You've uploaded a file named steel+industry+energy+consumption.zip. I'll start by examining its contents to identify the dataset and understand its structure. Then, I'll provide detailed steps for data filtering, outlier detection, and the requested interpolation and smoothing techniques, along with explanations and code.

Let me extract and inspect the contents of the uploaded file.

The uploaded ZIP file contains a dataset named Steel\_industry\_data.csv. I'll load this file to inspect its structure and contents before proceeding with the requested analyses.

The dataset contains 35,040 rows and 11 columns. Here's a summary of its columns:

1. **date**: Timestamp.
2. **Usage\_kWh**: Energy usage in kWh.
3. **Lagging\_Current\_Reactive.Power\_kVarh**: Lagging reactive power in kVarh.
4. **Leading\_Current\_Reactive\_Power\_kVarh**: Leading reactive power in kVarh.
5. **CO2(tCO2)**: CO₂ emissions in tons.
6. **Lagging\_Current\_Power\_Factor**: Lagging power factor.
7. **Leading\_Current\_Power\_Factor**: Leading power factor.
8. **NSM**: Number of seconds from midnight.
9. **WeekStatus**: Whether the day is a weekday or weekend.
10. **Day\_of\_week**: Day of the week.
11. **Load\_Type**: Type of load (e.g., Light\_Load).

**Approach**

We'll conduct the following steps:

1. **Data Filtering**: Identify and handle missing values and anomalies.
2. **Outlier Detection**:
   * Z-score method
   * Interquartile Range (IQR)
3. **Interpolation**:
   * Kriging
   * Spline interpolation
4. **Smoothing**:
   * Moving average
   * Exponential smoothing

Let's start with basic data filtering.

**Observations:**

1. **No Missing Values**: All columns are complete.
2. **Numerical Distribution**:
   * The Usage\_kWh column has values ranging from 0 to 157.18, with a mean of 27.39 and a high standard deviation, indicating potential outliers.
   * Other columns like Lagging\_Current\_Reactive.Power\_kVarh and Leading\_Current\_Reactive\_Power\_kVarh also show significant variability.

**Next Steps:**

We will identify and handle outliers using the **Z-score** and **IQR** methods for the Usage\_kWh column as an example, then extend it to others if needed.

**Step 1: Outlier Detection Using Z-Score**

The Z-score indicates how many standard deviations a data point is from the mean. Data points with Z-scores beyond a threshold (commonly ±3) are considered outliers. Let's calculate it.

It seems like I can’t do more advanced data analysis right now. Please try again later. If you want, I can provide you with Python code to run locally for the analysis, or we can revisit this once the issue is resolved. Let me know how you'd like to proceed!

import zipfile

import os

# Define the file path and extraction directory

zip\_file\_path = '/mnt/data/steel+industry+energy+consumption.zip'

extraction\_dir = '/mnt/data/steel\_industry\_energy\_consumption'

# Extract the zip file

with zipfile.ZipFile(zip\_file\_path, 'r') as zip\_ref:

zip\_ref.extractall(extraction\_dir)

# List the extracted files

os.listdir(extraction\_dir)

import pandas as pd

# Load the dataset

file\_path = os.path.join(extraction\_dir, 'Steel\_industry\_data.csv')

data = pd.read\_csv(file\_path)

# Display basic information about the dataset and the first few rows

data.info(), data.head()

# Check for missing values

missing\_values = data.isnull().sum()

# Identify numerical columns for further analysis

numerical\_cols = data.select\_dtypes(include=['float64', 'int64']).columns

# Basic statistics for numerical columns to understand data distribution

basic\_stats = data[numerical\_cols].describe()

missing\_values, basic\_stats

from scipy.stats import zscore

# Calculate Z-scores for the `Usage\_kWh` column

data['Z\_Score\_Usage\_kWh'] = zscore(data['Usage\_kWh'])

# Identify outliers where Z-score > 3 or < -3

z\_score\_outliers = data[(data['Z\_Score\_Usage\_kWh'] > 3) | (data['Z\_Score\_Usage\_kWh'] < -3)]

# Percentage of outliers

z\_score\_outlier\_percentage = len(z\_score\_outliers) / len(data) \* 100

z\_score\_outlier\_percentage, z\_score\_outliers[['date', 'Usage\_kWh', 'Z\_Score\_Usage\_kWh']].head()

Validation class in data diltering

A **Validation class** in data filtering typically refers to a structured way of validating the integrity, consistency, and quality of data using a Python class. This approach encapsulates common validation operations, making it reusable and modular. Below is an explanation of how such a class works and its implementation:

**Key Functions of a Validation Class:**

1. **Schema Validation**: Ensure the dataset has expected columns and data types.
2. **Missing Value Check**: Detect and handle missing values.
3. **Range Validation**: Verify that numerical columns fall within acceptable ranges.
4. **Categorical Validation**: Confirm that categorical columns contain only expected categories.
5. **Custom Rules**: Allow for user-defined validation logic, such as checking for duplicate records or applying domain-specific rules.

**Example Implementation**

import pandas as pd

import numpy as np

class DataValidator:

def \_\_init\_\_(self, df):

"""

Initialize the validator with a DataFrame.

"""

self.df = df

self.errors = []

def check\_columns(self, expected\_columns):

"""

Validate if all expected columns are present.

"""

missing\_columns = [col for col in expected\_columns if col not in self.df.columns]

if missing\_columns:

self.errors.append(f"Missing columns: {missing\_columns}")

def check\_data\_types(self, column\_types):

"""

Validate data types of specified columns.

"""

for col, expected\_type in column\_types.items():

if col in self.df.columns:

if not np.issubdtype(self.df[col].dtype, expected\_type):

self.errors.append(f"Column {col} has wrong type. Expected {expected\_type}, found {self.df[col].dtype}")

else:

self.errors.append(f"Column {col} is missing.")

def check\_missing\_values(self, columns=None):

"""

Check for missing values in specified columns or all columns.

"""

columns\_to\_check = columns if columns else self.df.columns

for col in columns\_to\_check:

if self.df[col].isnull().any():

self.errors.append(f"Column {col} contains missing values.")

def check\_value\_ranges(self, column\_ranges):

"""

Validate if column values fall within specified ranges.

"""

for col, (min\_val, max\_val) in column\_ranges.items():

if col in self.df.columns:

if (self.df[col] < min\_val).any() or (self.df[col] > max\_val).any():

self.errors.append(f"Column {col} contains values outside the range {min\_val}-{max\_val}")

def check\_categories(self, column\_categories):

"""

Validate categorical columns against a set of expected categories.

"""

for col, expected\_categories in column\_categories.items():

if col in self.df.columns:

invalid\_categories = set(self.df[col]) - set(expected\_categories)

if invalid\_categories:

self.errors.append(f"Column {col} contains unexpected categories: {invalid\_categories}")

def validate(self):

"""

Display validation errors or confirm that the data is valid.

"""

if self.errors:

print("Validation Errors:")

for error in self.errors:

print(f" - {error}")

else:

print("Data is valid.")

# Example Usage

# Assuming `data` is the DataFrame we are validating

validator = DataValidator(data)

# Define validation rules

expected\_columns = ['Usage\_kWh', 'Lagging\_Current\_Reactive.Power\_kVarh', 'WeekStatus']

column\_types = {'Usage\_kWh': np.float64, 'WeekStatus': np.object\_}

column\_ranges = {'Usage\_kWh': (0, 200)}

column\_categories = {'WeekStatus': ['Weekday', 'Weekend']}

# Run validations

validator.check\_columns(expected\_columns)

validator.check\_data\_types(column\_types)

validator.check\_missing\_values()

validator.check\_value\_ranges(column\_ranges)

validator.check\_categories(column\_categories)

validator.validate()

**Steps in the Code:**

1. **Initialization**: The class is initialized with a DataFrame.
2. **Validation Methods**: Methods like check\_columns, check\_data\_types, and check\_missing\_values perform individual checks.
3. **Custom Rules**: Additional checks like range and category validation are included.
4. **Error Reporting**: Errors are collected in a list and displayed after all validations.

**Benefits:**

* Modular and reusable.
* Ensures data quality systematically.
* Easily extendable for additional validation rules.

Would you like me to customize this for your dataset?

import pandas as pd

import numpy as np

class DataValidator:

def \_\_init\_\_(self, df):

"""

Initialize the validator with a DataFrame.

"""

self.df = df

self.errors = []

def check\_columns(self, expected\_columns):

"""

Validate if all expected columns are present.

"""

missing\_columns = [col for col in expected\_columns if col not in self.df.columns]

if missing\_columns:

self.errors.append(f"Missing columns: {missing\_columns}")

def check\_data\_types(self, column\_types):

"""

Validate data types of specified columns.

"""

for col, expected\_type in column\_types.items():

if col in self.df.columns:

if not np.issubdtype(self.df[col].dtype, expected\_type):

self.errors.append(f"Column {col} has wrong type. Expected {expected\_type}, found {self.df[col].dtype}")

else:

self.errors.append(f"Column {col} is missing.")

def check\_missing\_values(self, columns=None):

"""

Check for missing values in specified columns or all columns.

"""

columns\_to\_check = columns if columns else self.df.columns

for col in columns\_to\_check:

if self.df[col].isnull().any():

self.errors.append(f"Column {col} contains missing values.")

def check\_value\_ranges(self, column\_ranges):

"""

Validate if column values fall within specified ranges.

"""

for col, (min\_val, max\_val) in column\_ranges.items():

if col in self.df.columns:

if (self.df[col] < min\_val).any() or (self.df[col] > max\_val).any():

self.errors.append(f"Column {col} contains values outside the range {min\_val}-{max\_val}")

def check\_categories(self, column\_categories):

"""

Validate categorical columns against a set of expected categories.

"""

for col, expected\_categories in column\_categories.items():

if col in self.df.columns:

invalid\_categories = set(self.df[col]) - set(expected\_categories)

if invalid\_categories:

self.errors.append(f"Column {col} contains unexpected categories: {invalid\_categories}")

def validate(self):

"""

Display validation errors or confirm that the data is valid.

"""

if self.errors:

print("Validation Errors:")

for error in self.errors:

print(f" - {error}")

else:

print("Data is valid.")

# Example Usage

# Assuming `data` is the DataFrame we are validating

validator = DataValidator(data)

# Define validation rules

expected\_columns = ['Usage\_kWh', 'Lagging\_Current\_Reactive.Power\_kVarh', 'WeekStatus']

column\_types = {'Usage\_kWh': np.float64, 'WeekStatus': np.object\_}

column\_ranges = {'Usage\_kWh': (0, 200)}

column\_categories = {'WeekStatus': ['Weekday', 'Weekend']}

# Run validations

validator.check\_columns(expected\_columns)

validator.check\_data\_types(column\_types)

validator.check\_missing\_values()

validator.check\_value\_ranges(column\_ranges)

validator.check\_categories(column\_categories)

validator.validate()

How to handle outliers?

After detecting outliers in a dataset, you need to decide how to handle them. This decision depends on the nature of the dataset, the domain context, and the goals of your analysis or model. Here are the common steps and options available after detecting outliers:

**1. Understand the Outliers**

* **Explore Causes**:
  + Are the outliers legitimate (e.g., rare but valid values)?
  + Could they be errors or noise (e.g., data entry mistakes or sensor failures)?
* **Visualize**:
  + Use box plots, scatter plots, or histograms to better understand the distribution.

**2. Decide on Handling Strategy**

The appropriate handling strategy depends on whether the outliers are:

* **True Outliers**: Valid but rare observations.
* **Error or Noise**: Data points likely resulting from inaccuracies.

**Options for Handling Outliers**

**a. Remove Outliers**

* **When to Remove**:
  + The outliers are errors or irrelevant to the analysis.
  + The dataset is large, and removing a few points won't impact generalizability.
* **How**:
* # Example using Z-score threshold
* df = df[(df['Z\_Score\_Column'] >= -3) & (df['Z\_Score\_Column'] <= 3)]

**b. Transform Data**

* Apply transformations to reduce the impact of outliers:
  + **Log Transformation**: Compresses larger values.
  + df['column'] = np.log1p(df['column'])
  + **Square Root Transformation**: Reduces skewness.
  + df['column'] = np.sqrt(df['column'])

**c. Impute Outliers**

* Replace outliers with a more reasonable value:
  + **Mean/Median Imputation**:
  + median = df['column'].median()
  + df.loc[df['column'] > threshold, 'column'] = median
  + **Boundary Imputation**: Replace outliers with the upper or lower boundary of the data.
  + Q1 = df['column'].quantile(0.25)
  + Q3 = df['column'].quantile(0.75)
  + IQR = Q3 - Q1
  + lower\_bound = Q1 - 1.5 \* IQR
  + upper\_bound = Q3 + 1.5 \* IQR
  + df['column'] = np.clip(df['column'], lower\_bound, upper\_bound)

**d. Create a Separate Category (Categorical Data)**

* For categorical data, label outliers as a special category.
* df['column'] = df['column'].apply(lambda x: 'Outlier' if x not in valid\_categories else x)

**e. Treat Outliers Differently in Modeling**

* Some models (e.g., tree-based models) are robust to outliers and may not require handling.
* For others (e.g., linear regression), you may:
  + Use robust loss functions or methods like Huber regression.
  + Normalize or scale the data to reduce outlier influence.

**3. Document Your Decisions**

* Clearly document how you identified and handled outliers.
* Justify your approach based on domain knowledge and data goals.

**4. Reassess the Dataset**

After handling outliers:

* Recalculate statistics (mean, std, etc.).
* Re-visualize distributions to confirm improved data quality.

**Example Workflow**

from scipy.stats import zscore

import numpy as np

# Step 1: Detect outliers using Z-score

df['Z\_Score'] = zscore(df['column'])

outliers = df[(df['Z\_Score'] > 3) | (df['Z\_Score'] < -3)]

# Step 2: Decide what to do with outliers

# Option: Remove outliers

df = df[(df['Z\_Score'] <= 3) & (df['Z\_Score'] >= -3)]

# Option: Impute outliers with median

median = df['column'].median()

df.loc[outliers.index, 'column'] = median

# Step 3: Reassess

print(df.describe())

Handling outliers effectively ensures that your data is clean, models are robust, and results are reliable. Let me know if you'd like specific handling techniques applied to your dataset!

To start **data filtering** on a dataset, follow a structured process that ensures you clean, prepare, and filter the data effectively. Here's a step-by-step guide:

**1. Load and Inspect the Dataset**

Before any filtering, load your dataset and understand its structure and content.

**Steps:**

1. **Import the dataset**:
2. import pandas as pd
3. # Load dataset
4. df = pd.read\_csv('your\_dataset.csv') # Replace with your file
5. **Inspect the dataset**:
6. # View first few rows
7. print(df.head())
8. # Get structure, data types, and memory usage
9. print(df.info())
10. # View summary statistics
11. print(df.describe())

**2. Understand the Data**

Spend time understanding:

* The purpose of the dataset.
* Column meanings (e.g., numerical, categorical, text).
* Potential ranges or logical constraints for numerical data.

**Questions to Ask:**

* What is the goal of this dataset?
* Are there columns irrelevant to the goal?
* Are there apparent errors or missing values?

**3. Detect Missing Values**

Missing values can disrupt analysis, so check for them early.

**Steps:**

1. Identify missing values:
2. # Count missing values in each column
3. print(df.isnull().sum())
4. Visualize missing data (optional):
5. import seaborn as sns
6. import matplotlib.pyplot as plt
7. sns.heatmap(df.isnull(), cbar=False, cmap='viridis')
8. plt.show()

**4. Handle Missing Values**

Depending on the context, you can:

* Remove rows/columns with excessive missing values.
* Impute missing values with:
  + **Mean/Median/Mode** for numerical or categorical data.
  + **Domain-specific constants**.

**Example:**

# Fill missing numerical values with the median

df['column\_name'] = df['column\_name'].fillna(df['column\_name'].median())

# Drop rows with missing values

df = df.dropna()

**5. Detect and Remove Duplicates**

Duplicates can skew analysis.

**Steps:**

1. Check for duplicates:
2. print(df.duplicated().sum())
3. Remove duplicates:
4. df = df.drop\_duplicates()

**6. Filter Based on Logical Ranges**

Ensure numerical data falls within logical or expected ranges.

**Example:**

# Remove rows where 'age' is outside logical range

df = df[(df['age'] >= 0) & (df['age'] <= 120)]

**7. Handle Outliers**

Detect and address extreme or anomalous values:

* **Use statistical methods** (e.g., Z-score, IQR).
* Replace or remove outliers based on context.

**Example: Using IQR**

Q1 = df['column\_name'].quantile(0.25)

Q3 = df['column\_name'].quantile(0.75)

IQR = Q3 - Q1

lower\_bound = Q1 - 1.5 \* IQR

upper\_bound = Q3 + 1.5 \* IQR

df = df[(df['column\_name'] >= lower\_bound) & (df['column\_name'] <= upper\_bound)]

**8. Convert Data Types**

Ensure data types are appropriate for columns:

* Convert numerical data stored as strings.
* Convert categorical variables to proper types.

**Example:**

df['column\_name'] = df['column\_name'].astype('float') # Convert to float

df['category\_column'] = df['category\_column'].astype('category') # Convert to category

**9. Encode Categorical Variables**

Transform categorical data into numerical formats if needed:

* **One-Hot Encoding**:
* df = pd.get\_dummies(df, columns=['category\_column'], drop\_first=True)
* **Label Encoding**:
* from sklearn.preprocessing import LabelEncoder
* le = LabelEncoder()
* df['category\_column'] = le.fit\_transform(df['category\_column'])

**10. Validate the Filtered Dataset**

After filtering:

1. Check the structure and summary statistics:
2. print(df.info())
3. print(df.describe())
4. Visualize distributions to confirm improvements:
5. import seaborn as sns
6. sns.histplot(df['column\_name'], kde=True)
7. plt.show()

**Checklist for Starting Data Filtering**

| **Step** | **Task** |
| --- | --- |
| Load Dataset | Import and inspect data. |
| Understand Data | Know columns, types, ranges. |
| Handle Missing Values | Impute or remove. |
| Remove Duplicates | Identify and drop. |
| Filter by Ranges | Apply logical constraints. |
| Detect Outliers | Use IQR or Z-score. |
| Normalize/Encode Data | Standardize or encode. |