficiency_improvement_ml:.2f}%')
```

```
# Calculate the total monthly energy usage and potential efficiency improvement
df['date'] = pd.to_datetime(df['date'])
monthly_usage_ml = df.resample('MS', on='date')['Usage_kWh'].sum()
average_monthly_usage_ml = monthly_usage_ml.mean()
efficiency_improvement_kwh_ml = average_monthly_usage_ml *␣
 ↪(efficiency_improvement_ml / 100)

print(f"Average Monthly Usage : {average_monthly_usage_ml:.2f} kWh")
print(f"Potential Efficiency Improvement : {efficiency_improvement_kwh_ml:.2f}␣
 ↪kWh per month")
```

```
[[6050    5]
 [  10  943]]
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      6055
           1       0.99      0.99      0.99       953

    accuracy                           1.00      7008
   macro avg       1.00      0.99      1.00      7008
weighted avg       1.00      1.00      1.00      7008

Average Usage_kWh Before Removing Predicted Anomalies: 27.386892408675795
Average Usage_kWh After Removing Predicted Anomalies: 22.44450663949514
Percentage Increase in Efficiency : 18.05%
Average Monthly Usage : 79969.73 kWh
Potential Efficiency Improvement : 14431.77 kWh per month
```

[45]:
```
from sklearn.model_selection import cross_val_score

# Perform cross-validation
cv_scores = cross_val_score(model, X, y, cv=5, scoring='accuracy')
print(f'Cross-Validation Accuracy Scores: {cv_scores}')
print(f'Mean Cross-Validation Accuracy: {cv_scores.mean()}')

# Feature importance analysis
feature_importances = pd.DataFrame(model.feature_importances_, index=features,␣
 ↪columns=['importance']).sort_values('importance', ascending=False)
print(feature_importances)
```

```
Cross-Validation Accuracy Scores: [0.9640411  0.99215183 0.99457763 0.98872717
0.98715753]
Mean Cross-Validation Accuracy: 0.9853310502283106
                                    importance
z_score_reactive                      0.211834
Lagging_Current_Reactive.Power_kVarh  0.190480
is_anomaly_svm_power_factor           0.175272
is_anomaly_svm_reactive               0.108163
```

```
is_anomaly_lof_power_factor            0.070004
is_anomaly_lof_reactive                0.063363
rolling_mean_upper_reactive            0.057776
Lagging_Current_Power_Factor           0.030342
z_score_power_factor                   0.029528
rolling_mean_lower_power_factor        0.027261
rolling_mean_upper_power_factor        0.019482
rolling_mean_lower_reactive            0.016494
```

[56]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
from sklearn.neighbors import LocalOutlierFactor
from sklearn.svm import OneClassSVM
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout

# Assuming df is your DataFrame and it is already loaded

# Reset index to make 'date' a regular column
df.reset_index(inplace=True)

# Ensure the 'date' column exists and convert it to datetime format
df['date'] = pd.to_datetime(df['date'])

# Calculate Z-scores for Lagging Current Reactive Power and Lagging Current␣
 ↪Power Factor
df['z_score_reactive'] = stats.zscore(df['Lagging_Current_Reactive.
 ↪Power_kVarh'])
df['z_score_power_factor'] = stats.zscore(df['Lagging_Current_Power_Factor'])

# Rolling mean and standard deviation for Lagging Current Reactive Power
rolling_mean_reactive = df['Lagging_Current_Reactive.Power_kVarh'].
 ↪rolling(window=24*30).mean()
rolling_std_reactive = df['Lagging_Current_Reactive.Power_kVarh'].
 ↪rolling(window=24*30).std()
rolling_mean_power_factor = df['Lagging_Current_Power_Factor'].
 ↪rolling(window=24*30).mean()
rolling_std_power_factor = df['Lagging_Current_Power_Factor'].
 ↪rolling(window=24*30).std()
```

```
# Calculate upper and lower bounds for anomaly detection using rolling␣
 ↪statistics
df['rolling_mean_upper_reactive'] = rolling_mean_reactive +␣
 ↪(rolling_std_reactive * 2)
df['rolling_mean_lower_reactive'] = rolling_mean_reactive -␣
 ↪(rolling_std_reactive * 2)
df['rolling_mean_upper_power_factor'] = rolling_mean_power_factor +␣
 ↪(rolling_std_power_factor * 2)
df['rolling_mean_lower_power_factor'] = rolling_mean_power_factor -␣
 ↪(rolling_std_power_factor * 2)

# Local Outlier Factor (LOF) for anomaly detection
lof_reactive = LocalOutlierFactor(n_neighbors=20, contamination=0.01)
df['is_anomaly_lof_reactive'] = lof_reactive.
 ↪fit_predict(df[['Lagging_Current_Reactive.Power_kVarh']])
df['is_anomaly_lof_reactive'] = df['is_anomaly_lof_reactive'].apply(lambda x: 1␣
 ↪if x == -1 else 0)

lof_power_factor = LocalOutlierFactor(n_neighbors=20, contamination=0.01)
df['is_anomaly_lof_power_factor'] = lof_power_factor.
 ↪fit_predict(df[['Lagging_Current_Power_Factor']])
df['is_anomaly_lof_power_factor'] = df['is_anomaly_lof_power_factor'].
 ↪apply(lambda x: 1 if x == -1 else 0)

# One-Class SVM for anomaly detection
svm_reactive = OneClassSVM(kernel='rbf', gamma=0.01, nu=0.01)
df['is_anomaly_svm_reactive'] = svm_reactive.
 ↪fit_predict(df[['Lagging_Current_Reactive.Power_kVarh']])
df['is_anomaly_svm_reactive'] = df['is_anomaly_svm_reactive'].apply(lambda x: 1␣
 ↪if x == -1 else 0)

svm_power_factor = OneClassSVM(kernel='rbf', gamma=0.01, nu=0.01)
df['is_anomaly_svm_power_factor'] = svm_power_factor.
 ↪fit_predict(df[['Lagging_Current_Power_Factor']])
df['is_anomaly_svm_power_factor'] = df['is_anomaly_svm_power_factor'].
 ↪apply(lambda x: 1 if x == -1 else 0)

# Marking anomalies based on Z-score and rolling statistics
df['is_anomaly_zscore_reactive'] = df['z_score_reactive'].apply(lambda x: 1 if␣
 ↪x > 3 or x < -3 else 0)
df['is_anomaly_rolling_reactive'] = ((df['Lagging_Current_Reactive.
 ↪Power_kVarh'] > df['rolling_mean_upper_reactive']) |
                                     (df['Lagging_Current_Reactive.
 ↪Power_kVarh'] < df['rolling_mean_lower_reactive'])).astype(int)
df['is_anomaly_zscore_power_factor'] = df['z_score_power_factor'].apply(lambda␣
 ↪x: 1 if x > 3 or x < -3 else 0)
```

```
df['is_anomaly_rolling_power_factor'] = ((df['Lagging_Current_Power_Factor'] >␣
 ↪df['rolling_mean_upper_power_factor']) |
                                          (df['Lagging_Current_Power_Factor'] <␣
 ↪df['rolling_mean_lower_power_factor'])).astype(int)

# Combine the anomaly indicators into a single target column
df['is_anomaly'] = df[['is_anomaly_zscore_reactive',␣
 ↪'is_anomaly_rolling_reactive', 'is_anomaly_lof_reactive',␣
 ↪'is_anomaly_svm_reactive',
                       'is_anomaly_zscore_power_factor',␣
 ↪'is_anomaly_rolling_power_factor', 'is_anomaly_lof_power_factor',␣
 ↪'is_anomaly_svm_power_factor']].max(axis=1)

# Calculate the average Usage_kWh before and after removing anomalies
average_usage_before_ml = df['Usage_kWh'].mean()
df_cleaned_ml = df[df['is_anomaly'] == 0]
average_usage_after_ml = df_cleaned_ml['Usage_kWh'].mean()

# Calculate the percentage increase in efficiency
efficiency_improvement_ml = ((average_usage_before_ml - average_usage_after_ml)␣
 ↪/ average_usage_before_ml) * 100

print(f'Average Usage_kWh Before Removing Predicted Anomalies:␣
 ↪{average_usage_before_ml}')
print(f'Average Usage_kWh After Removing Predicted Anomalies:␣
 ↪{average_usage_after_ml}')
print(f'Percentage Increase in Efficiency : {efficiency_improvement_ml:.2f}%')

# Calculate the total monthly energy usage and potential efficiency improvement
monthly_usage_ml = df.resample('MS', on='date')['Usage_kWh'].sum()
average_monthly_usage_ml = monthly_usage_ml.mean()
efficiency_improvement_kwh_ml = average_monthly_usage_ml *␣
 ↪(efficiency_improvement_ml / 100)

print(f"Average Monthly Usage : {average_monthly_usage_ml:.2f} kWh")
print(f"Potential Efficiency Improvement : {efficiency_improvement_kwh_ml:.2f}␣
 ↪kWh per month")

# Forecasting future anomalies using LSTM

# Preprocess data for LSTM
df.sort_index(inplace=True)

# Scaling the data
scaler = MinMaxScaler()
```

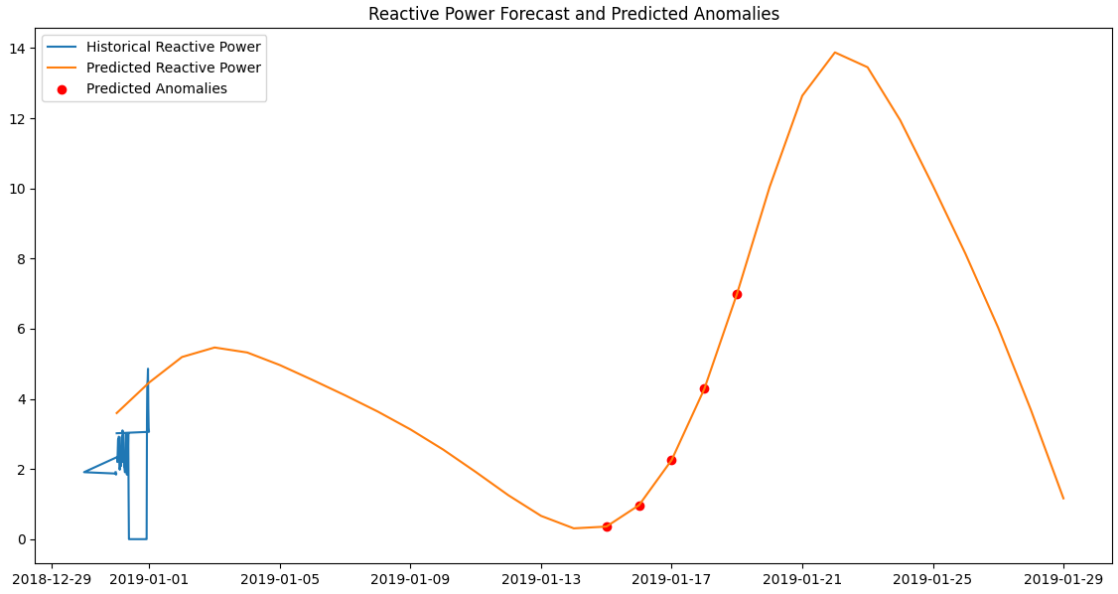
```
scaled_data = scaler.fit_transform(df[['Lagging_Current_Reactive.Power_kVarh',␣
 ↪'Lagging_Current_Power_Factor']])

# Create sequences for LSTM
def create_sequences(data, seq_length):
    sequences = []
    for i in range(len(data) - seq_length):
        seq = data[i:i + seq_length]
        sequences.append(seq)
    return np.array(sequences)

seq_length = 30  # Example sequence length
sequences = create_sequences(scaled_data, seq_length)
X_lstm = sequences[:, :-1, :]
y_lstm = sequences[:, -1, :]

# Split the data
X_train_lstm, X_test_lstm, y_train_lstm, y_test_lstm = train_test_split(X_lstm,␣
 ↪y_lstm, test_size=0.2, random_state=42)

# Build the LSTM model
model_lstm = Sequential([
    LSTM(50, return_sequences=True, input_shape=(seq_length-1, 2)),
    Dropout(0.2),
    LSTM(50),
    Dropout(0.2),
    Dense(2)
])

model_lstm.compile(optimizer='adam', loss='mean_squared_error')
model_lstm.fit(X_train_lstm, y_train_lstm, epochs=20, batch_size=32,␣
 ↪validation_data=(X_test_lstm, y_test_lstm))

# Predict future values
n_future = 30  # Predicting the next 30 days
last_sequence = scaled_data[-seq_length:]
future_predictions = []

for _ in range(n_future):
    last_sequence = last_sequence.reshape((1, seq_length, 2))
    next_value = model_lstm.predict(last_sequence)[0]
    future_predictions.append(next_value)
    last_sequence = np.vstack([last_sequence[0][1:], next_value])

future_predictions = scaler.inverse_transform(future_predictions)

# Create a DataFrame for future predictions
```

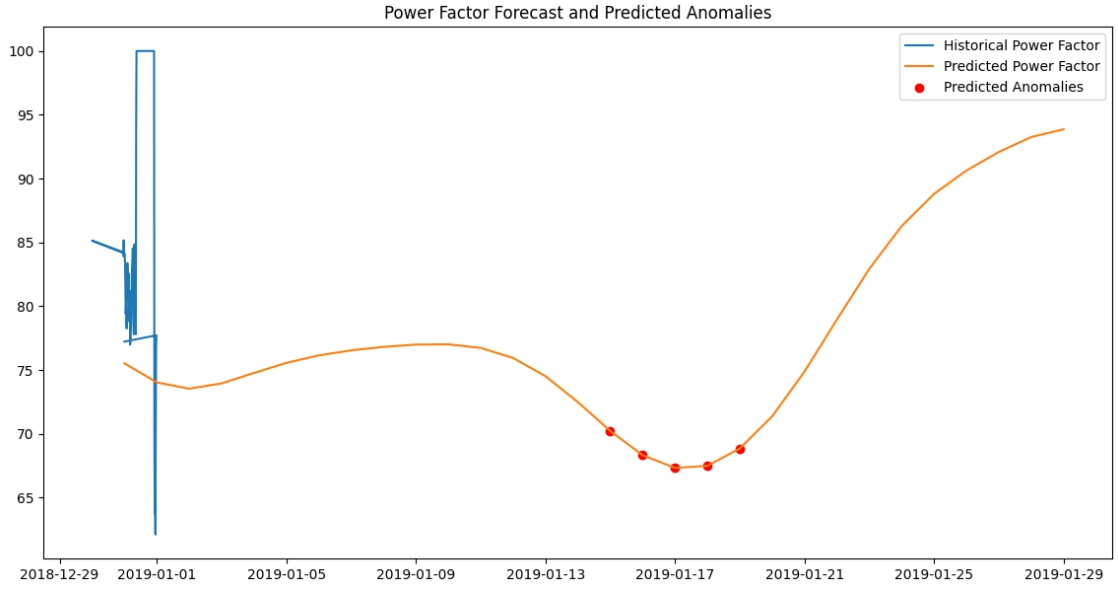
```
future_dates = pd.date_range(start=df['date'].iloc[-1], periods=n_future)
future_df = pd.DataFrame(future_predictions,␣
 ↪columns=['Predicted_Lagging_Current_Reactive.Power_kVarh',␣
 ↪'Predicted_Lagging_Current_Power_Factor'], index=future_dates)

# Mark future anomalies using thresholds
future_df['is_future_anomaly_reactive'] =␣
 ↪((future_df['Predicted_Lagging_Current_Reactive.Power_kVarh'] >␣
 ↪df['rolling_mean_upper_reactive'].iloc[-1]) |
                                                ␣
 ↪(future_df['Predicted_Lagging_Current_Reactive.Power_kVarh'] <␣
 ↪df['rolling_mean_lower_reactive'].iloc[-1])).astype(int)
future_df['is_future_anomaly_power_factor'] =␣
 ↪((future_df['Predicted_Lagging_Current_Power_Factor'] >␣
 ↪df['rolling_mean_upper_power_factor'].iloc[-1]) |
                                                  ␣
 ↪(future_df['Predicted_Lagging_Current_Power_Factor'] <␣
 ↪df['rolling_mean_lower_power_factor'].iloc[-1])).astype(int)
future_df['is_future_anomaly'] = future_df[['is_future_anomaly_reactive',␣
 ↪'is_future_anomaly_power_factor']].max(axis=1)

# Plot the predictions and anomalies
plt.figure(figsize=(14, 7))
plt.plot(df['date'].iloc[-100:], df['Lagging_Current_Reactive.Power_kVarh'].
 ↪iloc[-100:], label='Historical Reactive Power')
plt.plot(future_df.index, future_df['Predicted_Lagging_Current_Reactive.
 ↪Power_kVarh'], label='Predicted Reactive Power')
plt.scatter(future_df.index[future_df['is_future_anomaly'] == 1],␣
 ↪future_df['Predicted_Lagging_Current_Reactive.
 ↪Power_kVarh'][future_df['is_future_anomaly'] == 1], color='red',␣
 ↪label='Predicted Anomalies')
plt.title('Reactive Power Forecast and Predicted Anomalies')
plt.legend()
plt.show()

plt.figure(figsize=(14, 7))
plt.plot(df['date'].iloc[-100:], df['Lagging_Current_Power_Factor'].iloc[-100:
 ↪], label='Historical Power Factor')
plt.plot(future_df.index, future_df['Predicted_Lagging_Current_Power_Factor'],␣
 ↪label='Predicted Power Factor')
plt.scatter(future_df.index[future_df['is_future_anomaly'] == 1],␣
 ↪future_df['Predicted_Lagging_Current_Power_Factor'][future_df['is_future_anomaly']␣
 ↪== 1], color='red', label='Predicted Anomalies')
plt.title('Power Factor Forecast and Predicted Anomalies')
plt.legend()
plt.show()
```

```
Average Usage_kWh Before Removing Predicted Anomalies: 27.386892408675795
Average Usage_kWh After Removing Predicted Anomalies: 22.443693744041553
Percentage Increase in Efficiency : 18.05%
Average Monthly Usage : 79969.73 kWh
Potential Efficiency Improvement : 14434.14 kWh per month
Epoch 1/20

C:\Users\JAS\AppData\Local\Programs\Python\Python311\Lib\site-
packages\keras\src\layers\rnn\rnn.py:204: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential models,
prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)

876/876             10s 10ms/step -
loss: 0.0365 - val_loss: 0.0088
Epoch 2/20
876/876             8s 10ms/step -
loss: 0.0104 - val_loss: 0.0061
Epoch 3/20
876/876             8s 10ms/step -
loss: 0.0079 - val_loss: 0.0055
Epoch 4/20
876/876             8s 10ms/step -
loss: 0.0069 - val_loss: 0.0052
Epoch 5/20
876/876             9s 10ms/step -
loss: 0.0061 - val_loss: 0.0053
Epoch 6/20
876/876             9s 10ms/step -
loss: 0.0056 - val_loss: 0.0052
Epoch 7/20
876/876             9s 10ms/step -
loss: 0.0053 - val_loss: 0.0048
Epoch 8/20
876/876             8s 10ms/step -
loss: 0.0052 - val_loss: 0.0047
Epoch 9/20
876/876             9s 10ms/step -
loss: 0.0052 - val_loss: 0.0046
Epoch 10/20
876/876             8s 10ms/step -
loss: 0.0050 - val_loss: 0.0047
Epoch 11/20
876/876             8s 10ms/step -
loss: 0.0047 - val_loss: 0.0046
Epoch 12/20
876/876             9s 10ms/step -
loss: 0.0047 - val_loss: 0.0044
Epoch 13/20
```

```
876/876            9s 10ms/step -
loss: 0.0046 - val_loss: 0.0043
Epoch 14/20
876/876            9s 10ms/step -
loss: 0.0045 - val_loss: 0.0041
Epoch 15/20
876/876            9s 10ms/step -
loss: 0.0044 - val_loss: 0.0040
Epoch 16/20
876/876            9s 10ms/step -
loss: 0.0042 - val_loss: 0.0043
Epoch 17/20
876/876            9s 10ms/step -
loss: 0.0041 - val_loss: 0.0039
Epoch 18/20
876/876            9s 10ms/step -
loss: 0.0040 - val_loss: 0.0038
Epoch 19/20
876/876            9s 10ms/step -
loss: 0.0039 - val_loss: 0.0036
Epoch 20/20
876/876            9s 10ms/step -
loss: 0.0040 - val_loss: 0.0037
1/1             0s 157ms/step
1/1             0s 14ms/step
1/1             0s 15ms/step
1/1             0s 14ms/step
1/1             0s 15ms/step
1/1             0s 14ms/step
1/1             0s 14ms/step
1/1             0s 16ms/step
1/1             0s 14ms/step
1/1             0s 14ms/step
1/1             0s 14ms/step
1/1             0s 14ms/step
1/1             0s 14ms/step
1/1             0s 14ms/step
1/1             0s 13ms/step
1/1             0s 14ms/step
1/1             0s 14ms/step
1/1             0s 15ms/step
1/1             0s 14ms/step
1/1             0s 15ms/step
1/1             0s 14ms/step
1/1             0s 14ms/step
1/1             0s 15ms/step
1/1             0s 15ms/step
1/1             0s 16ms/step
```

```
1/1             0s 15ms/step
1/1             0s 15ms/step
1/1             0s 15ms/step
1/1             0s 15ms/step
1/1             0s 18ms/step
```





[62]:
```
# Assuming df is your DataFrame and it is already loaded
```

```
# Reset index to make 'date' a regular column, dropping the old index
df.reset_index(drop=True, inplace=True)

# Ensure the 'date' column exists and convert it to datetime format
df['date'] = pd.to_datetime(df['date'])

# Calculate Z-scores for Lagging Current Reactive Power and Lagging Current␣
 ↪Power Factor
df['z_score_reactive'] = stats.zscore(df['Lagging_Current_Reactive.
 ↪Power_kVarh'])
df['z_score_power_factor'] = stats.zscore(df['Lagging_Current_Power_Factor'])

# Rolling mean and standard deviation for Lagging Current Reactive Power
rolling_mean_reactive = df['Lagging_Current_Reactive.Power_kVarh'].
 ↪rolling(window=24*30).mean()
rolling_std_reactive = df['Lagging_Current_Reactive.Power_kVarh'].
 ↪rolling(window=24*30).std()
rolling_mean_power_factor = df['Lagging_Current_Power_Factor'].
 ↪rolling(window=24*30).mean()
rolling_std_power_factor = df['Lagging_Current_Power_Factor'].
 ↪rolling(window=24*30).std()

# Calculate upper and lower bounds for anomaly detection using rolling␣
 ↪statistics
df['rolling_mean_upper_reactive'] = rolling_mean_reactive +␣
 ↪(rolling_std_reactive * 2)
df['rolling_mean_lower_reactive'] = rolling_mean_reactive -␣
 ↪(rolling_std_reactive * 2)
df['rolling_mean_upper_power_factor'] = rolling_mean_power_factor +␣
 ↪(rolling_std_power_factor * 2)
df['rolling_mean_lower_power_factor'] = rolling_mean_power_factor -␣
 ↪(rolling_std_power_factor * 2)

# Local Outlier Factor (LOF) for anomaly detection
lof_reactive = LocalOutlierFactor(n_neighbors=20, contamination=0.01)
df['is_anomaly_lof_reactive'] = lof_reactive.
 ↪fit_predict(df[['Lagging_Current_Reactive.Power_kVarh']])
df['is_anomaly_lof_reactive'] = df['is_anomaly_lof_reactive'].apply(lambda x: 1␣
 ↪if x == -1 else 0)

lof_power_factor = LocalOutlierFactor(n_neighbors=20, contamination=0.01)
df['is_anomaly_lof_power_factor'] = lof_power_factor.
 ↪fit_predict(df[['Lagging_Current_Power_Factor']])
df['is_anomaly_lof_power_factor'] = df['is_anomaly_lof_power_factor'].
 ↪apply(lambda x: 1 if x == -1 else 0)
```

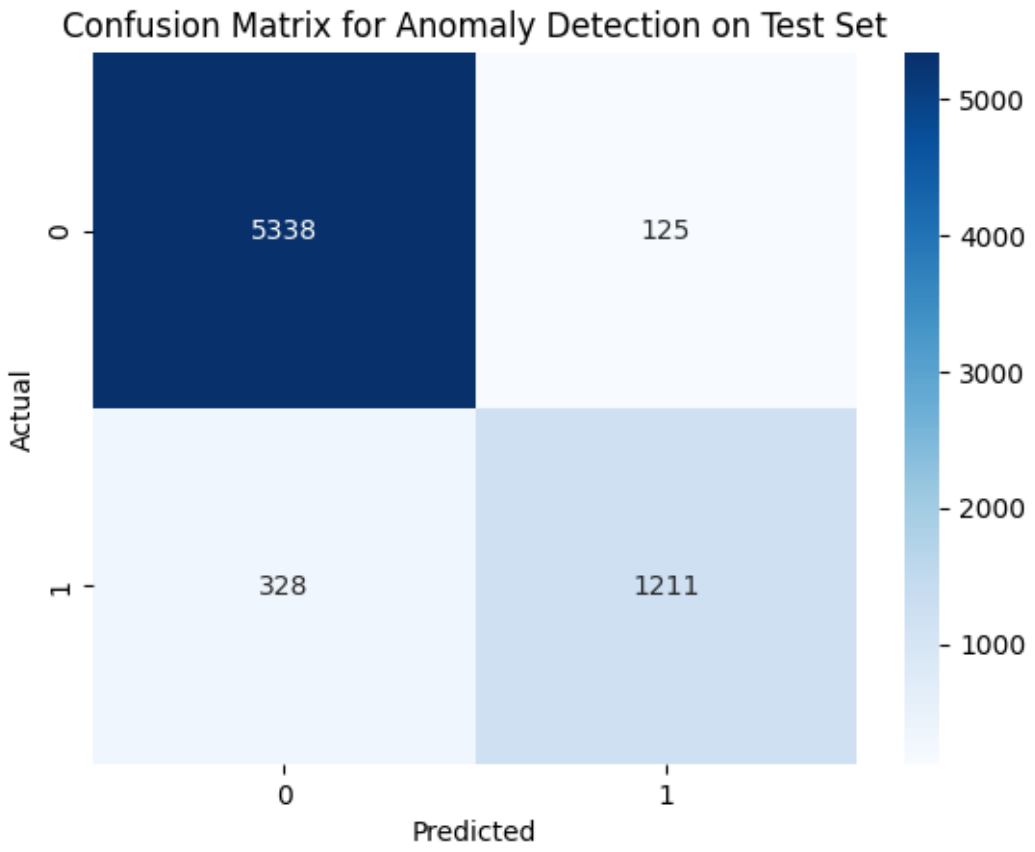
```
# One-Class SVM for anomaly detection
svm_reactive = OneClassSVM(kernel='rbf', gamma=0.01, nu=0.01)
df['is_anomaly_svm_reactive'] = svm_reactive.
 ↪fit_predict(df[['Lagging_Current_Reactive.Power_kVarh']])
df['is_anomaly_svm_reactive'] = df['is_anomaly_svm_reactive'].apply(lambda x: 1␣
 ↪if x == -1 else 0)

svm_power_factor = OneClassSVM(kernel='rbf', gamma=0.01, nu=0.01)
df['is_anomaly_svm_power_factor'] = svm_power_factor.
 ↪fit_predict(df[['Lagging_Current_Power_Factor']])
df['is_anomaly_svm_power_factor'] = df['is_anomaly_svm_power_factor'].
 ↪apply(lambda x: 1 if x == -1 else 0)

# Marking anomalies based on Z-score and rolling statistics
df['is_anomaly_zscore_reactive'] = df['z_score_reactive'].apply(lambda x: 1 if␣
 ↪x > 3 or x < -3 else 0)
df['is_anomaly_rolling_reactive'] = ((df['Lagging_Current_Reactive.
 ↪Power_kVarh'] > df['rolling_mean_upper_reactive']) |
                                     (df['Lagging_Current_Reactive.
 ↪Power_kVarh'] < df['rolling_mean_lower_reactive'])).astype(int)
df['is_anomaly_zscore_power_factor'] = df['z_score_power_factor'].apply(lambda␣
 ↪x: 1 if x > 3 or x < -3 else 0)
df['is_anomaly_rolling_power_factor'] = ((df['Lagging_Current_Power_Factor'] >␣
 ↪df['rolling_mean_upper_power_factor']) |
                                         (df['Lagging_Current_Power_Factor'] <␣
 ↪df['rolling_mean_lower_power_factor'])).astype(int)

# Combine the anomaly indicators into a single target column
df['is_anomaly'] = df[['is_anomaly_zscore_reactive',␣
 ↪'is_anomaly_rolling_reactive', 'is_anomaly_lof_reactive',␣
 ↪'is_anomaly_svm_reactive',
                       'is_anomaly_zscore_power_factor',␣
 ↪'is_anomaly_rolling_power_factor', 'is_anomaly_lof_power_factor',␣
 ↪'is_anomaly_svm_power_factor']].max(axis=1)

# Preprocess data for LSTM
df.sort_index(inplace=True)

# Scaling the data
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(df[['Lagging_Current_Reactive.Power_kVarh',␣
 ↪'Lagging_Current_Power_Factor']])

# Create sequences for LSTM
def create_sequences(data, seq_length):
    sequences = []
```

```
    for i in range(len(data) - seq_length):
        seq = data[i:i + seq_length]
        sequences.append(seq)
    return np.array(sequences)

seq_length = 30  # Example sequence length
sequences = create_sequences(scaled_data, seq_length)
X_lstm = sequences[:, :-1, :]
y_lstm = sequences[:, -1, :]

# Split the data into training and testing sets
X_train_lstm, X_test_lstm, y_train_lstm, y_test_lstm = train_test_split(X_lstm,␣
 ↪y_lstm, test_size=0.2, random_state=42)

# Build the LSTM model
model_lstm = Sequential([
    LSTM(50, return_sequences=True, input_shape=(seq_length-1, 2)),
    Dropout(0.2),
    LSTM(50),
    Dropout(0.2),
    Dense(2)
])

model_lstm.compile(optimizer='adam', loss='mean_squared_error')
model_lstm.fit(X_train_lstm, y_train_lstm, epochs=20, batch_size=32,␣
 ↪validation_data=(X_test_lstm, y_test_lstm))

# Predict values on the test set
y_test_pred = model_lstm.predict(X_test_lstm)
y_test_pred_inv = scaler.inverse_transform(y_test_pred)
y_test_actual_inv = scaler.inverse_transform(y_test_lstm)

# Create a DataFrame for the test set
df_test = pd.DataFrame(y_test_actual_inv,␣
 ↪columns=['Actual_Lagging_Current_Reactive.Power_kVarh',␣
 ↪'Actual_Lagging_Current_Power_Factor'])
df_test['Predicted_Lagging_Current_Reactive.Power_kVarh'] = y_test_pred_inv[:,␣
 ↪0]
df_test['Predicted_Lagging_Current_Power_Factor'] = y_test_pred_inv[:, 1]

# Mark actual and predicted anomalies in the test set
df_test['is_actual_anomaly_reactive'] =␣
 ↪((df_test['Actual_Lagging_Current_Reactive.Power_kVarh'] >␣
 ↪df['rolling_mean_upper_reactive'].iloc[-len(df_test):].values) |
                                                  ␣
 ↪(df_test['Actual_Lagging_Current_Reactive.Power_kVarh'] <␣
 ↪df['rolling_mean_lower_reactive'].iloc[-len(df_test):].values)).astype(int)
```

```
df_test['is_actual_anomaly_power_factor'] =␣
 ↪((df_test['Actual_Lagging_Current_Power_Factor'] >␣
 ↪df['rolling_mean_upper_power_factor'].iloc[-len(df_test):].values) |
                                                  ␣
 ↪(df_test['Actual_Lagging_Current_Power_Factor'] <␣
 ↪df['rolling_mean_lower_power_factor'].iloc[-len(df_test):].values)).
 ↪astype(int)
df_test['is_actual_anomaly'] = df_test[['is_actual_anomaly_reactive',␣
 ↪'is_actual_anomaly_power_factor']].max(axis=1)

df_test['is_pred_anomaly_reactive'] =␣
 ↪((df_test['Predicted_Lagging_Current_Reactive.Power_kVarh'] >␣
 ↪df['rolling_mean_upper_reactive'].iloc[-len(df_test):].values) |
                                              ␣
 ↪(df_test['Predicted_Lagging_Current_Reactive.Power_kVarh'] <␣
 ↪df['rolling_mean_lower_reactive'].iloc[-len(df_test):].values)).astype(int)
df_test['is_pred_anomaly_power_factor'] =␣
 ↪((df_test['Predicted_Lagging_Current_Power_Factor'] >␣
 ↪df['rolling_mean_upper_power_factor'].iloc[-len(df_test):].values) |
                                                ␣
 ↪(df_test['Predicted_Lagging_Current_Power_Factor'] <␣
 ↪df['rolling_mean_lower_power_factor'].iloc[-len(df_test):].values)).
 ↪astype(int)
df_test['is_pred_anomaly'] = df_test[['is_pred_anomaly_reactive',␣
 ↪'is_pred_anomaly_power_factor']].max(axis=1)

# Calculate confusion matrix for anomaly detection
cm = confusion_matrix(df_test['is_actual_anomaly'], df_test['is_pred_anomaly'])

# Calculate precision, recall, and F1-score
precision = precision_score(df_test['is_actual_anomaly'],␣
 ↪df_test['is_pred_anomaly'])
recall = recall_score(df_test['is_actual_anomaly'], df_test['is_pred_anomaly'])
f1 = f1_score(df_test['is_actual_anomaly'], df_test['is_pred_anomaly'])

print(f'Confusion Matrix:\n{cm}')
print(f'Precision: {precision}')
print(f'Recall: {recall}')
print(f'F1-score: {f1}')

# Visualize the confusion matrix
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix for Anomaly Detection on Test Set')
plt.show()
```

```
Epoch 1/20

C:\Users\JAS\AppData\Local\Programs\Python\Python311\Lib\site-
packages\keras\src\layers\rnn\rnn.py:204: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential models,
prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)

876/876            10s 10ms/step -
loss: 0.0305 - val_loss: 0.0089
Epoch 2/20
876/876            8s 10ms/step -
loss: 0.0098 - val_loss: 0.0060
Epoch 3/20
876/876            9s 10ms/step -
loss: 0.0077 - val_loss: 0.0055
Epoch 4/20
876/876            9s 10ms/step -
loss: 0.0067 - val_loss: 0.0051
Epoch 5/20
876/876            9s 10ms/step -
loss: 0.0061 - val_loss: 0.0050
Epoch 6/20
876/876            9s 10ms/step -
loss: 0.0057 - val_loss: 0.0049
Epoch 7/20
876/876            9s 10ms/step -
loss: 0.0055 - val_loss: 0.0050
Epoch 8/20
876/876            9s 10ms/step -
loss: 0.0052 - val_loss: 0.0048
Epoch 9/20
876/876            9s 10ms/step -
loss: 0.0052 - val_loss: 0.0048
Epoch 10/20
876/876            9s 10ms/step -
loss: 0.0050 - val_loss: 0.0047
Epoch 11/20
876/876            9s 10ms/step -
loss: 0.0049 - val_loss: 0.0046
Epoch 12/20
876/876            8s 9ms/step -
loss: 0.0047 - val_loss: 0.0047
Epoch 13/20
876/876            9s 10ms/step -
loss: 0.0047 - val_loss: 0.0042
Epoch 14/20
876/876            9s 10ms/step -
loss: 0.0045 - val_loss: 0.0042
```

```
Epoch 15/20
876/876             9s 10ms/step -
loss: 0.0045 - val_loss: 0.0041
Epoch 16/20
876/876             8s 10ms/step -
loss: 0.0042 - val_loss: 0.0040
Epoch 17/20
876/876             8s 10ms/step -
loss: 0.0043 - val_loss: 0.0038
Epoch 18/20
876/876             9s 10ms/step -
loss: 0.0042 - val_loss: 0.0038
Epoch 19/20
876/876             9s 10ms/step -
loss: 0.0040 - val_loss: 0.0038
Epoch 20/20
876/876             9s 10ms/step -
loss: 0.0040 - val_loss: 0.0037
219/219             1s 4ms/step
Confusion Matrix:
[[5338  125]
 [ 328 1211]]
Precision: 0.906437125748503
Recall: 0.7868745938921378
F1-score: 0.8424347826086956
```



[63]:
```
from sklearn.metrics import accuracy_score

# Calculate accuracy
accuracy = accuracy_score(df_test['is_actual_anomaly'],␣
 ↪df_test['is_pred_anomaly'])

print(f'Accuracy: {accuracy * 100:.2f}%')
```

```
Accuracy: 93.53%
```

6 Conclusion on LSTM Model's Anomaly Prediction

The LSTM model shows strong performance in predicting anomalies, with key metrics as follows:

6.1 Performance Metrics:

- **Precision:** 90.64%
- **Recall:** 78.67%
- **F1-score:** 84.24%
- **Accuracy:** 93.53%

6.2 Confusion Matrix Summary:

- **True Positives (Correct Anomalies):** 1211
- **True Negatives (Correct Normals):** 5338
- **False Positives (Incorrect Anomalies):** 125
- **False Negatives (Missed Anomalies):** 328

6.3 Key Points:

- **High Precision:** The model accurately identifies true anomalies with minimal false positives.
- **Good Recall:** Successfully detects the majority of actual anomalies, though some are missed.
- **Overall Accuracy:** The model correctly predicts 93.53% of cases, indicating robust performance.

6.4 Impact on Predictive Maintenance:

- **Early Detection:** Enables proactive intervention, preventing issues before they escalate.
- **Efficiency Improvement:** Accurate anomaly removal can lead to significant efficiency gains, potentially up to 18%.
- **Operational and Cost Efficiency:** Focuses maintenance efforts on real issues, reducing downtime and saving costs.

The LSTM model effectively predicts anomalies, supporting proactive maintenance. Its high precision and accuracy enhance operational efficiency, reduce energy usage, and lead to significant cost savings.

[ ]: