In [ ]:

SIG720 Task 1P

Solve the following set of problems using Python and submit the code file with extension .ipynb.

Part A: Data Preprocessing (Microclimate Data)

1. Load the "Microclimate sensors data" dataset and print the feature name with numbers of missing entries.
2. Fill in the missing entries. For filling any feature, you can use either the mean or median value of the feature values from observed entries. Explain the reason behind your choice and print replacement value of each feature.
3. Split the "LatLong" column into separate Latitude and Longitude columns using appropriate encoding.
4. Apply Min-Max scaling on the continuous features.
5. Plot the distribution of scaled features before and after scaling. Comment on any changes observed.

Part B: Clustering and PCA

6. Load the "Obesity" dataset and remove the class label "NObeyesdad".
7. Encode the categorical features using appropriate techniques.
8. Use the Silhouette Coefficient to determine the optimal number of clusters (k).
9. Apply KMeans and KMeans++ clustering algorithms using the optimal k. Compare and report the clustering performance.
10. Load the "gene expression" dataset and apply PCA to reduce the data to 3 principal components. Report the variance explained by these components.
11. Apply KMeans on the original features of the gene dataset and the first three components returned by PCA. Compare the results using the given labels.

Part A: Data Preprocessing (Microclimate Data)

1. Load the "Microclimate sensors data" dataset and print the feature name with numbers of missing entries.

2. Fill in the missing entries. For filling any feature, you can use either the mean or median value of the feature values from observed entries. Explain the reason behind your choice and print replacement value of each feature.

3. Split the “LatLong” column into separate Latitude and Longitude columns using appropriate encoding.

4. Apply Min-Max scaling on the continuous features.

5. Plot the distribution of scaled features before and after scaling. Comment on any changes observed.

solution

Importing libraries for mathematical and visualization and of dataset

In [2]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

PART A 1. Load the "**Microclimate sensors data**" dataset and print the feature name with numbers of missing entries.

In [3]:
```
# Read the CSV file containing microclimate sensor data from the specified path
# and store it as a pandas DataFrame named 'data1
data1 = pd.read_csv(r"E:\2. MDS DEAKIN UNIVERSITY\3.SIG 720- Machine learning\TASK\melbourne data set\microcontroller sensor D
```

In [4]:
```
# Display the dimensions of the DataFrame: (number of rows, number of columns)
data1.shape
```

Out[4]:
```
(386468, 16)
```

In [6]:
```
# Display the first 5 rows of the DataFrame to get a quick overview of the data.
data1.head()
```

Out[6]:

	Device_id	Time	SensorLocation	LatLong	MinimumWindDirection	AverageWindDirection	MaximumWindDirectic
0	ICTMicroclimate-08	2025-02-09T11:54:37+11:00	Swanston St - Tram Stop 13 adjacent Federation...	-37.8184515, 144.9678474	0.0	153.0	358
1	ICTMicroclimate-11	2025-02-09T12:02:11+11:00	1 Treasury Place	-37.812888, 144.9750857	0.0	144.0	356
2	ICTMicroclimate-05	2025-02-09T12:03:24+11:00	Enterprize Park - Pole ID: COM1667	-37.8204083, 144.9591192	0.0	45.0	133
3	ICTMicroclimate-01	2025-02-09T12:02:43+11:00	Birrarung Marr Park - Pole 1131	-37.8185931, 144.9716404	NaN	150.0	Na
4	ICTMicroclimate-09	2025-02-09T12:17:37+11:00	SkyFarm (Jeff's Shed). Rooftop - Melbourne Con...	-37.8223306, 144.9521696	0.0	241.0	359

In [7]:
```
# Generate descriptive statistics (count, mean, std, min, max, quartiles) for each column,
# then transpose the output to display columns as rows for better readability.
data1.describe().T
```

Out[7]:

	count	mean	std	min	25%	50%	75%	max
MinimumWindDirection	347670.0	20.451080	57.486759	0.0	0.0	0.0	0.000000	359.000000
AverageWindDirection	385967.0	166.441820	124.911093	0.0	44.0	159.0	300.000000	359.000000
MaximumWindDirection	347512.0	305.753093	89.471621	0.0	314.0	353.0	358.000000	360.000000
MinimumWindSpeed	347512.0	4.856960	38.860659	0.0	0.0	0.0	0.100000	359.000000
AverageWindSpeed	385967.0	1.058148	0.977531	0.0	0.4	0.8	1.500000	11.100000
GustWindSpeed	347512.0	3.445130	2.602599	0.0	1.6	2.8	4.800000	52.500000
AirTemperature	385967.0	16.442966	6.082852	-0.8	12.2	16.0	19.800000	40.800000
RelativeHumidity	385967.0	66.371760	18.417743	4.0	55.0	68.1	79.599998	99.800003
AtmosphericPressure	385967.0	1002.085677	109.144872	20.9	1008.9	1014.7	1020.200012	1042.900000
PM25	368138.0	20.271311	119.052080	0.0	1.0	3.0	7.000000	1030.700000
PM10	368138.0	8.105126	12.539965	0.0	3.0	5.0	9.000000	308.000000
Noise	368138.0	66.404766	13.067686	0.0	59.0	68.3	72.400000	131.100000

In [8]:
```
# Display a concise summary of the DataFrame, including column names, data types,
# non-null counts, and memory usage.
data1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 386468 entries, 0 to 386467
Data columns (total 16 columns):
 #   Column                Non-Null Count   Dtype  
---  ------                --------------   -----  
 0   Device_id             386468 non-null  object 
 1   Time                  386468 non-null  object 
 2   SensorLocation        380325 non-null  object 
 3   LatLong               374985 non-null  object 
 4   MinimumWindDirection  347670 non-null  float64
 5   AverageWindDirection  385967 non-null  float64
 6   MaximumWindDirection  347512 non-null  float64
 7   MinimumWindSpeed      347512 non-null  float64
 8   AverageWindSpeed      385967 non-null  float64
 9   GustWindSpeed         347512 non-null  float64
 10  AirTemperature        385967 non-null  float64
 11  RelativeHumidity      385967 non-null  float64
 12  AtmosphericPressure   385967 non-null  float64
 13  PM25                  368138 non-null  float64
 14  PM10                  368138 non-null  float64
 15  Noise                 368138 non-null  float64
dtypes: float64(12), object(4)
memory usage: 47.2+ MB
```

In [9]:
```python
# Filter the DataFrame to include only rows where the 'Device_id' is 'ICTMicroclimate-01',
# and store this filtered DataFrame in 'df_device2'.

df_device2 = data1.loc[data1['Device_id'] == 'ICTMicroclimate-01']
# Display the filtered DataFrame for the specified device.
df_device2
```

Out[9]:

	Device_id	Time	SensorLocation	LatLong	MinimumWindDirection	AverageWindDirection	MaximumWindD
3	ICTMicroclimate-01	2025-02-09T12:02:43+11:00	Birrarung Marr Park - Pole 1131	-37.8185931, 144.9716404	NaN	150.0	
8	ICTMicroclimate-01	2025-02-09T12:32:44+11:00	Birrarung Marr Park - Pole 1131	-37.8185931, 144.9716404	NaN	143.0	
20	ICTMicroclimate-01	2025-02-09T13:17:46+11:00	Birrarung Marr Park - Pole 1131	-37.8185931, 144.9716404	NaN	142.0	
37	ICTMicroclimate-01	2025-02-09T14:32:48+11:00	Birrarung Marr Park - Pole 1131	-37.8185931, 144.9716404	NaN	149.0	
70	ICTMicroclimate-01	2024-11-03T12:02:38+11:00	Birrarung Marr Park - Pole 1131	-37.8185931, 144.9716404	NaN	10.0	
...	...	...	...	...	...	...	
386442	ICTMicroclimate-01	2025-07-04T18:32:38+10:00	Birrarung Marr Park - Pole 1131	-37.8185931, 144.9716404	NaN	118.0	
386449	ICTMicroclimate-01	2025-07-04T21:32:44+10:00	Birrarung Marr Park - Pole 1131	-37.8185931, 144.9716404	NaN	0.0	
386450	ICTMicroclimate-01	2025-07-04T21:47:45+10:00	Birrarung Marr Park - Pole 1131	-37.8185931, 144.9716404	NaN	45.0	
386458	ICTMicroclimate-01	2025-07-04T22:17:46+10:00	Birrarung Marr Park - Pole 1131	-37.8185931, 144.9716404	NaN	112.0	
386459	ICTMicroclimate-01	2025-07-04T22:39:26+10:00	Birrarung Marr Park - Pole 1131	-37.8185931, 144.9716404	NaN	95.0	

38483 rows × 16 columns

In [10]:

```
# Select all columns in the DataFrame that have the data type 'object' (typically text/categorical columns)
# and store them in 'object_columns'.
object_columns=data1.select_dtypes(include='object')
```

```
# Display the first 5 rows of the selected object-type columns.
object_columns.head()
```

Out[10]:

	Device_id	Time	SensorLocation	LatLong
0	ICTMicroclimate-08	2025-02-09T11:54:37+11:00	Swanston St - Tram Stop 13 adjacent Federation...	-37.8184515, 144.9678474
1	ICTMicroclimate-11	2025-02-09T12:02:11+11:00	1 Treasury Place	-37.812888, 144.9750857
2	ICTMicroclimate-05	2025-02-09T12:03:24+11:00	Enterprize Park - Pole ID: COM1667	-37.8204083, 144.9591192
3	ICTMicroclimate-01	2025-02-09T12:02:43+11:00	Birrarung Marr Park - Pole 1131	-37.8185931, 144.9716404
4	ICTMicroclimate-09	2025-02-09T12:17:37+11:00	SkyFarm (Jeff's Shed). Rooftop - Melbourne Con...	-37.8223306, 144.9521696

In [12]:
```
# Display all unique values present in the 'LatLong' column of the DataFrame.
data1.LatLong.unique()
```

Out[12]:
```
array(['-37.8184515, 144.9678474', '-37.812888, 144.9750857',
       '-37.8204083, 144.9591192', '-37.8185931, 144.9716404',
       '-37.8223306, 144.9521696', '-37.814604, 144.9702991',
       '-37.8222341, 144.9829409', '-37.8221828, 144.9562225',
       '-37.8194993, 144.9787211', '-37.7956167, 144.9519007',
       '-37.8140348, 144.96728', '-37.8128595, 144.9745395', nan],
      dtype=object)
```

In [13]:
```
# Select all columns in the DataFrame that have a numeric data type (e.g., int, float)
# and store them in 'numerical_columns'.
numerical_columns=data1.select_dtypes(include='number')

# Display the first 5 rows of the selected numerical columns.
numerical_columns.head()
```

Out[13]:

	MinimumWindDirection	AverageWindDirection	MaximumWindDirection	MinimumWindSpeed	AverageWindSpeed	GustWindSpeed	Ai
0	0.0	153.0	358.0	0.0	3.9	7.9	
1	0.0	144.0	356.0	0.0	2.0	7.8	
2	0.0	45.0	133.0	0.0	1.5	2.7	
3	NaN	150.0	NaN	NaN	1.6	NaN	
4	0.0	241.0	359.0	0.0	0.9	4.4	

Answer PART 1

In [15]:
```
print(f" 1) the feature has {len(object_columns.columns)} columns which has object values and those are {object_columns.column
print(f" 2) the feature has {len( numerical_columns.columns)} columns which has contious numerical values and those are \n {nu
```

```
 1) the feature has 4 columns which has object values and those are Index(['Device_id', 'Time', 'SensorLocation', 'LatLong'], d
type='object')
 2) the feature has 12 columns which has contious numerical values and those are
 Index(['MinimumWindDirection', 'AverageWindDirection', 'MaximumWindDirection',
       'MinimumWindSpeed', 'AverageWindSpeed', 'GustWindSpeed',
       'AirTemperature', 'RelativeHumidity', 'AtmosphericPressure', 'PM25',
       'PM10', 'Noise'],
      dtype='object')
```

In [16]:
```
# Calculate the total number of missing (null) values in each column of the DataFrame
# and store the result in 'missing_columns'.
missing_columns=data1.isnull().sum()
```

In [14]:
```
def show_missing_and_present(df):
    # Get the total number of rows in the DataFrame.
    total_rows = len(df)

    # Calculate the number of missing (NaN) values for each column.
    miss = df.isna().sum()

    # Calculate the number of non-missing (non-null) values for each column.
```

```
    pres = df.count()  # count of non-null entries

    # Create and return a new DataFrame showing:
    # - missing count
    # - present count
    # - missing ratio (missing count divided by total rows)
    # - present ratio (present count divided by total rows)
    return pd.DataFrame({
        'missing_count': miss,
        'present_count': pres,
        'missing_ratio': miss / total_rows,
        'present_ratio': pres / total_rows
    })
```

In [15]:
```
# Call the custom function 'show_missing_and_present' to calculate
# the count and ratio of missing and present values for each column,
# and store the result in 'missing_data'.
missing_data=show_missing_and_present(data1)

# Display the DataFrame containing the missing and present data summary.
missing_data
```

Out[15]:

	missing_count	present_count	missing_ratio	present_ratio
Device_id	0	386468	0.000000	1.000000
Time	0	386468	0.000000	1.000000
SensorLocation	6143	380325	0.015895	0.984105
LatLong	11483	374985	0.029713	0.970287
MinimumWindDirection	38798	347670	0.100391	0.899609
AverageWindDirection	501	385967	0.001296	0.998704
MaximumWindDirection	38956	347512	0.100800	0.899200
MinimumWindSpeed	38956	347512	0.100800	0.899200
AverageWindSpeed	501	385967	0.001296	0.998704
GustWindSpeed	38956	347512	0.100800	0.899200
AirTemperature	501	385967	0.001296	0.998704
RelativeHumidity	501	385967	0.001296	0.998704
AtmosphericPressure	501	385967	0.001296	0.998704
PM25	18330	368138	0.047430	0.952570
PM10	18330	368138	0.047430	0.952570
Noise	18330	368138	0.047430	0.952570

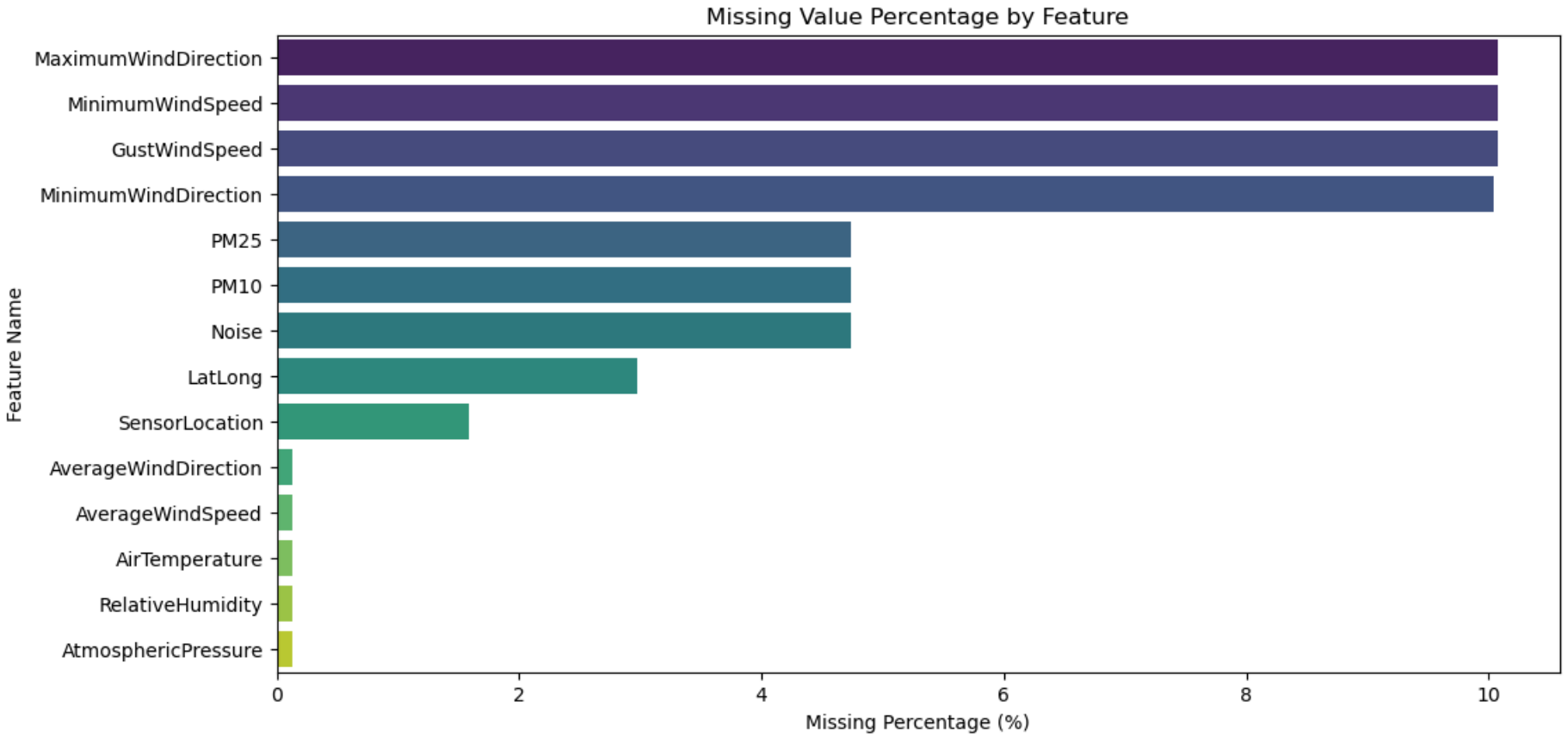
visualizing missing values

In [20]:

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Calculate percentage of missing values for each column
missing_percent = data1.isnull().sum() * 100 / len(data1)
```

```python
#  Convert to DataFrame for easy plotting
missing_df = missing_percent.reset_index()
missing_df.columns = ['Feature', 'MissingPercentage']

#  Filter only features with missing values > 0%
missing_df = missing_df[missing_df['MissingPercentage'] > 0]

# Sort for better visual order
missing_df = missing_df.sort_values(by='MissingPercentage', ascending=False)

#  Plot barplot
plt.figure(figsize=(12, 6))
sns.barplot(x='MissingPercentage', y='Feature', data=missing_df, palette='viridis')
plt.title('Missing Value Percentage by Feature')
plt.xlabel('Missing Percentage (%)')
plt.ylabel('Feature Name')
plt.show()
```

```
C:\Users\Chirag Pc\AppData\Local\Temp\ipykernel_21212\442220658.py:20: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `y` variable to `hue` and se
t `legend=False` for the same effect.

  sns.barplot(x='MissingPercentage', y='Feature', data=missing_df, palette='viridis')
```



Observations

Key Observations

1.Dataset size:

- The dataset has 386,468 rows and 16 columns, so it's large enough for robust analysis and modeling.
- It covers multiple device IDs, timestamps, locations, and various environmental sensor readings.

2. Wind Direction & Wind Speed

- **MinimumWindDirection** has a mean of ~20° but a very high standard deviation (~57°), with a median of 0 — suggesting many records with calm or no wind.
- **AverageWindDirection** has a mean ~166°, a wide spread, and a reasonable median (~159°).
- **MaximumWindDirection** leans high with a mean ~305°, indicating stronger gusts.
- **MinimumWindSpeed** shows a mean ~4.85, but the median is 0, implying calm periods are common.
- **AverageWindSpeed** has a mean ~1.05, indicating generally light wind.
- **GustWindSpeed mean** is ~3.44, maxing at 52.5 — some extreme gusts are recorded.

Possible insight: Many records have calm or low wind — the high maximum values suggest occasional strong gusts.

3. Air Temperature

- Mean temperature is 16.4°C, with a range from -0.8°C to 40.8°C.
- This indicates a temperate climate with occasional cold and hot extremes — normal for Melbourne.

4. Relative Humidity

- Mean RH is about 66%, with a range from 4% to almost 100%.
- Shows a generally humid microclimate, but with wide daily variations.

5. Atmospheric Pressure

- Mean is ~1002 hPa, with surprisingly wide std dev (109) and minimum 20.9 — this minimum likely signals erroneous sensor readings.
- Typical sea-level pressure is ~1013 hPa, so apart from outliers, the average looks fine.

6. PM2.5, PM10, and Noise

- PM2.5 has a mean of 20.2 but an unusually high max (1030.7) — this may be an outlier or data error, as typical urban PM2.5 rarely exceeds 500 $\mu g/m^3$.
- PM10 mean is 8.1 — lower than PM2.5 mean, which is odd because PM10 is usually higher; might indicate measurement/label mismatch.
- Noise has a mean of 66.4 dB, ranging from 0 to 131 dB, which indicates very noisy environments at peak times.

7. First Rows Check

- The sample rows show:
- Multiple devices (e.g., ICTMicroclimate-08, ICTMicroclimate-11, etc.).
- Each record has a timestamp, sensor location, and coordinates.
- Some rows contain NaNs for wind data, matching your missing values summary.
- Realistic air temperature, humidity, noise, and PM readings.

Observations on Missing Data

- 1. Device_id and Time columns have complete data — no missing values at all (0% missing). This means the dataset is well-indexed by device and timestamp, which is ideal for time series analysis.
- 2. SensorLocation and LatLong have small but notable missing portions:
  - SensorLocation: ~1.6% missing — suggests occasional gaps in sensor placement records or logging errors.
  - LatLong: ~3% missing — means some records lack exact geo-coordinates, which could affect spatial analysis.
- 3. Wind Direction & Wind Speed (Minimum, Maximum, Gust) have relatively high missingness (~10%) — indicating sensors might fail under certain weather conditions or may not always record at low speeds.
- 4. Average Wind Direction & Average Wind Speed, Air Temperature, Relative Humidity, and Atmospheric Pressure are almost complete — only ~0.1% missing, which is negligible for most analyses.
- 5. PM2.5, PM10, and Noise all show ~4.7% missing — possibly due to sensor downtimes or calibration intervals for air quality and sound sensors.

PART A 2. Fill in the missing entries. For filling any feature, you can use either the mean or median value of the feature values from observed entries. Explain the reason behind your choice and print replacement value of each feature.

In [21]:
```
#creating copy of original dataset
data2=data1.copy()
```

In [22]:
```
### fist lets see again missing entries
data2.isnull().sum()
```

Out[22]:
```
Device_id                  0
Time                       0
SensorLocation          6143
LatLong                11483
MinimumWindDirection   38798
AverageWindDirection     501
MaximumWindDirection   38956
MinimumWindSpeed       38956
AverageWindSpeed         501
GustWindSpeed          38956
AirTemperature           501
RelativeHumidity         501
AtmosphericPressure      501
PM25                   18330
PM10                   18330
Noise                  18330
dtype: int64
```

filling null entries for sensor location

In [23]:
```
#sensor location
print(data2['SensorLocation'].value_counts(dropna=False))
```

```
SensorLocation
1 Treasury Place                                                        47650
Birrarung Marr Park - Pole 1131                                         38376
Tram Stop 7C - Melbourne Tennis Centre Precinct - Rod Laver Arena       38311
CH1 rooftop                                                             38243
101 Collins St L11 Rooftop                                              38151
Tram Stop 7B - Melbourne Tennis Centre Precinct - Rod Laver Arena       38122
Swanston St - Tram Stop 13 adjacent Federation Sq & Flinders St Station 37620
SkyFarm (Jeff's Shed). Rooftop - Melbourne Conference & Exhibition Centre (MCEC)    36611
Batman Park                                                             26126
Enterprize Park - Pole ID: COM1667                                      23334
Royal Park Asset ID: COM2707                                            17781
NaN                                                                      6143
Name: count, dtype: int64
```

observation:

1 .High-frequency locations

- 1 Treasury Place (47,650 records) and Birrarung Marr Park (38,376 records) are the most common locations—likely indicating frequent device assignments or high data collection frequency at these sites.
- Locations like Tram Stop 7C (38,311), CH1 rooftop (38,243), and 101 Collins St L11 Rooftop (38,151) also appear heavily.

2. Mid-tier locations

- Places like Tram Stop 7B (38,122), Swanston St tram stop (37,620), and SkyFarm (MCEC) (36,611) fall into the mid-range, suggesting balanced data coverage across these sites.

3. Lower-frequency sites

- Batman Park (26,126), Enterprize Park (23,334), and Royal Park (17,781) appear less often. This could indicate fewer sensors, shorter deployment periods, or occasional data collection there.

4. Missing location data

- NaN appears in 6,143 records—about 1.6% of the total dataset. This is a non-trivial amount and should be addressed, either via imputation, removal, or further investigation into why location data is missing.

In [24]:
```
# Replace `df` with your actual DataFrame — here you are using `data2`
data = data2

# Step 1: Group by 'Device_id' and collect unique 'SensorLocation' values for each device
# This creates a new DataFrame with:
# - Device_id
# - A list of unique locations for that device
location_history = (
    data.groupby('Device_id')['SensorLocation']
        .unique()
        .reset_index(name='location_list')
)

# Step 2: Add a new column 'has_moved'
# For each device, check if the list of locations has more than 1 unique entry
```

```python
# If True → device has moved; else → it stayed stationary
location_history['has_moved'] = location_history['location_list'].apply(lambda locs: len(locs) > 1)

# Step 3: Loop through each row and print the movement status and all locations
for _, row in location_history.iterrows():
    dev = row['Device_id']                  # The device ID
    locs = row['location_list']             # The list of unique locations
    moved = row['has_moved']                # True if device moved, False otherwise
    status = "MOVED" if moved else "STATIONARY"  # Label the status
    print(f"Device '{dev}' → Status: {status}, Locations: {list(locs)}")
```

```
Device 'ICTMicroclimate-01' → Status: MOVED, Locations: ['Birrarung Marr Park - Pole 1131', nan]
Device 'ICTMicroclimate-02' → Status: MOVED, Locations: ['101 Collins St L11 Rooftop', nan]
Device 'ICTMicroclimate-03' → Status: MOVED, Locations: ['CH1 rooftop', nan]
Device 'ICTMicroclimate-04' → Status: MOVED, Locations: ['Batman Park', nan]
Device 'ICTMicroclimate-05' → Status: STATIONARY, Locations: ['Enterprize Park - Pole ID: COM1667']
Device 'ICTMicroclimate-06' → Status: MOVED, Locations: ['Tram Stop 7B - Melbourne Tennis Centre Precinct - Rod Laver Arena', n
an]
Device 'ICTMicroclimate-07' → Status: MOVED, Locations: ['Tram Stop 7C - Melbourne Tennis Centre Precinct - Rod Laver Arena', n
an]
Device 'ICTMicroclimate-08' → Status: MOVED, Locations: ['Swanston St - Tram Stop 13 adjacent Federation Sq & Flinders St Stati
on', nan]
Device 'ICTMicroclimate-09' → Status: MOVED, Locations: ["SkyFarm (Jeff's Shed). Rooftop - Melbourne Conference & Exhibition Ce
ntre (MCEC)", nan]
Device 'ICTMicroclimate-10' → Status: MOVED, Locations: ['1 Treasury Place', nan]
Device 'ICTMicroclimate-11' → Status: MOVED, Locations: ['1 Treasury Place', nan]
Device 'aws5-0999' → Status: MOVED, Locations: ['Royal Park Asset ID: COM2707', nan]
```

Observation: As we can see, each device has moved to either NaN or "Stationary" status.

Conclusion: From this observation, we can infer that the missing entries (NaN) are likely due to no new recorded movement.

Solution: We can replace each NaN with the respective device's last known location.

Example Implementation:

- Device ICTMicroclimate-01 has a known location: 'Birrarung Marr Park - Pole 1131' → Replace its NaN with 'Birrarung Marr Park - Pole 1131'
- Apply the same logic for all other devices, using their respective last recorded locations.

This ensures data consistency while preserving accurate device positioning.

1.filling missing entries for 'SensorLocation'.

In [25]:
```
#   Step 1: For each Device_id, find the most frequent (mode) SensorLocation
# - Group the data by 'Device_id'
# - Drop any NaN values from the 'SensorLocation' column within each group
# - Calculate the mode (most common value) for each device
# - Take the first mode in case there are multiple modes (just one is enough)
# - Convert the result to a dictionary: {Device_id: mode_location}
mode_loc = (
    data2.groupby('Device_id')['SensorLocation']
      .apply(lambda x: x.dropna().mode().iloc[0])
      .to_dict()
)
```

In [26]:
```
#  Step 2: Fill missing 'SensorLocation' values using the mode for each 'Device_id'
# - For each row, check if 'SensorLocation' is NaN
# - If it is NaN, replace it with the mode location from the 'mode_loc' dictionary for that device
# - If it is not NaN, keep the original value
# - 'axis=1' means the lambda function works row-wise
data2['SensorLocation'] = data2.apply(
    lambda r: mode_loc[r['Device_id']] if pd.isna(r['SensorLocation']) else r['SensorLocation'],
    axis=1
)
```

lets crossckeck have we succesfully remove nan for sensor_loaction

In [27]:
```
# Replace `data` with your actual DataFrame
data = data2

# Step 1: Group by Device_id and collect unique SensorLocations
location_history = (
    data.groupby('Device_id')['SensorLocation']
        .unique()
        .reset_index(name='location_list')
)

# Step 2: For each device, check if it has moved (more than one location)
```

```
location_history['has_moved'] = location_history['location_list'].apply(lambda locs: len(locs) > 1)

# Step 3: Print results
for _, row in location_history.iterrows():
    dev = row['Device_id']
    locs = row['location_list']
    moved = row['has_moved']
    status = "MOVED" if moved else "STATIONARY"
    print(f"Device '{dev}' → Status: {status}, Locations: {list(locs)}")
```

```
Device 'ICTMicroclimate-01' → Status: STATIONARY, Locations: ['Birrarung Marr Park - Pole 1131']
Device 'ICTMicroclimate-02' → Status: STATIONARY, Locations: ['101 Collins St L11 Rooftop']
Device 'ICTMicroclimate-03' → Status: STATIONARY, Locations: ['CH1 rooftop']
Device 'ICTMicroclimate-04' → Status: STATIONARY, Locations: ['Batman Park']
Device 'ICTMicroclimate-05' → Status: STATIONARY, Locations: ['Enterprize Park - Pole ID: COM1667']
Device 'ICTMicroclimate-06' → Status: STATIONARY, Locations: ['Tram Stop 7B - Melbourne Tennis Centre Precinct - Rod Laver Aren
a']
Device 'ICTMicroclimate-07' → Status: STATIONARY, Locations: ['Tram Stop 7C - Melbourne Tennis Centre Precinct - Rod Laver Aren
a']
Device 'ICTMicroclimate-08' → Status: STATIONARY, Locations: ['Swanston St - Tram Stop 13 adjacent Federation Sq & Flinders St
Station']
Device 'ICTMicroclimate-09' → Status: STATIONARY, Locations: ["SkyFarm (Jeff's Shed). Rooftop - Melbourne Conference & Exhibiti
on Centre (MCEC)"]
Device 'ICTMicroclimate-10' → Status: STATIONARY, Locations: ['1 Treasury Place']
Device 'ICTMicroclimate-11' → Status: STATIONARY, Locations: ['1 Treasury Place']
Device 'aws5-0999' → Status: STATIONARY, Locations: ['Royal Park Asset ID: COM2707']
```

Success fully fill entries for sensor location

2.filling missing enteries of 'latlong' column

In [28]:
```
#  Display all unique values in the 'LatLong' column of the DataFrame.
# This helps check how many unique geolocation pairs exist in the data.
data2['LatLong'].unique()
```

Out[28]:
```
array(['-37.8184515, 144.9678474', '-37.812888, 144.9750857',
       '-37.8204083, 144.9591192', '-37.8185931, 144.9716404',
       '-37.8223306, 144.9521696', '-37.814604, 144.9702991',
       '-37.8222341, 144.9829409', '-37.8221828, 144.9562225',
       '-37.8194993, 144.9787211', '-37.7956167, 144.9519007',
       '-37.8140348, 144.96728', '-37.8128595, 144.9745395', nan],
      dtype=object)
```

In [29]:
```
# Count the frequency of each unique value in the 'LatLong' column,
# including NaN values (dropna=False keeps NaNs in the count).
# This shows how many times each unique coordinate pair appears in the dataset.
data2['LatLong'].value_counts(dropna=False)
```

Out[29]:
```
LatLong
-37.8185931, 144.9716404    38376
-37.8222341, 144.9829409    38311
-37.8140348, 144.96728      38243
-37.814604, 144.9702991     38151
-37.8194993, 144.9787211    38122
-37.8184515, 144.9678474    37620
-37.8223306, 144.9521696    36611
-37.812888, 144.9750857     26496
-37.8221828, 144.9562225    26126
-37.8204083, 144.9591192    23334
-37.7956167, 144.9519007    17781
-37.8128595, 144.9745395    15814
NaN                         11483
Name: count, dtype: int64
```

In [31]:
```
# Replace with your actual DataFrame name (here it's data2)
data = data2

#  Group the DataFrame by 'Device_id' and collect all unique 'LatLong' values for each device
# - This returns one row per device
# - The 'unique_latlongs' column contains a list of unique coordinates for that device
unique_latlongs = (
    data.groupby('Device_id')['LatLong']
        .unique()
        .reset_index(name='unique_latlongs')
)
```

```
# Add a new column 'count_unique' that counts how many unique LatLongs each device has
# - Use .apply(len) to count the number of unique entries in each list
unique_latlongs['count_unique'] = unique_latlongs['unique_latlongs'].apply(len)

# Print the final DataFrame: shows each device, its list of unique LatLongs, and how many there are
print(unique_latlongs)
```

```
            Device_id                 unique_latlongs  count_unique
0   ICTMicroclimate-01  [-37.8185931, 144.9716404, nan]             2
1   ICTMicroclimate-02   [-37.814604, 144.9702991, nan]             2
2   ICTMicroclimate-03    [-37.8140348, 144.96728, nan]             2
3   ICTMicroclimate-04  [-37.8221828, 144.9562225, nan]             2
4   ICTMicroclimate-05       [-37.8204083, 144.9591192]             1
5   ICTMicroclimate-06  [-37.8194993, 144.9787211, nan]             2
6   ICTMicroclimate-07  [-37.8222341, 144.9829409, nan]             2
7   ICTMicroclimate-08  [-37.8184515, 144.9678474, nan]             2
8   ICTMicroclimate-09  [-37.8223306, 144.9521696, nan]             2
9   ICTMicroclimate-10  [-37.8128595, 144.9745395, nan]             2
10  ICTMicroclimate-11   [-37.812888, 144.9750857, nan]             2
11           aws5-0999  [-37.7956167, 144.9519007, nan]             2
```

Observations

- All devices except ICTMicroclimate-05 have exactly 2 unique LatLong values — one real coordinate and one NaN.
- ICTMicroclimate-05 appears stable with only 1 unique coordinate and no missing entries.
- This pattern suggests the missing LatLongs (NaN) correspond to times when location wasn't recorded—not actual movement to distinct places.

In [32]:
```
# Replace with your actual DataFrame name (here it's data2)
df = data2

#  Step 1: For each Device_id, find the most frequent (mode) LatLong that is not null
# - Group by 'Device_id'
# - Drop NaNs within each group
# - Get the mode (most common LatLong)
# - Take the first mode if there are ties
# - Convert the result to a dictionary: {Device_id: mode_LatLong}
mode_latlong = (
    df.groupby('Device_id')['LatLong']
```

```
      .apply(lambda x: x.dropna().mode().iloc[0])
      .to_dict()
)

#  Step 2: Fill missing LatLong values row by row using the mode for that device
# - If the row's LatLong is NaN, replace it with the mode_latlong for its Device_id
# - If not NaN, keep the original LatLong
# - axis=1 → apply works row-wise
df['LatLong'] = df.apply(
    lambda r: mode_latlong[r['Device_id']]
              if pd.isna(r['LatLong'])
              else r['LatLong'],
    axis=1
)
```

let verify for latlong misssing / nan values

In [37]:
```
# Count the frequency of each unique LatLong value in the DataFrame.
# This helps check how many times each coordinate pair appears after filling missing values.
# Useful to verify if some LatLongs dominate or if the filling step worked as expected.
df['LatLong'].value_counts()
```

Out[37]:
```
LatLong
-37.8185931, 144.9716404    38483
-37.8222341, 144.9829409    38418
-37.8140348, 144.96728      38350
-37.814604, 144.9702991     38259
-37.8194993, 144.9787211    38229
-37.8184515, 144.9678474    37723
-37.8223306, 144.9521696    36718
-37.812888, 144.9750857     27786
-37.8221828, 144.9562225    26135
-37.8128595, 144.9745395    25204
-37.8204083, 144.9591192    23334
-37.7956167, 144.9519007    17829
Name: count, dtype: int64
```

Successfully filled nan of 'LatLong'

step 3.filling missing value for all other floating values.

In [40]:
```
# Define the list of sensor measurement columns to keep.
# These are the main numeric environmental readings from the dataset.
sensor_cols = [
    'MinimumWindDirection', 'AverageWindDirection', 'MaximumWindDirection',
    'MinimumWindSpeed', 'AverageWindSpeed', 'GustWindSpeed',
    'AirTemperature', 'RelativeHumidity', 'AtmosphericPressure',
    'PM25', 'PM10', 'Noise'
]

# Combine the ID, timestamp, and location columns with the sensor columns
# to form the final list of columns to keep in your cleaned DataFrame.
cols_to_keep = ['Device_id', 'Time', 'SensorLocation', 'LatLong'] + sensor_cols
```

In [41]:
```
# Include identifiers + sensor columns

df_filtered = data2[cols_to_keep]

# Group by 'Device_id' and build a dict of sub-DataFrames
data3 = df_filtered.groupby('Device_id')
```

In [34]:
```
data3.head()
```

Out[34]:

	Device_id	Time	SensorLocation	LatLong	MinimumWindDirection	AverageWindDirection	MaximumWindDirec
0	ICTMicroclimate-08	2025-02-09T11:54:37+11:00	Swanston St - Tram Stop 13 adjacent Federation...	-37.8184515, 144.9678474	0.0	153.0	3
1	ICTMicroclimate-11	2025-02-09T12:02:11+11:00	1 Treasury Place	-37.812888, 144.9750857	0.0	144.0	3
2	ICTMicroclimate-05	2025-02-09T12:03:24+11:00	Enterprize Park - Pole ID: COM1667	-37.8204083, 144.9591192	0.0	45.0	1
3	ICTMicroclimate-01	2025-02-09T12:02:43+11:00	Birrarung Marr Park - Pole 1131	-37.8185931, 144.9716404	NaN	150.0	
4	ICTMicroclimate-09	2025-02-09T12:17:37+11:00	SkyFarm (Jeff's Shed). Rooftop - Melbourne Con...	-37.8223306, 144.9521696	0.0	241.0	3
5	ICTMicroclimate-05	2025-02-09T12:18:26+11:00	Enterprize Park - Pole ID: COM1667	-37.8204083, 144.9591192	0.0	357.0	
6	ICTMicroclimate-02	2025-02-09T12:26:51+11:00	101 Collins St L11 Rooftop	-37.814604, 144.9702991	0.0	357.0	3
7	ICTMicroclimate-07	2025-02-09T12:35:39+11:00	Tram Stop 7C - Melbourne Tennis Centre Precinc...	-37.8222341, 144.9829409	0.0	91.0	3
8	ICTMicroclimate-01	2025-02-09T12:32:44+11:00	Birrarung Marr Park - Pole 1131	-37.8185931, 144.9716404	NaN	143.0	
9	ICTMicroclimate-04	2025-02-09T12:38:22+11:00	Batman Park	-37.8221828, 144.9562225	0.0	10.0	3
10	ICTMicroclimate-09	2025-02-09T12:47:40+11:00	SkyFarm (Jeff's Shed). Rooftop	-37.8223306, 144.9521696	0.0	232.0	3

	Device_id	Time	SensorLocation	LatLong	MinimumWindDirection	AverageWindDirection	MaximumWindDirec
			- Melbourne Con...				
11	ICTMicroclimate-11	2025-02-09T12:47:14+11:00	1 Treasury Place	-37.812888, 144.9750857	0.0	192.0	3
12	ICTMicroclimate-07	2025-02-09T12:50:40+11:00	Tram Stop 7C - Melbourne Tennis Centre Precinc...	-37.8222341, 144.9829409	0.0	121.0	3
13	ICTMicroclimate-06	2025-02-09T12:52:35+11:00	Tram Stop 7B - Melbourne Tennis Centre Precinc...	-37.8194993, 144.9787211	0.0	166.0	3
14	ICTMicroclimate-08	2025-02-09T12:54:40+11:00	Swanston St - Tram Stop 13 adjacent Federation...	-37.8184515, 144.9678474	0.0	147.0	3
15	ICTMicroclimate-08	2025-02-09T12:09:37+11:00	Swanston St - Tram Stop 13 adjacent Federation...	-37.8184515, 144.9678474	0.0	168.0	3
16	aws5-0999	2025-02-09T12:09:47+11:00	Royal Park Asset ID: COM2707	-37.7956167, 144.9519007	0.0	2.0	3
17	ICTMicroclimate-11	2025-02-09T12:32:12+11:00	1 Treasury Place	-37.812888, 144.9750857	0.0	120.0	3
18	ICTMicroclimate-11	2025-02-09T13:02:15+11:00	1 Treasury Place	-37.812888, 144.9750857	0.0	170.0	3
19	ICTMicroclimate-05	2025-02-09T13:03:31+11:00	Enterprize Park - Pole ID: COM1667	-37.8204083, 144.9591192	0.0	359.0	3
20	ICTMicroclimate-01	2025-02-09T13:17:46+11:00	Birrarung Marr Park - Pole 1131	-37.8185931, 144.9716404	NaN	142.0	

	Device_id	Time	SensorLocation	LatLong	MinimumWindDirection	AverageWindDirection	MaximumWindDirec
21	ICTMicroclimate-04	2025-02-09T13:23:27+11:00	Batman Park	-37.8221828, 144.9562225	0.0	358.0	3
22	ICTMicroclimate-03	2025-02-09T13:20:29+11:00	CH1 rooftop	-37.8140348, 144.96728	0.0	165.0	2
23	ICTMicroclimate-08	2025-02-09T13:24:46+11:00	Swanston St - Tram Stop 13 adjacent Federation...	-37.8184515, 144.9678474	0.0	105.0	3
24	ICTMicroclimate-09	2025-02-09T13:32:45+11:00	SkyFarm (Jeff's Shed). Rooftop - Melbourne Con...	-37.8223306, 144.9521696	0.0	260.0	3
25	ICTMicroclimate-07	2025-02-09T13:35:47+11:00	Tram Stop 7C - Melbourne Tennis Centre Precinc...	-37.8222341, 144.9829409	0.0	148.0	3
26	ICTMicroclimate-05	2025-02-09T13:33:36+11:00	Enterprize Park - Pole ID: COM1667	-37.8204083, 144.9591192	0.0	349.0	3
27	ICTMicroclimate-03	2025-02-09T13:35:32+11:00	CH1 rooftop	-37.8140348, 144.96728	177.0	213.0	2
28	ICTMicroclimate-09	2025-02-09T13:47:47+11:00	SkyFarm (Jeff's Shed). Rooftop - Melbourne Con...	-37.8223306, 144.9521696	0.0	208.0	3
29	ICTMicroclimate-08	2025-02-09T13:54:47+11:00	Swanston St - Tram Stop 13 adjacent Federation...	-37.8184515, 144.9678474	0.0	120.0	3
30	ICTMicroclimate-02	2025-02-09T13:57:03+11:00	101 Collins St L11 Rooftop	-37.814604, 144.9702991	0.0	284.0	3

	Device_id	Time	SensorLocation	LatLong	MinimumWindDirection	AverageWindDirection	MaximumWindDirec
31	ICTMicroclimate-11	2025-02-09T14:02:19+11:00	1 Treasury Place	-37.812888, 144.9750857	0.0	170.0	3
33	ICTMicroclimate-06	2025-02-09T14:22:47+11:00	Tram Stop 7B - Melbourne Tennis Centre Precinc...	-37.8194993, 144.9787211	0.0	102.0	3
34	ICTMicroclimate-05	2025-02-09T14:18:41+11:00	Enterprize Park - Pole ID: COM1667	-37.8204083, 144.9591192	0.0	97.0	
35	ICTMicroclimate-02	2025-02-09T14:27:07+11:00	101 Collins St L11 Rooftop	-37.814604, 144.9702991	0.0	347.0	3
36	ICTMicroclimate-09	2025-02-09T14:32:51+11:00	SkyFarm (Jeff's Shed). Rooftop - Melbourne Con...	-37.8223306, 144.9521696	0.0	279.0	3
37	ICTMicroclimate-01	2025-02-09T14:32:48+11:00	Birrarung Marr Park - Pole 1131	-37.8185931, 144.9716404	NaN	149.0	
40	ICTMicroclimate-06	2025-02-09T14:07:44+11:00	Tram Stop 7B - Melbourne Tennis Centre Precinc...	-37.8194993, 144.9787211	0.0	131.0	3
41	ICTMicroclimate-07	2025-02-09T14:05:51+11:00	Tram Stop 7C - Melbourne Tennis Centre Precinc...	-37.8222341, 144.9829409	0.0	89.0	3
43	ICTMicroclimate-02	2025-02-09T14:42:09+11:00	101 Collins St L11 Rooftop	-37.814604, 144.9702991	0.0	245.0	3
44	ICTMicroclimate-06	2025-02-09T14:52:52+11:00	Tram Stop 7B - Melbourne Tennis Centre Precinc...	-37.8194993, 144.9787211	0.0	144.0	3

	Device_id	Time	SensorLocation	LatLong	MinimumWindDirection	AverageWindDirection	MaximumWindDirec
45	ICTMicroclimate-04	2025-02-09T14:53:40+11:00	Batman Park	-37.8221828, 144.9562225	0.0	143.0	3
48	ICTMicroclimate-10	2024-11-03T13:17:17+11:00	1 Treasury Place	-37.8128595, 144.9745395	87.0	186.0	2
50	ICTMicroclimate-06	2024-11-03T13:06:04+11:00	Tram Stop 7B - Melbourne Tennis Centre Precinc...	-37.8194993, 144.9787211	0.0	148.0	3
51	ICTMicroclimate-02	2024-11-03T12:59:01+11:00	101 Collins St L11 Rooftop	-37.814604, 144.9702991	297.0	359.0	3
53	ICTMicroclimate-03	2024-11-03T12:52:32+11:00	CH1 rooftop	-37.8140348, 144.96728	153.0	168.0	1
54	ICTMicroclimate-07	2024-11-03T13:07:48+11:00	Tram Stop 7C - Melbourne Tennis Centre Precinc...	-37.8222341, 144.9829409	0.0	184.0	3
59	ICTMicroclimate-04	2024-11-03T13:40:36+11:00	Batman Park	-37.8221828, 144.9562225	0.0	18.0	3
67	ICTMicroclimate-04	2024-11-03T12:25:26+11:00	Batman Park	-37.8221828, 144.9562225	0.0	215.0	3
70	ICTMicroclimate-01	2024-11-03T12:02:38+11:00	Birrarung Marr Park - Pole 1131	-37.8185931, 144.9716404	NaN	10.0	
73	ICTMicroclimate-03	2024-11-03T13:52:39+11:00	CH1 rooftop	-37.8140348, 144.96728	161.0	182.0	2
80	ICTMicroclimate-03	2024-11-03T14:22:42+11:00	CH1 rooftop	-37.8140348, 144.96728	171.0	179.0	2
84	ICTMicroclimate-10	2024-11-03T14:32:23+11:00	1 Treasury Place	-37.8128595, 144.9745395	126.0	166.0	2

	Device_id	Time	SensorLocation	LatLong	MinimumWindDirection	AverageWindDirection	MaximumWindDirec
98	ICTMicroclimate-10	2024-10-20T14:25:00+11:00	1 Treasury Place	-37.8128595, 144.9745395	82.0	138.0	2
207	aws5-0999	2025-02-09T11:24:56+11:00	Royal Park Asset ID: COM2707	-37.7956167, 144.9519007	0.0	328.0	3
215	ICTMicroclimate-10	2024-11-03T12:02:12+11:00	1 Treasury Place	-37.8128595, 144.9745395	155.0	173.0	2
220	aws5-0999	2024-11-03T12:14:39+11:00	Royal Park Asset ID: COM2707	-37.7956167, 144.9519007	0.0	16.0	3
223	ICTMicroclimate-10	2024-11-03T12:32:15+11:00	1 Treasury Place	-37.8128595, 144.9745395	76.0	144.0	3
239	aws5-0999	2024-11-03T13:44:29+11:00	Royal Park Asset ID: COM2707	-37.7956167, 144.9519007	341.0	334.0	3
274	aws5-0999	2024-10-20T04:08:01+11:00	Royal Park Asset ID: COM2707	-37.7956167, 144.9519007	0.0	9.0	3

In [42]:
```
#  Transform the DataFrame from wide format to long format using melt.
# This is useful for tidy data — one measurement per row.

df_long = data2.melt(
    id_vars=['Device_id'],   #  Columns to keep fixed (here, only 'Device_id')
    value_vars=sensor_cols,  #  Columns to unpivot (the sensor measurement columns)
    var_name='Feature',      #  Name of the new column that will hold the sensor feature names
    value_name='Value'       #  Name of the new column that will hold the actual measurement values
)
```

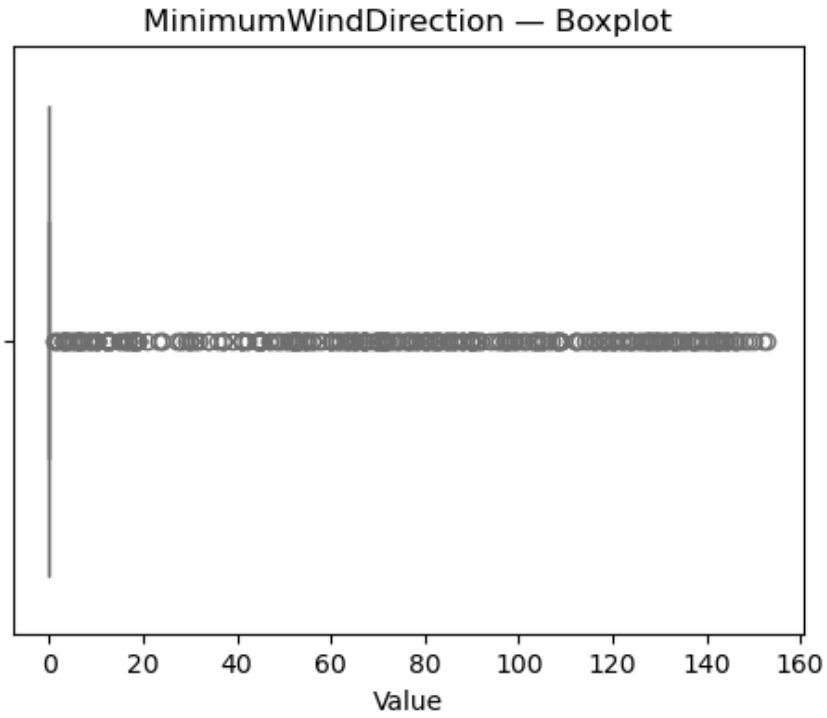
In [43]:
```
import seaborn as sns
import matplotlib.pyplot as plt

def plot_device_distributions(device_id):
    # Filter the long-format DataFrame for the selected device ID
    data = df_long[df_long['Device_id'] == device_id]
```

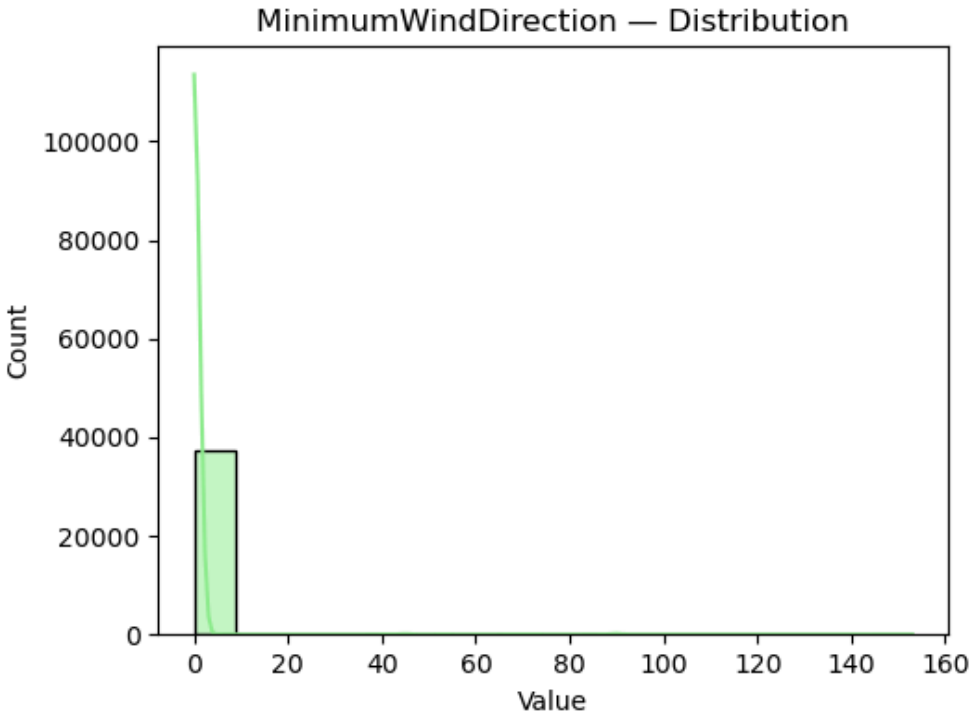
```
    #  Get the list of unique sensor features for this device
    features = data['Feature'].unique()
    n = len(features)  # Total number of features to plot

    #  Create subplots: 2 plots per feature (boxplot + histogram)
    fig, axes = plt.subplots(
        nrows=n,      # One row per feature
        ncols=2,      # Two columns: boxplot + distribution plot
        figsize=(10, 4 * n)  # Adjust figure size dynamically
    )

    # Loop through each feature and create its plots
    for i, feature in enumerate(features):
        # Filter data for this feature and drop missing values
        subset = data[data['Feature'] == feature]['Value'].dropna()

        # Select axes for boxplot and distribution plot
        ax_box = axes[i, 0]
        ax_dist = axes[i, 1]

        # Draw boxplot on the left
        sns.boxplot(x=subset, ax=ax_box, color='skyblue')
        ax_box.set_title(f"{feature} — Boxplot")

        # Draw histogram with KDE on the right
        sns.histplot(subset, kde=True, ax=ax_dist, color='lightgreen')
        ax_dist.set_title(f"{feature} — Distribution")

    #  Add a main title for the entire figure
    plt.suptitle(f"Device: {device_id}", y=1.02, fontsize=16)

    # Adjust layout so plots don't overlap
    plt.tight_layout()

    # Display the plots
    plt.show()
```

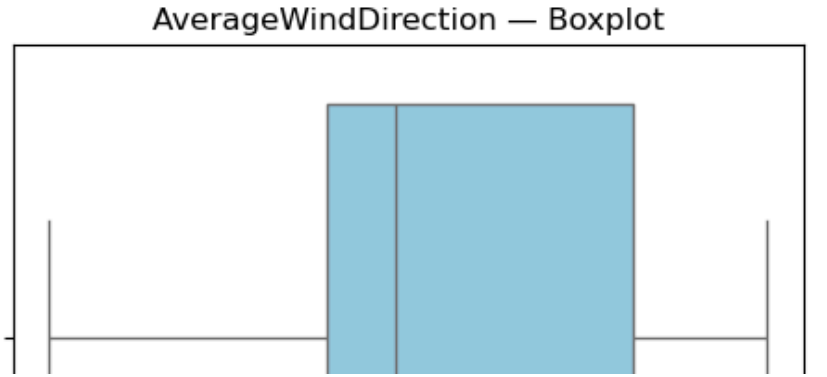
In [44]:
```
# Loop through each unique Device_id in your DataFrame
for dev in df['Device_id'].unique():
```

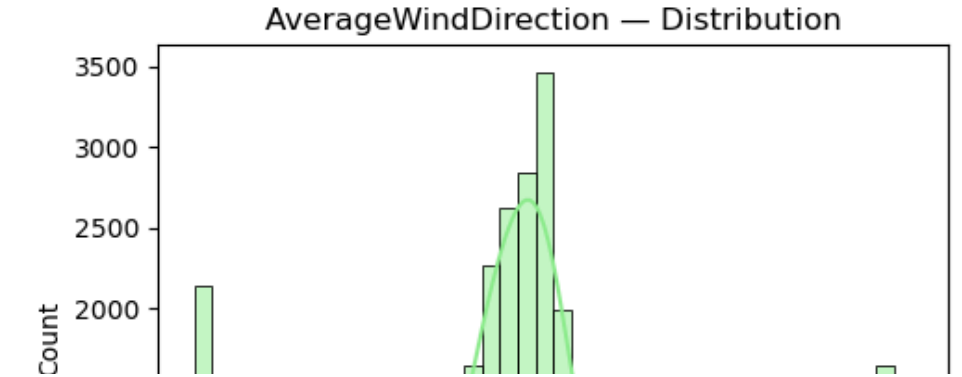
```
    # For each device, call the custom plotting function
    # This generates a boxplot + histogram for every feature of that device
    plot_device_distributions(dev)
```

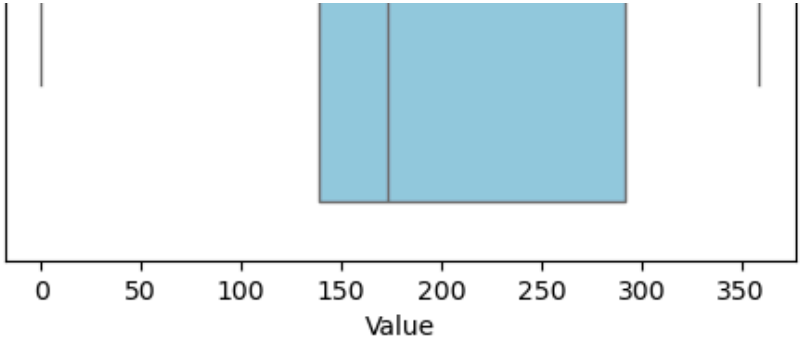
Device: ICTMicroclimate-08

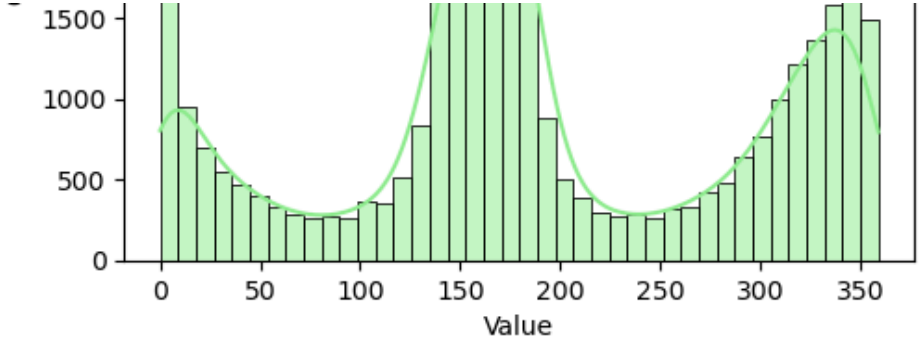




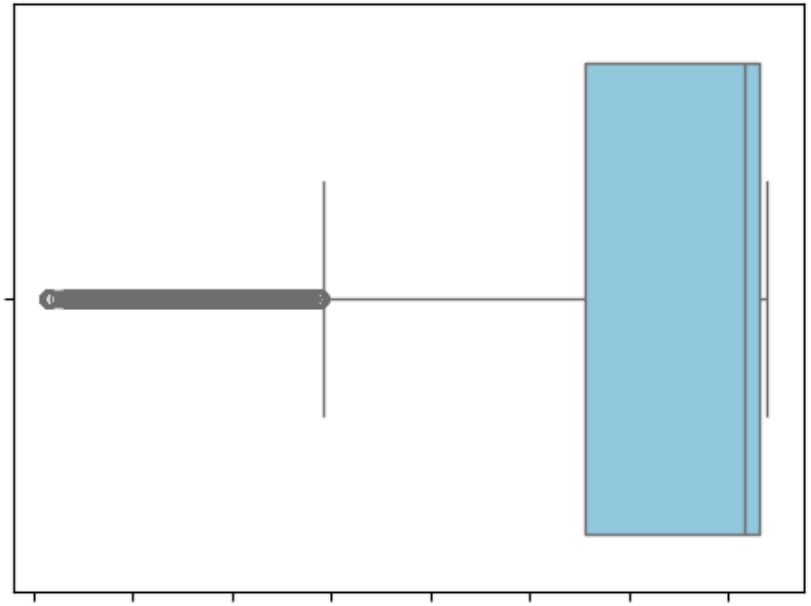




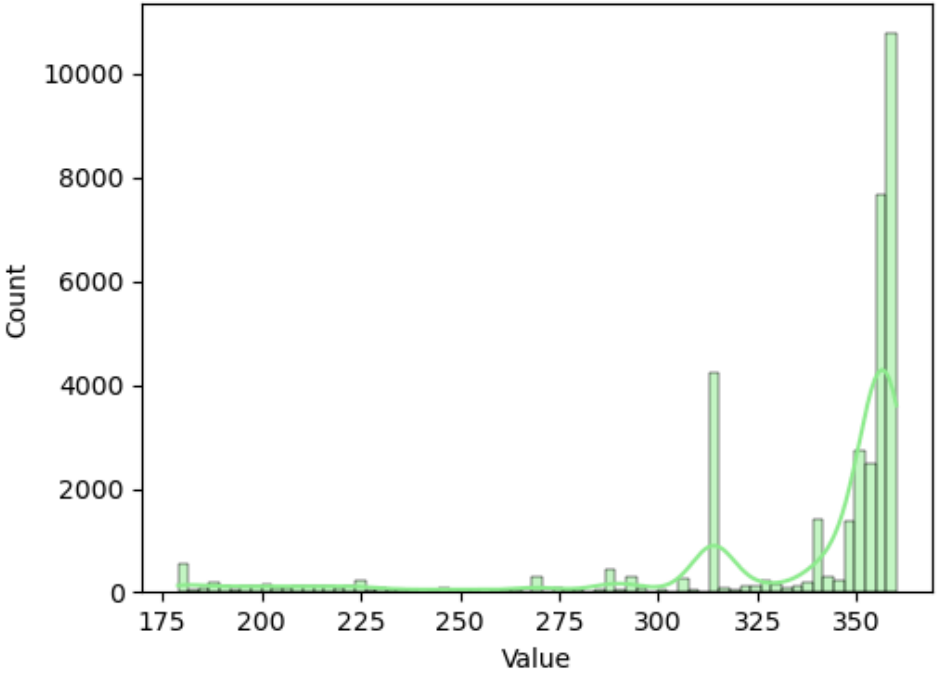




MaximumWindDirection — Boxplot

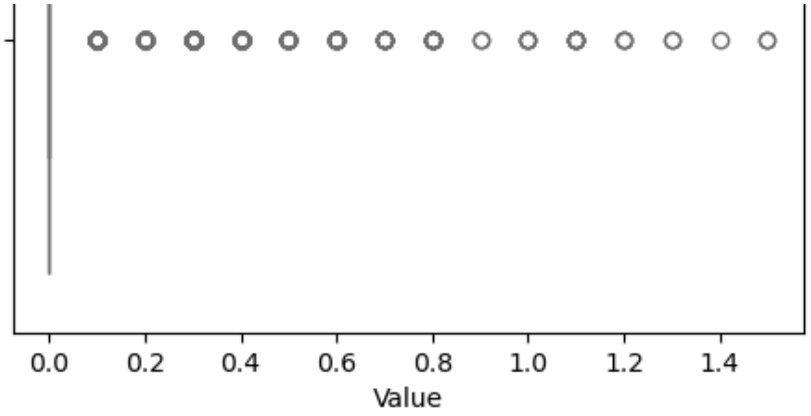


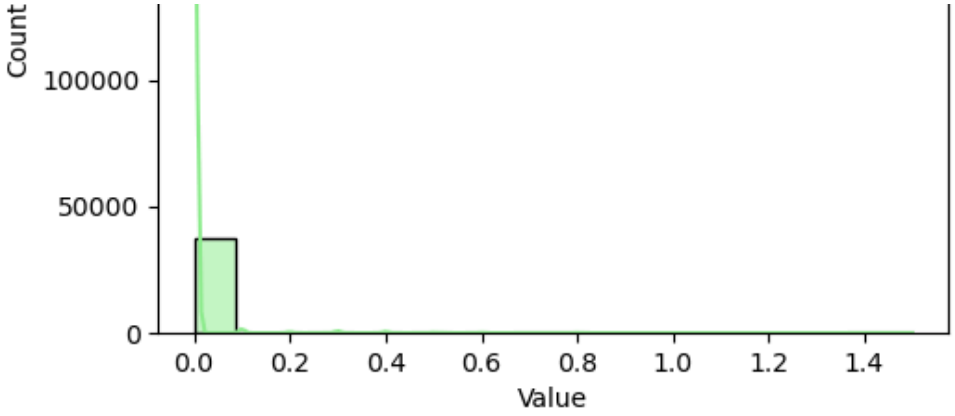
MaximumWindDirection — Distribution



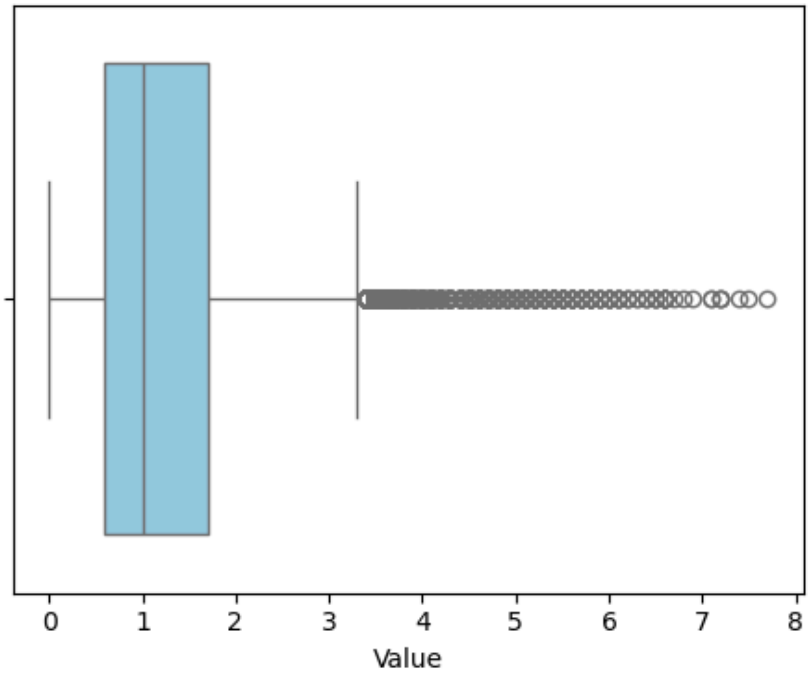
MinimumWindSpeed — Boxplot

MinimumWindSpeed — Distribution

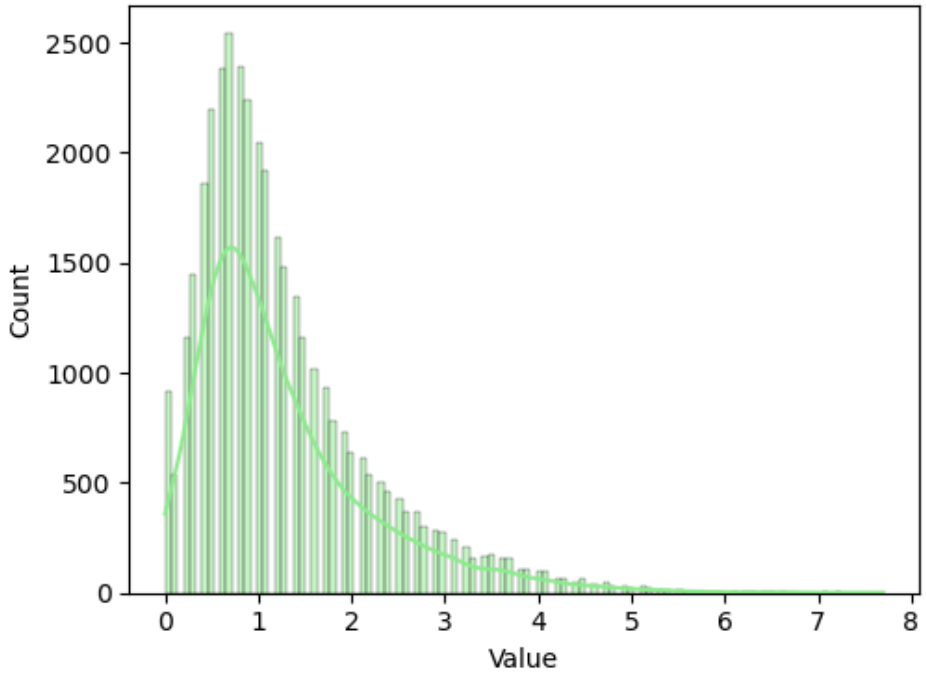




AverageWindSpeed — Boxplot

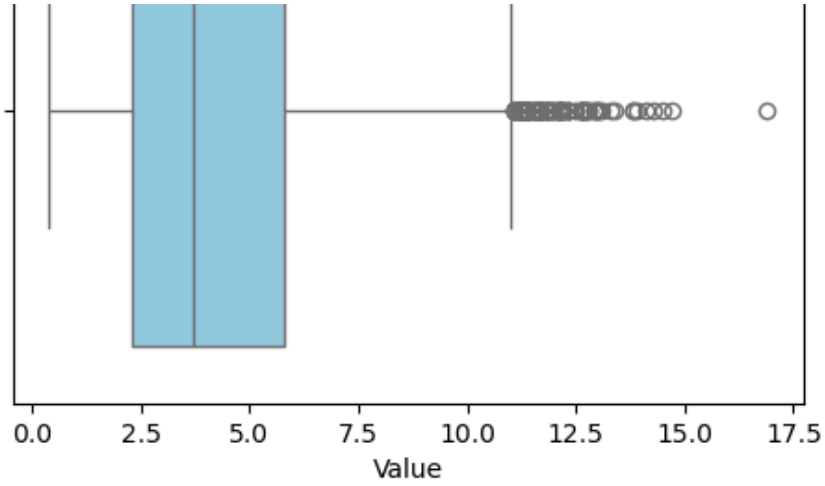


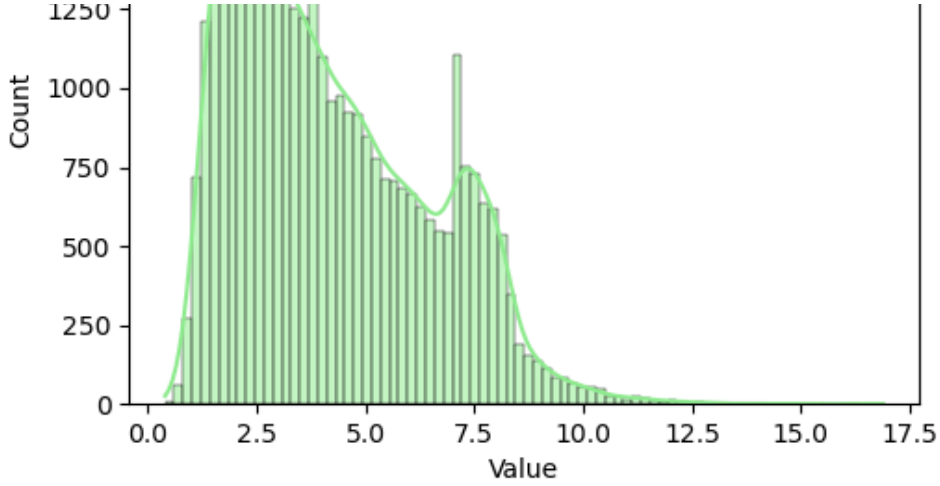
AverageWindSpeed — Distribution



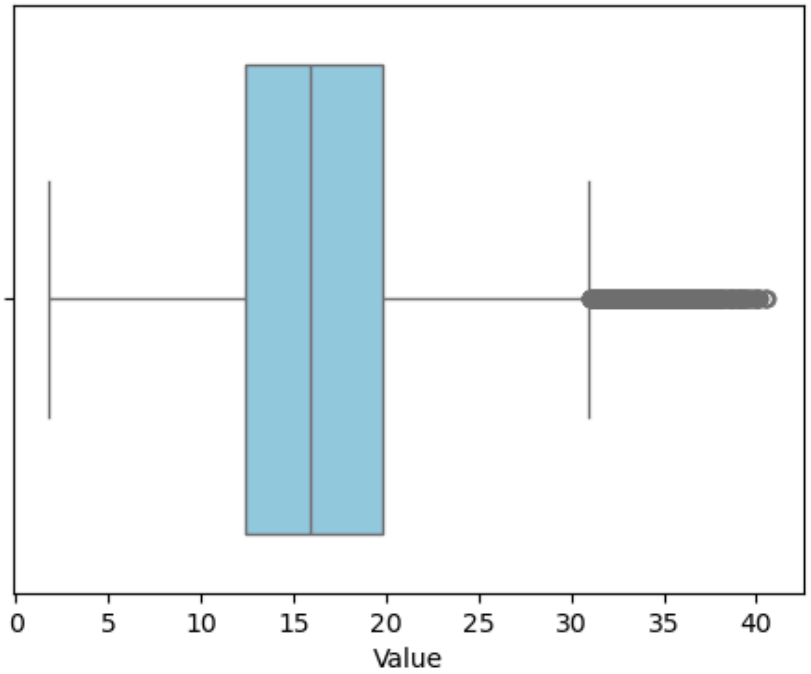
GustWindSpeed — Boxplot

GustWindSpeed — Distribution

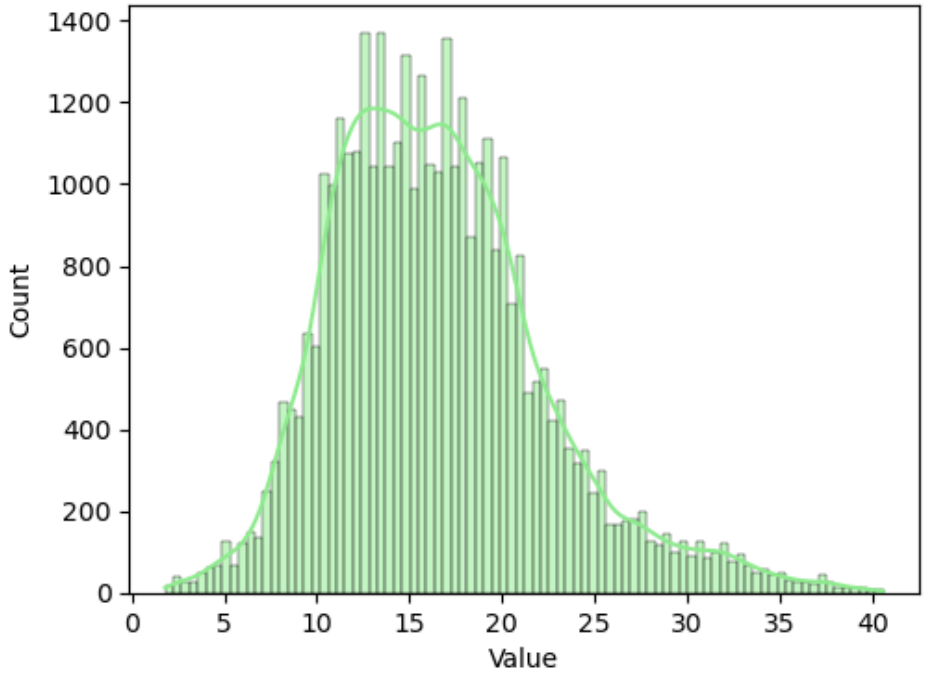




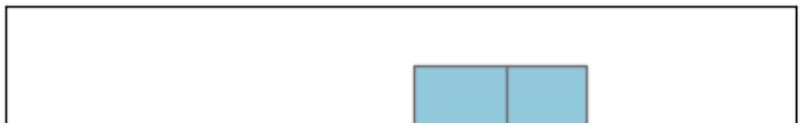
AirTemperature — Boxplot



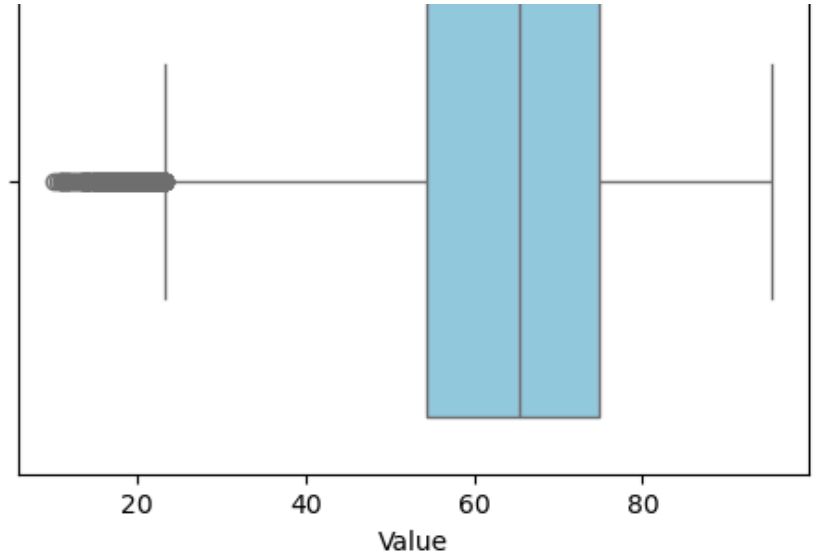
AirTemperature — Distribution

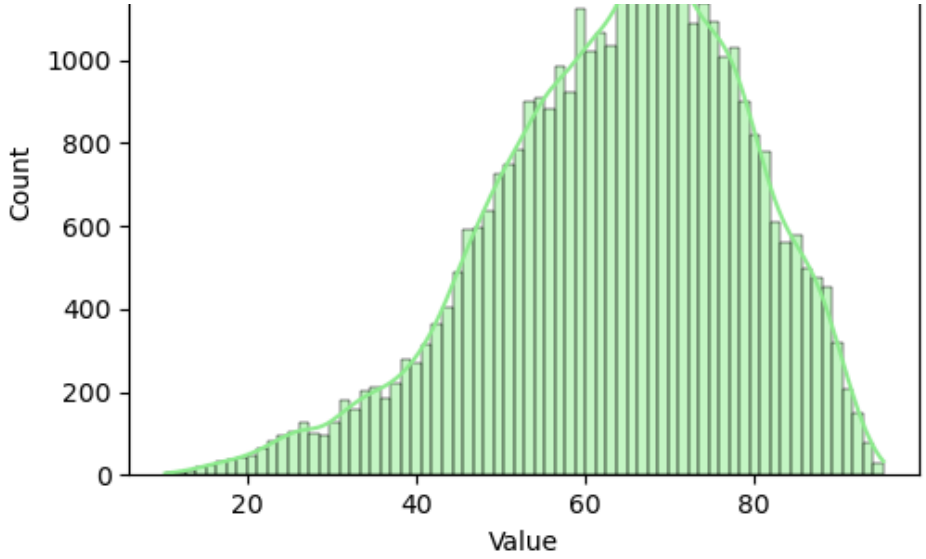


RelativeHumidity — Boxplot

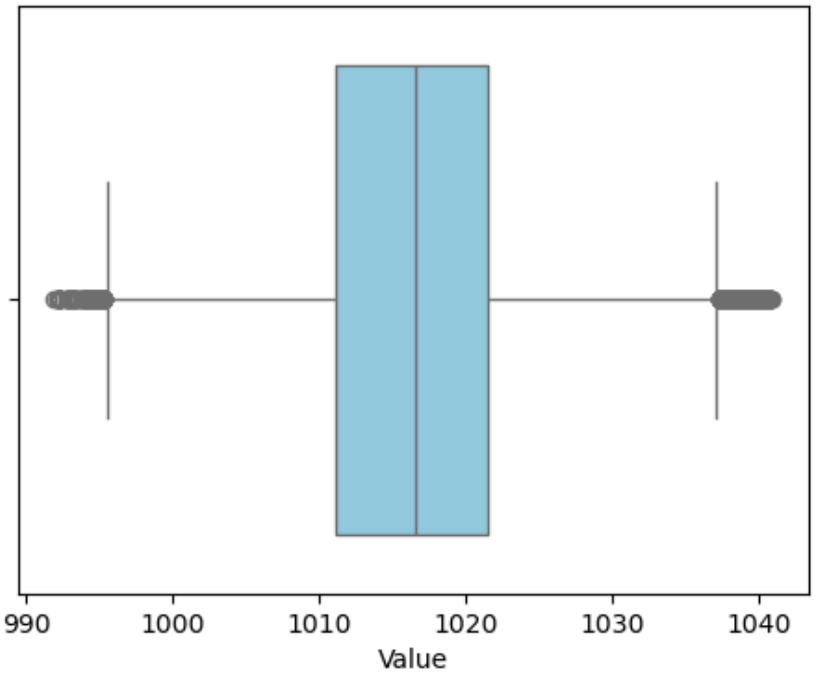


RelativeHumidity — Distribution

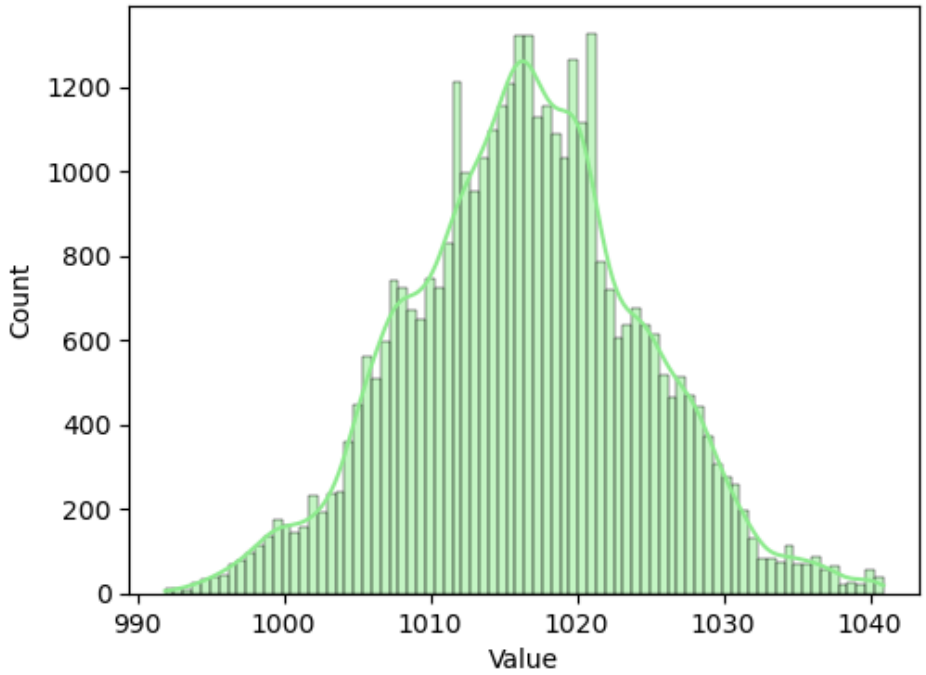




AtmosphericPressure — Boxplot

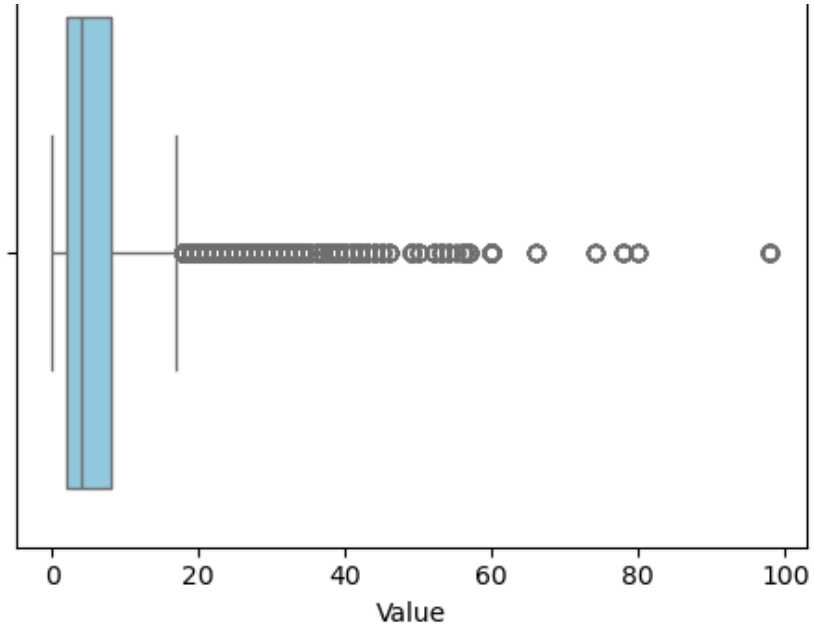


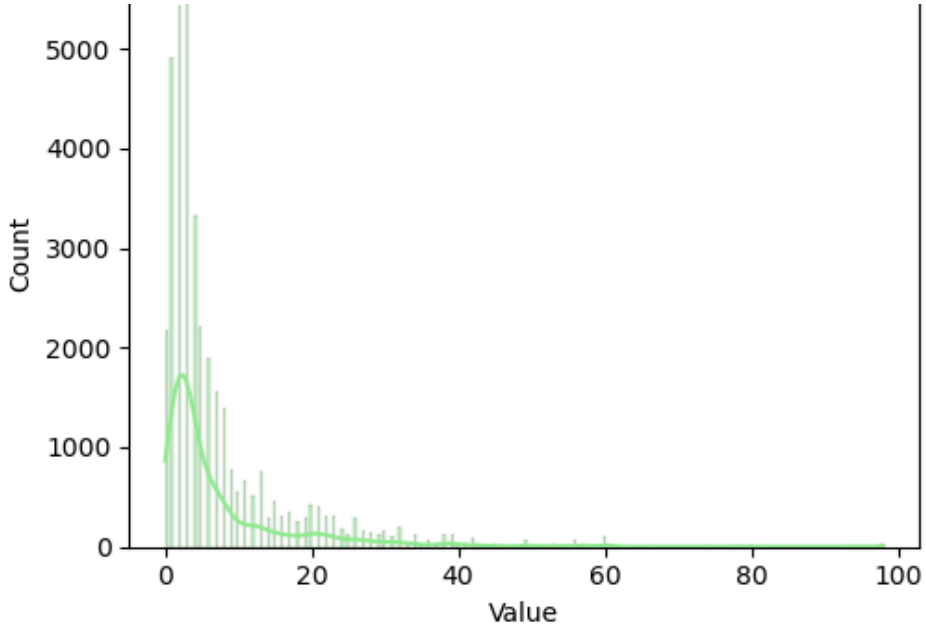
AtmosphericPressure — Distribution



PM25 — Boxplot

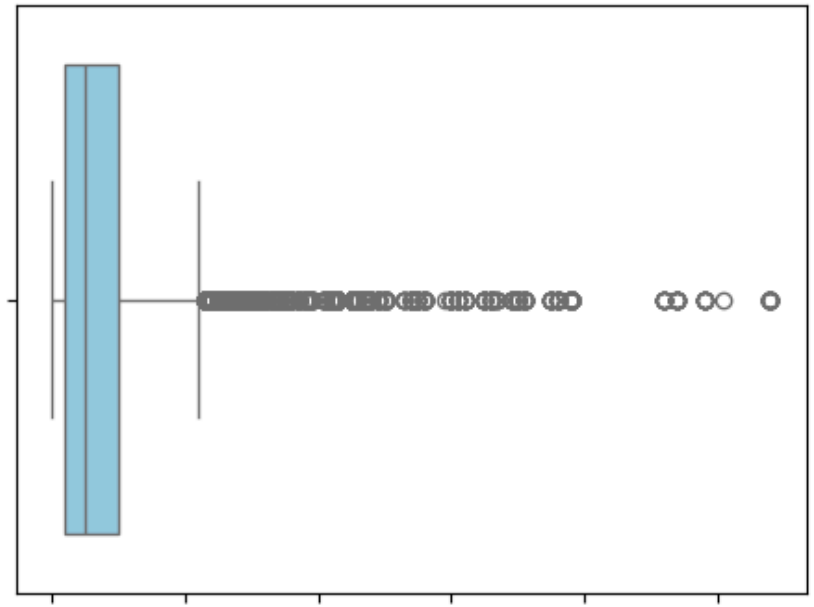
PM25 — Distribution

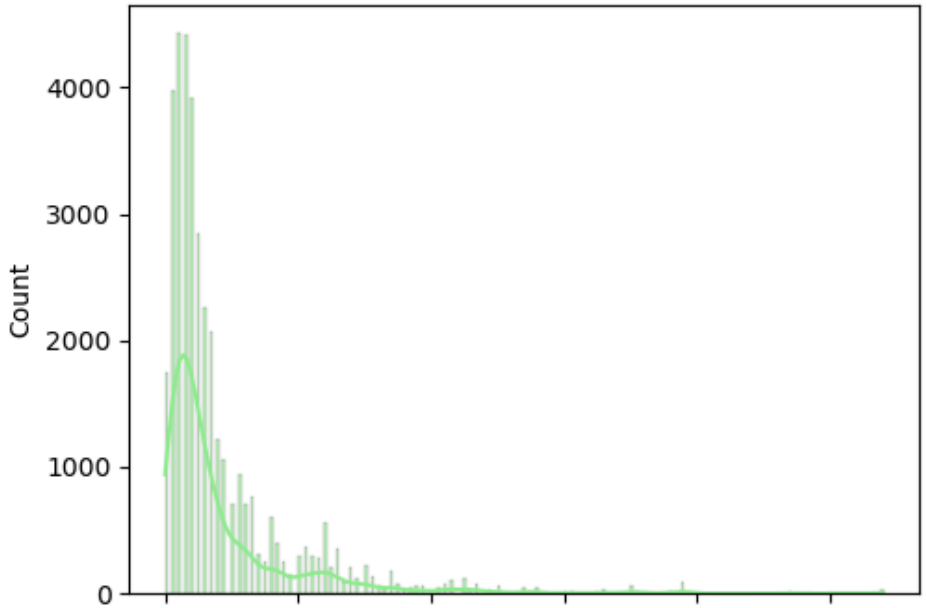




PM10 — Boxplot

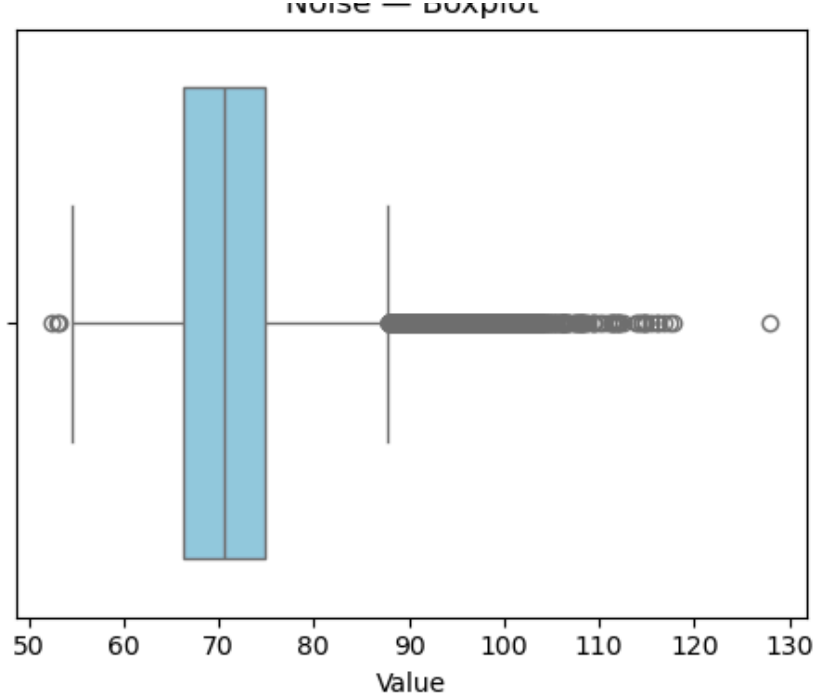
PM10 — Distribution

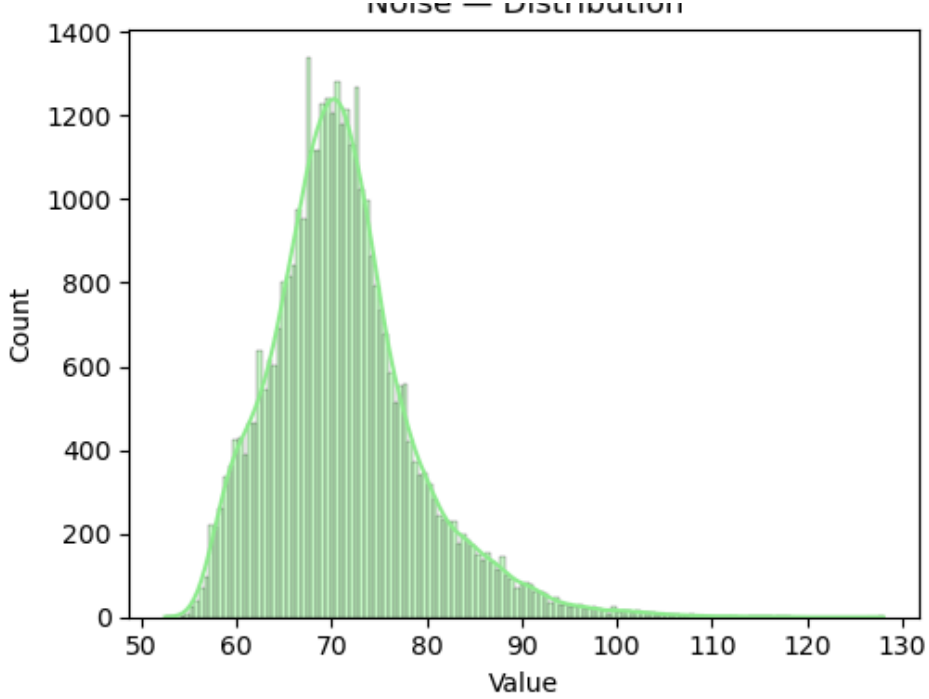




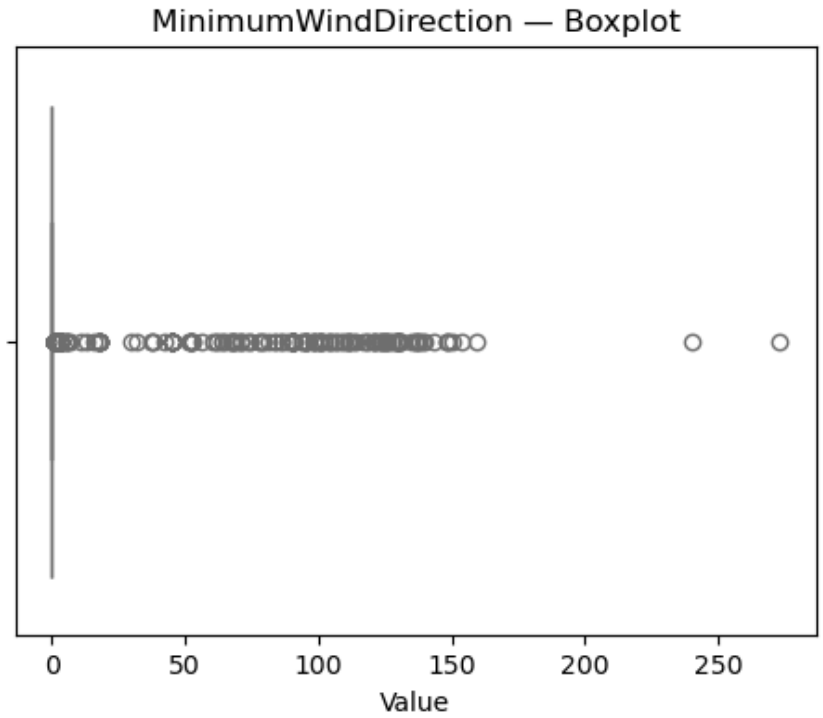
Noise — Boxplot

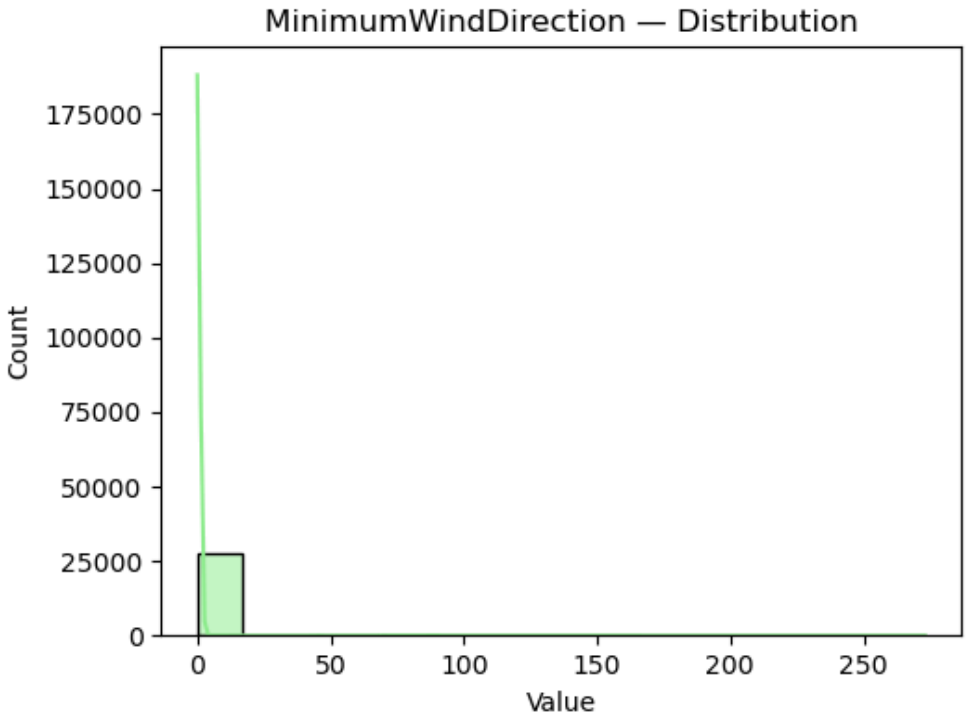
Noise — Distribution

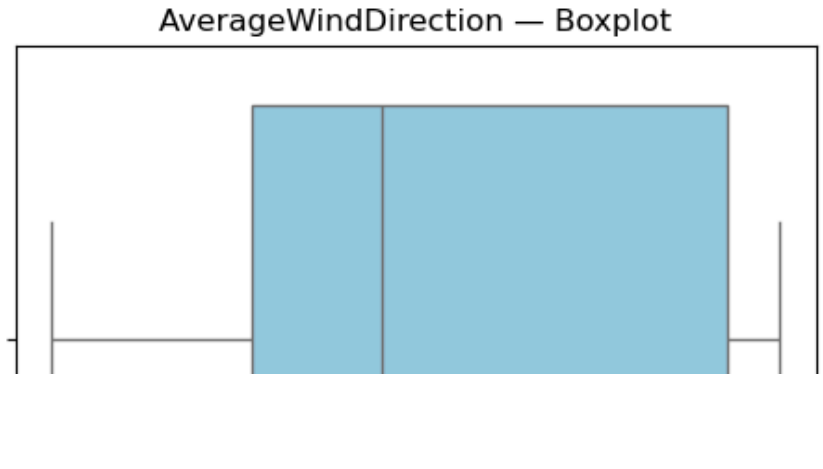

Noise — Boxplot
50
60
70
80
90
100
110
120
130
Value

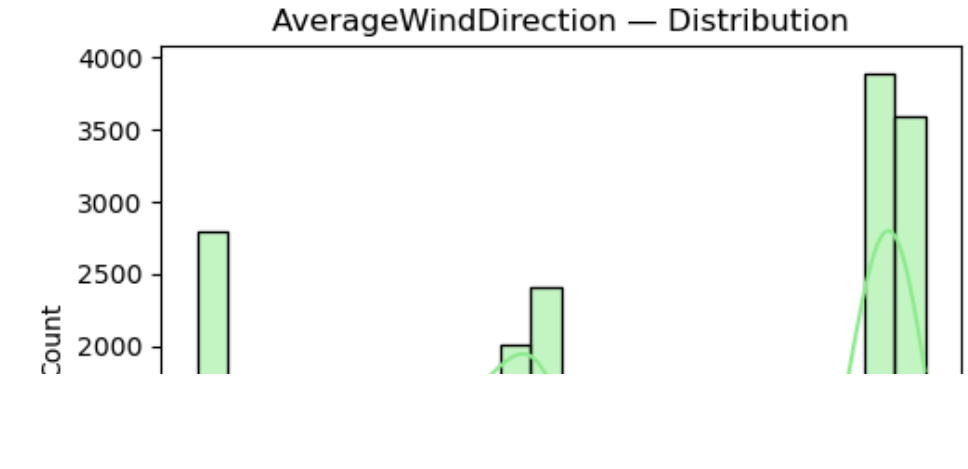

Noise — Distribution
1400
1200
1000
800
600
400
200
0
Count
50
60
70
80
90
100
110
120
130
Value

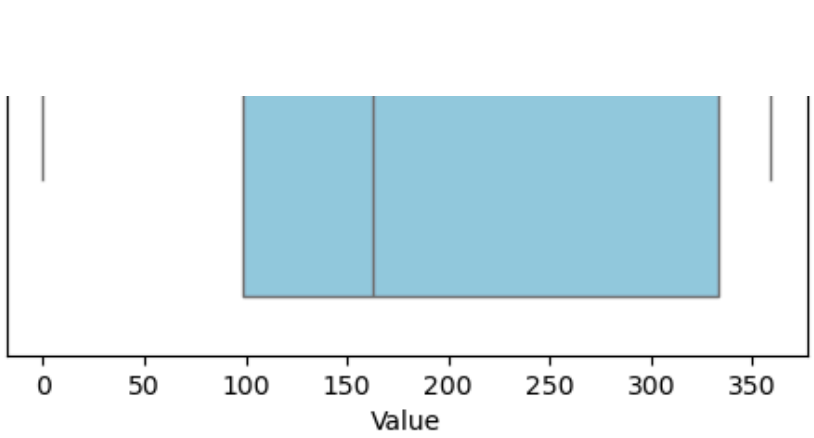
Device: ICTMicroclimate-11

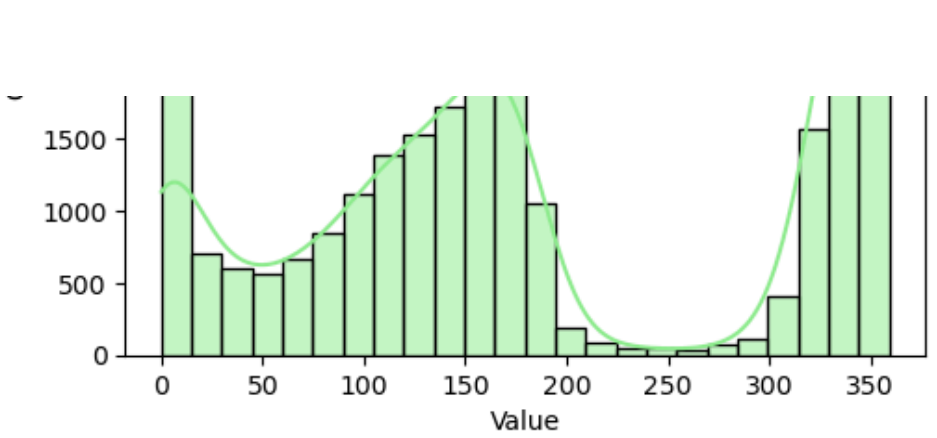




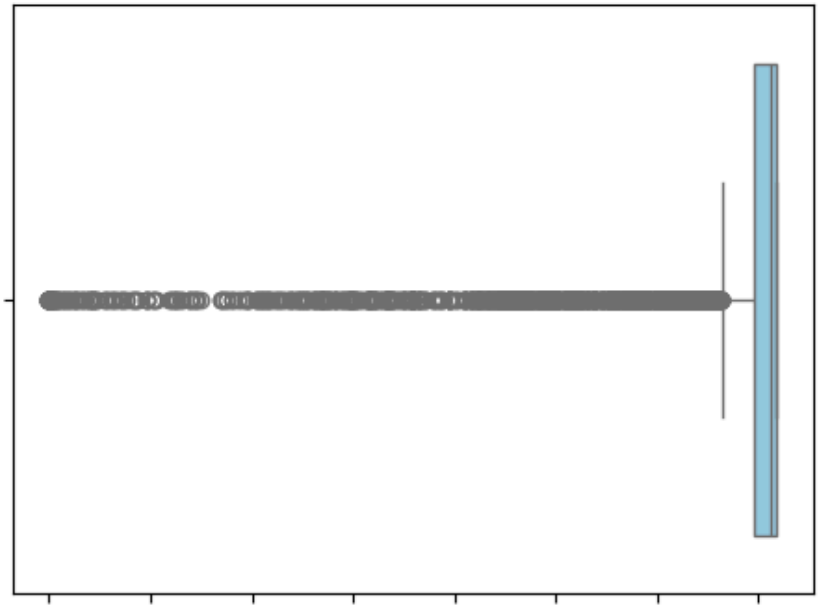




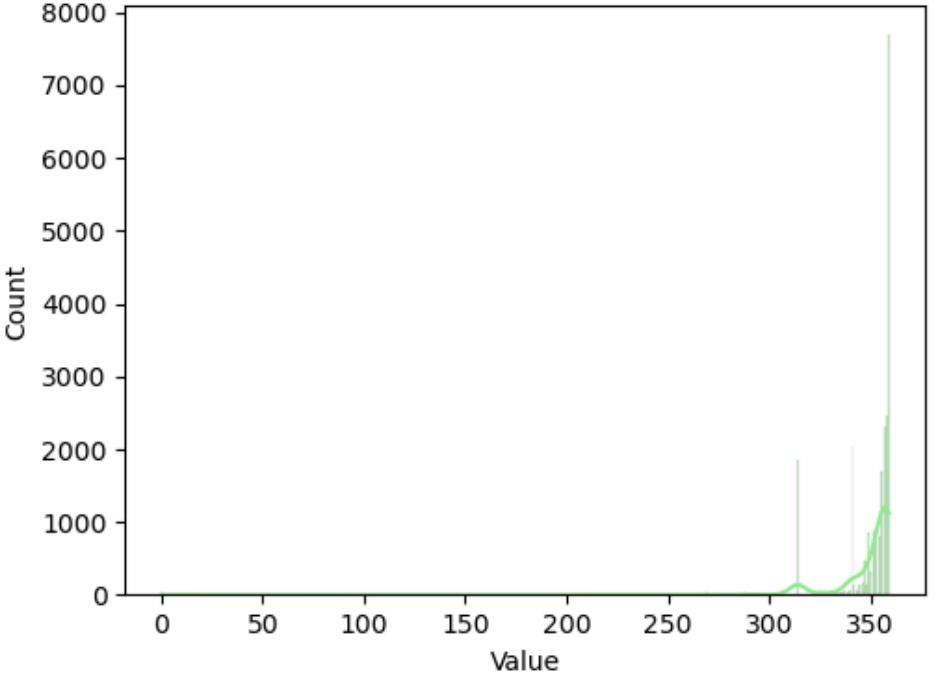




MaximumWindDirection — Boxplot

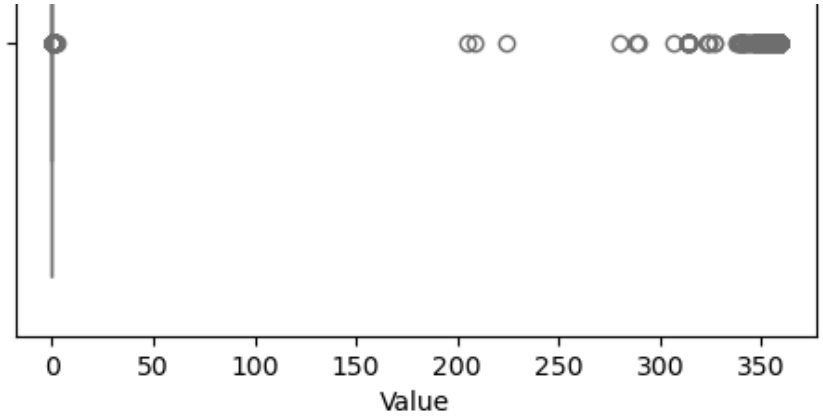


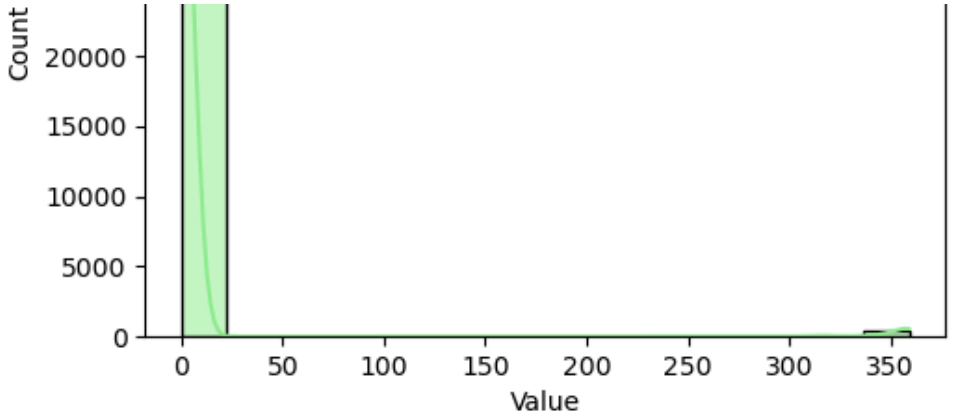
MaximumWindDirection — Distribution



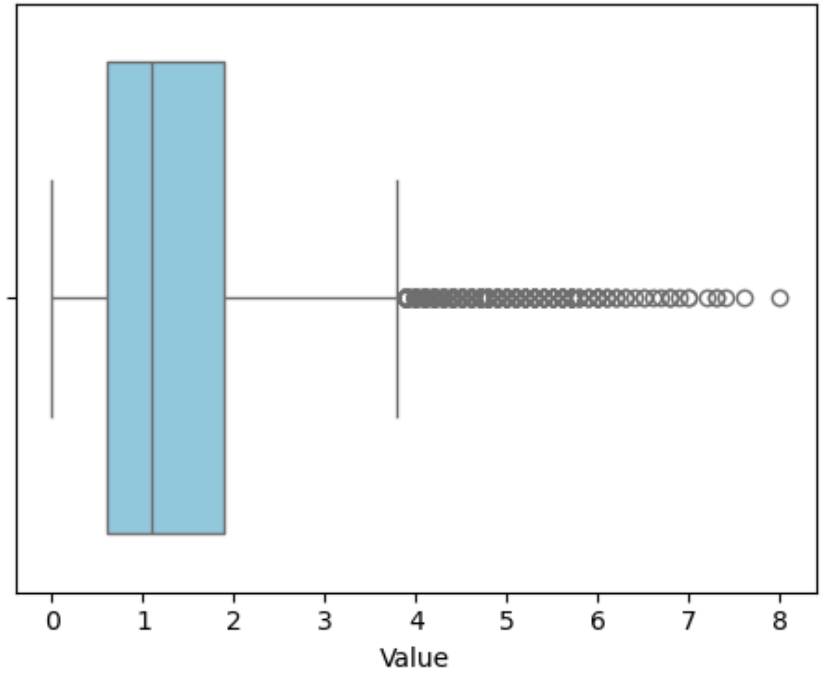
MinimumWindSpeed — Boxplot

MinimumWindSpeed — Distribution

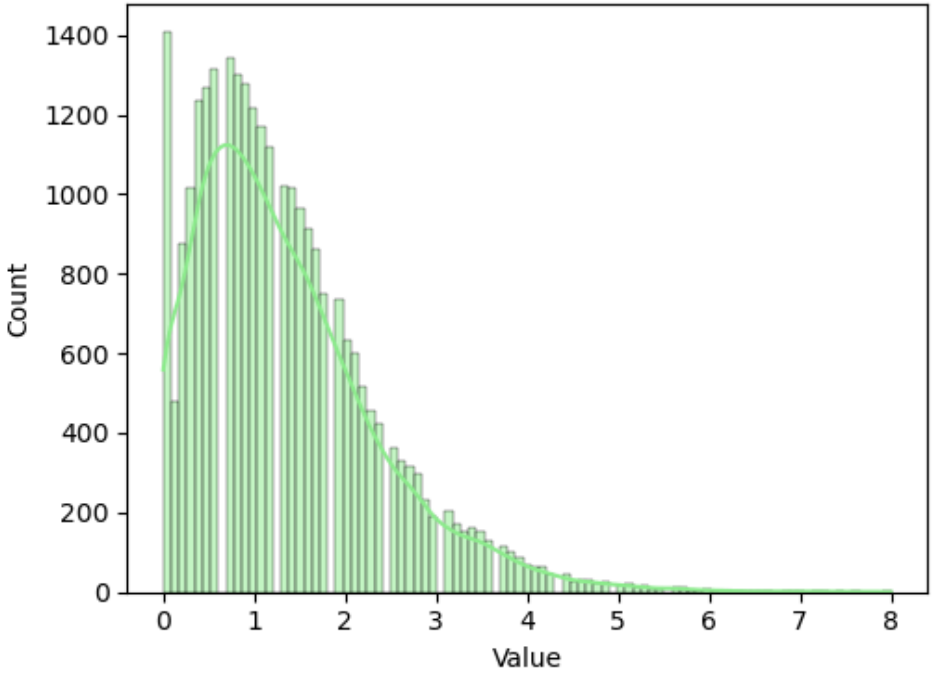




AverageWindSpeed — Boxplot

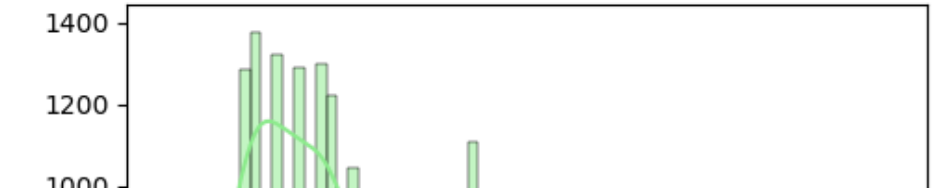


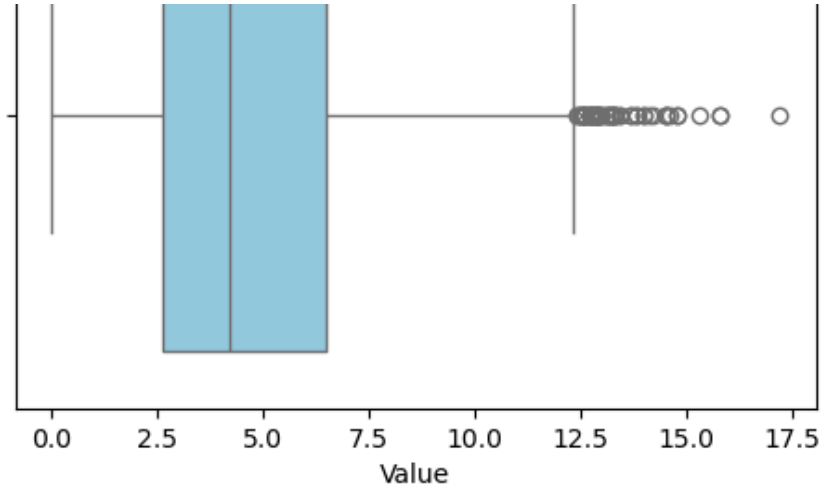
AverageWindSpeed — Distribution

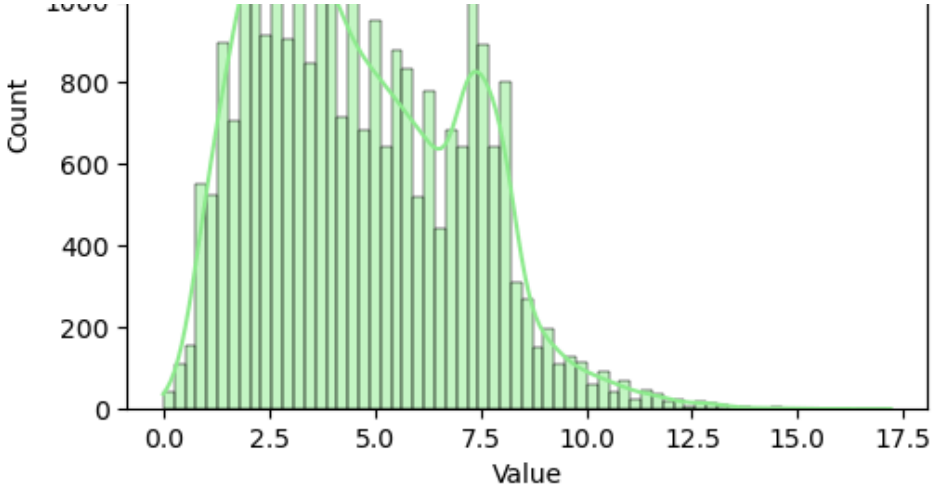


GustWindSpeed — Boxplot

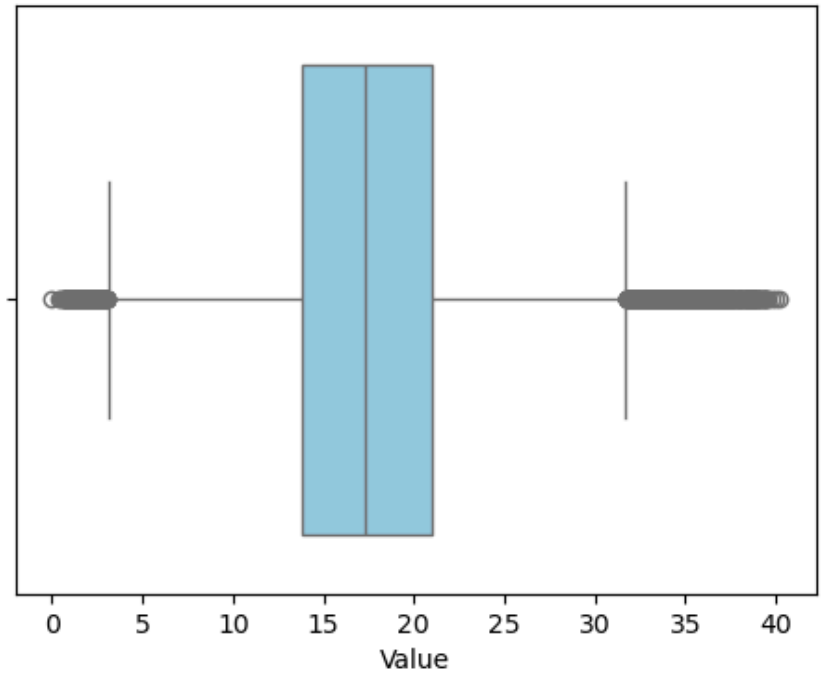
GustWindSpeed — Distribution



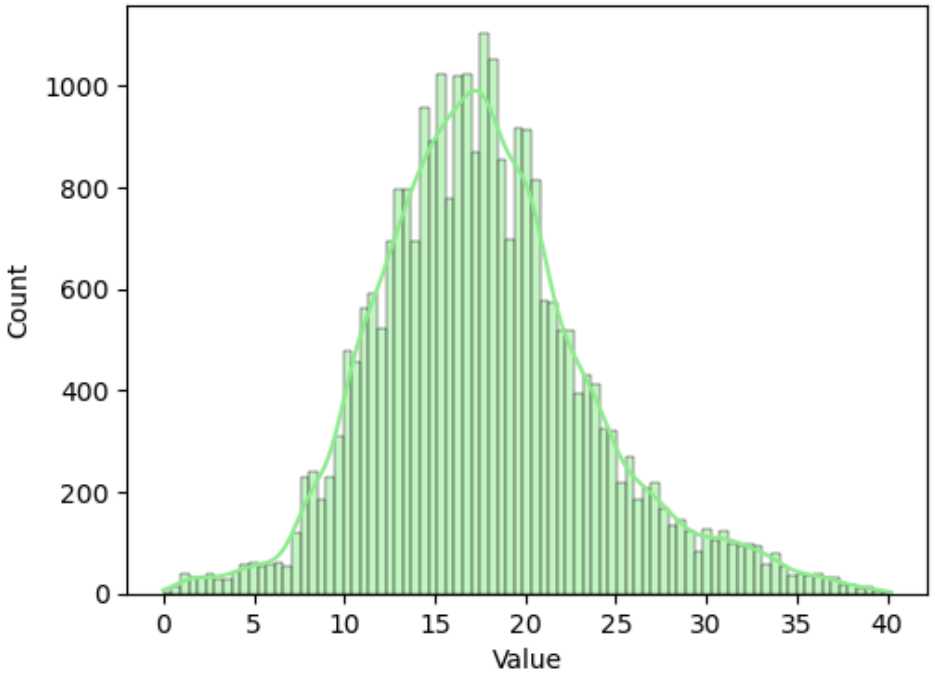




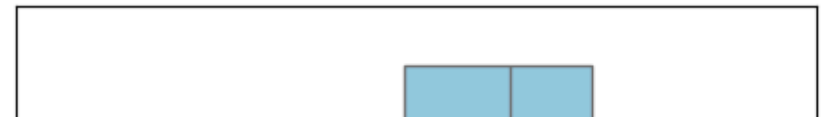
AirTemperature — Boxplot



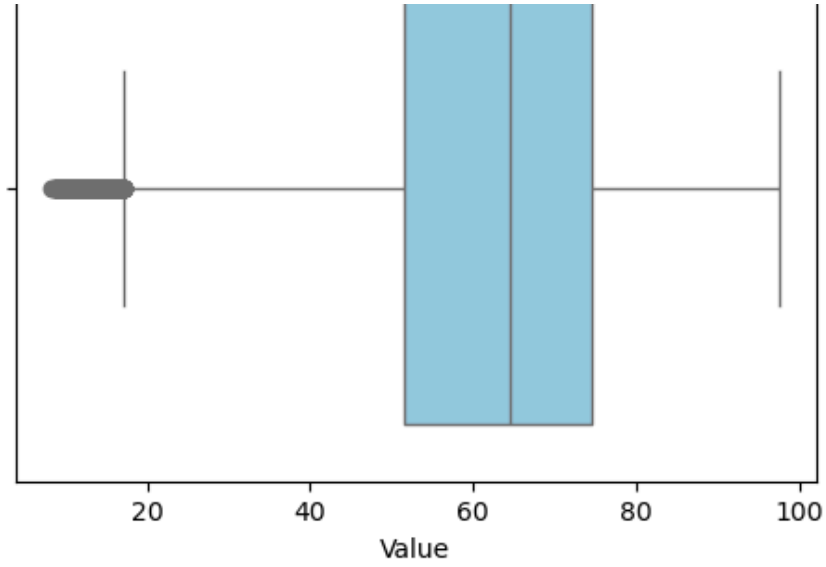
AirTemperature — Distribution

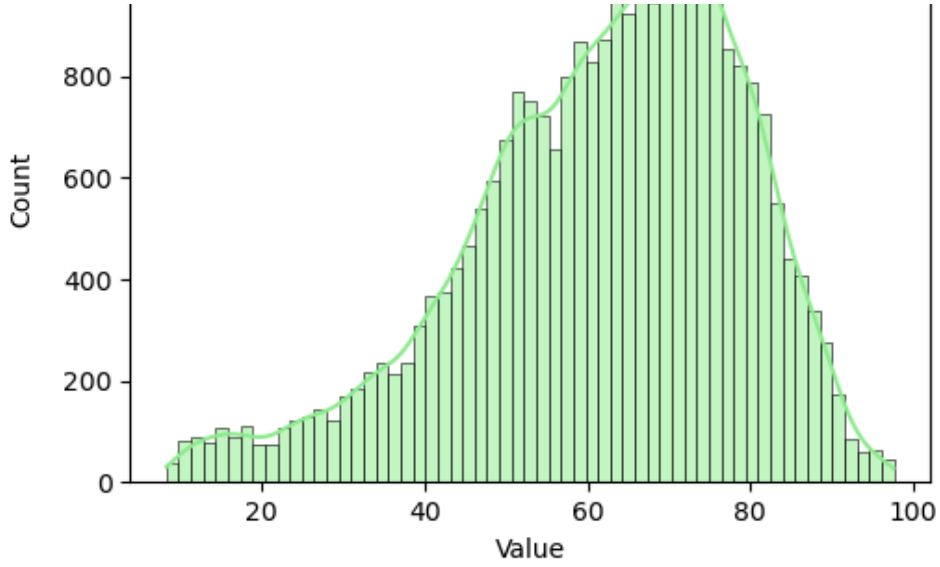


RelativeHumidity — Boxplot

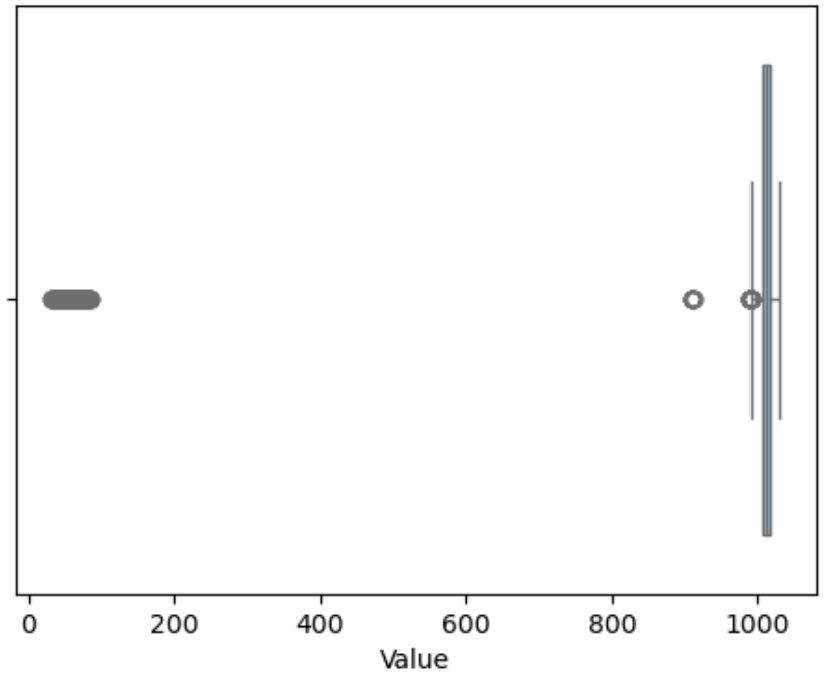


RelativeHumidity — Distribution

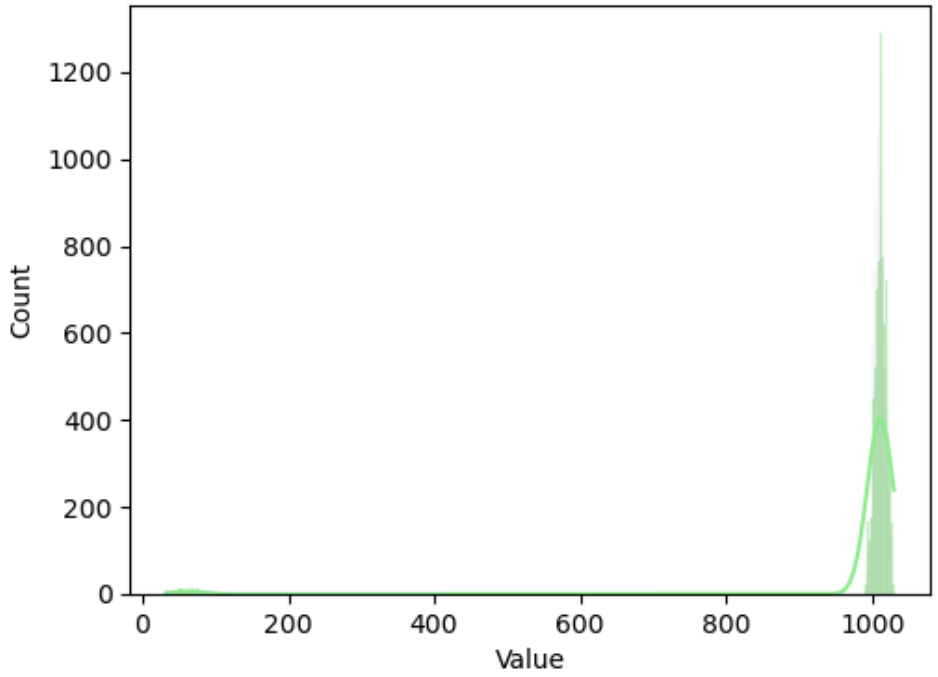




AtmosphericPressure — Boxplot

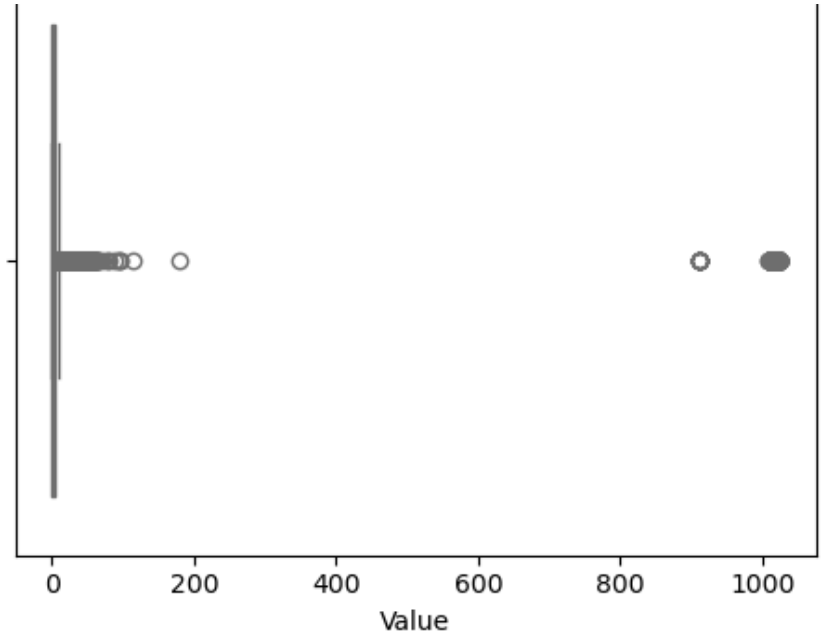


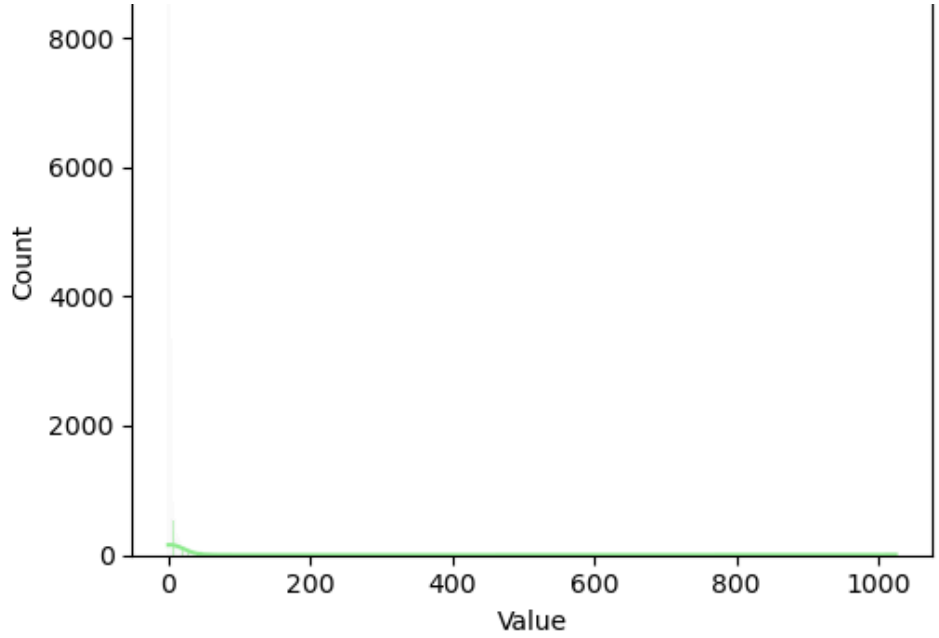
AtmosphericPressure — Distribution



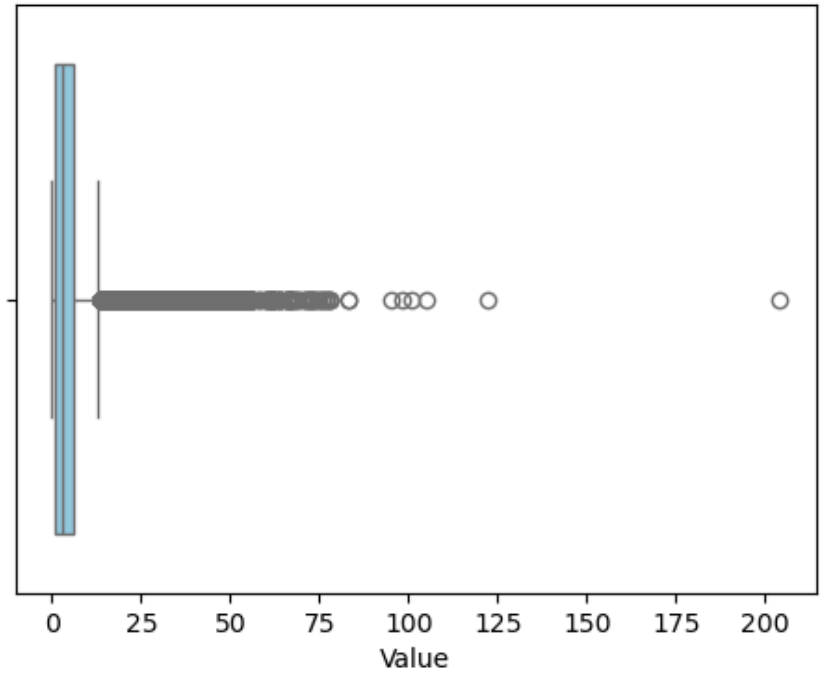
PM25 — Boxplot

PM25 — Distribution

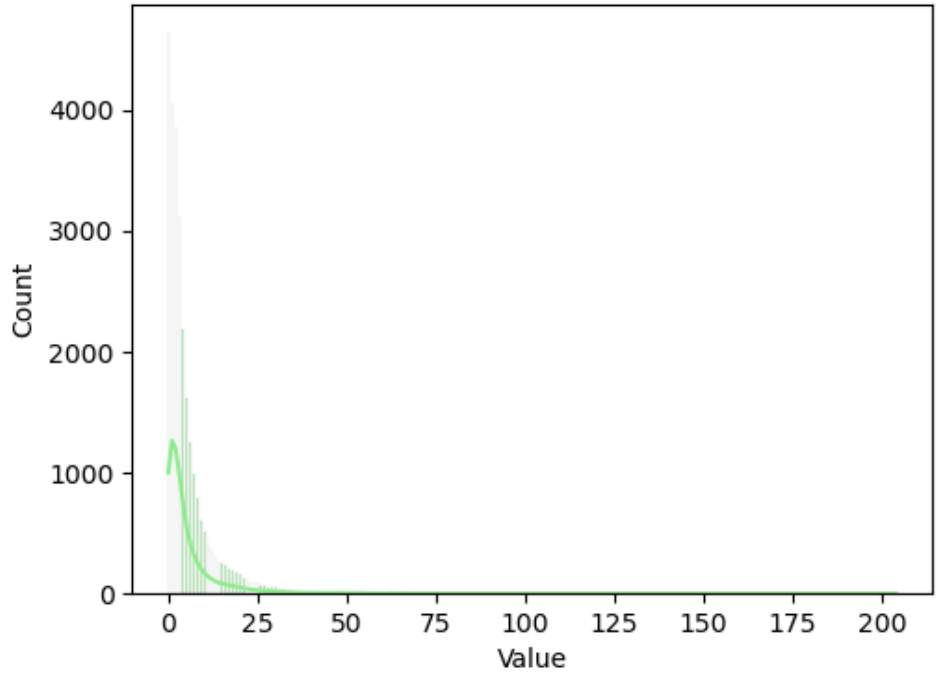


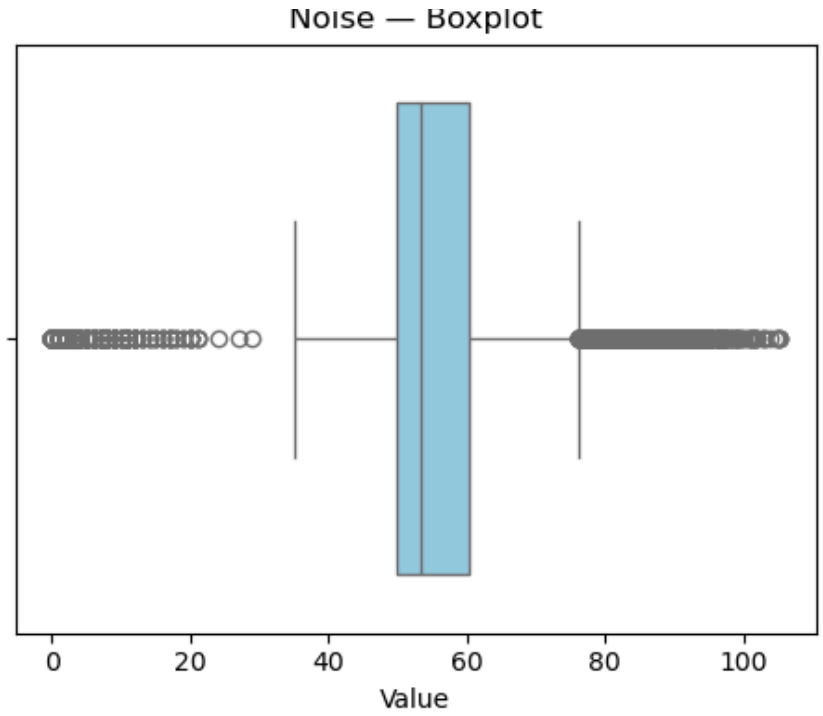


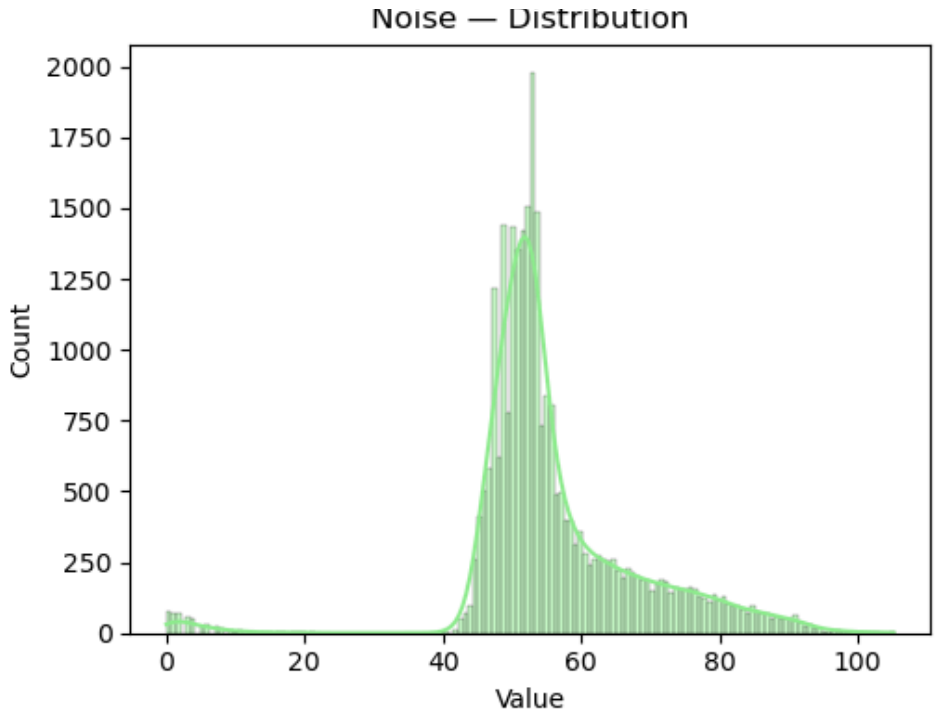
PM10 — Boxplot



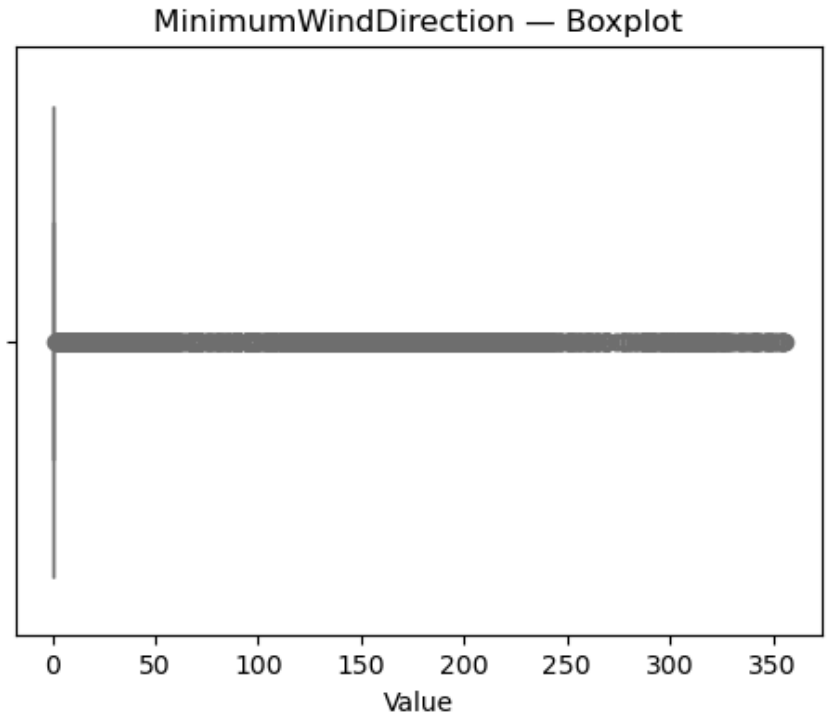
PM10 — Distribution

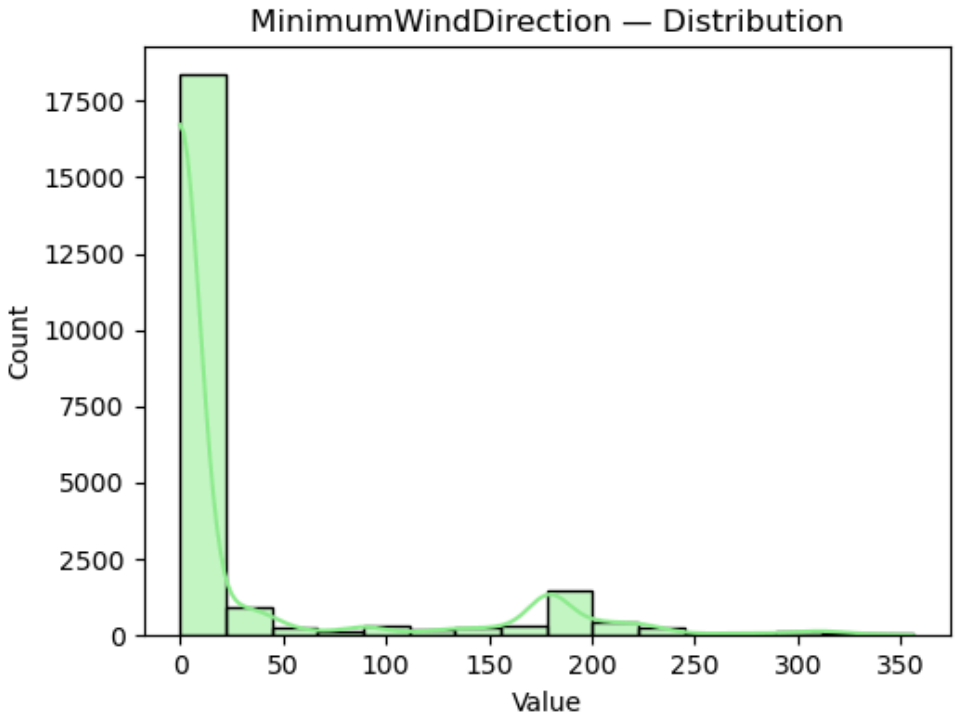


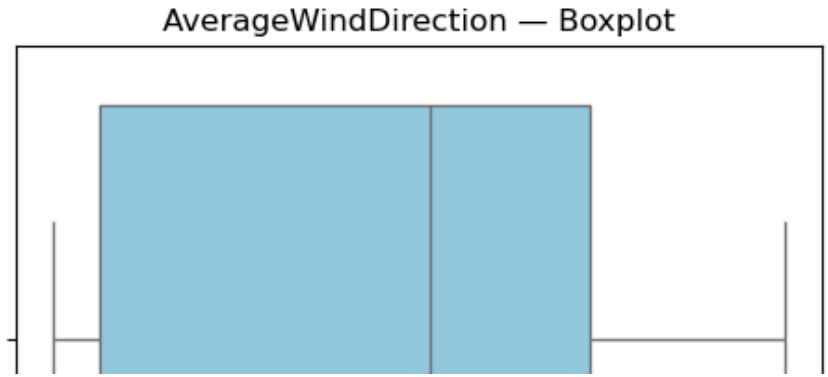

Noise — Boxplot
0
20
40
60
80
100
Value

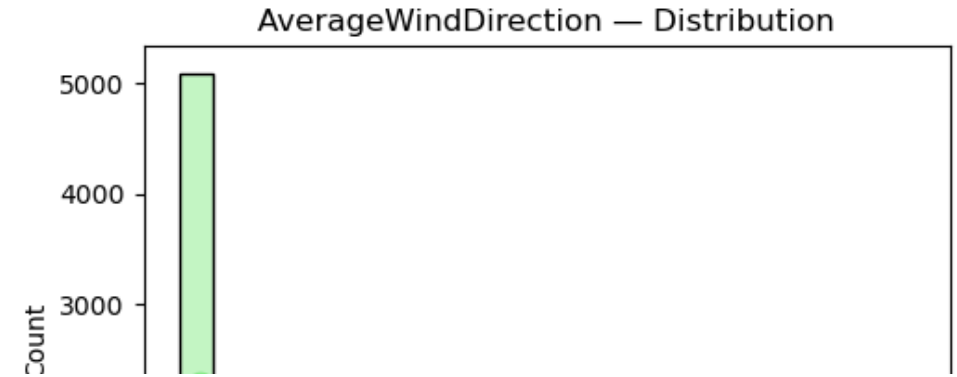

Noise — Distribution
2000
1750
1500
1250
1000
750
500
250
0
Count
0
20
40
60
80
100
Value

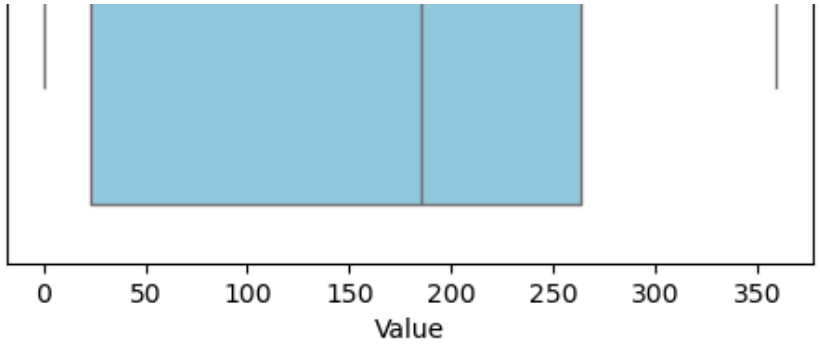
Device: ICTMicroclimate-05

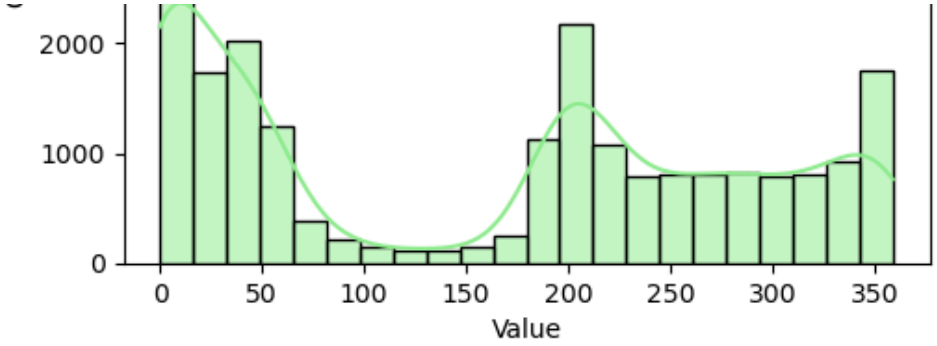




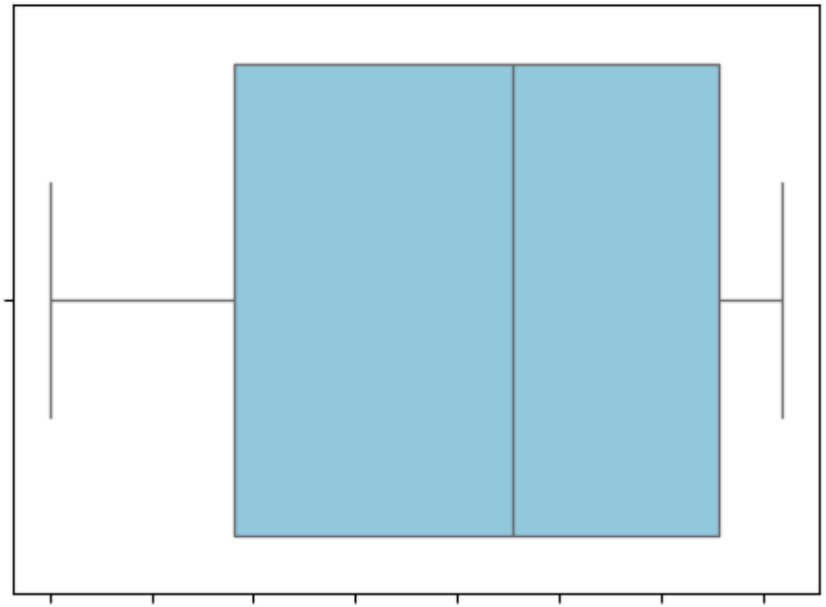




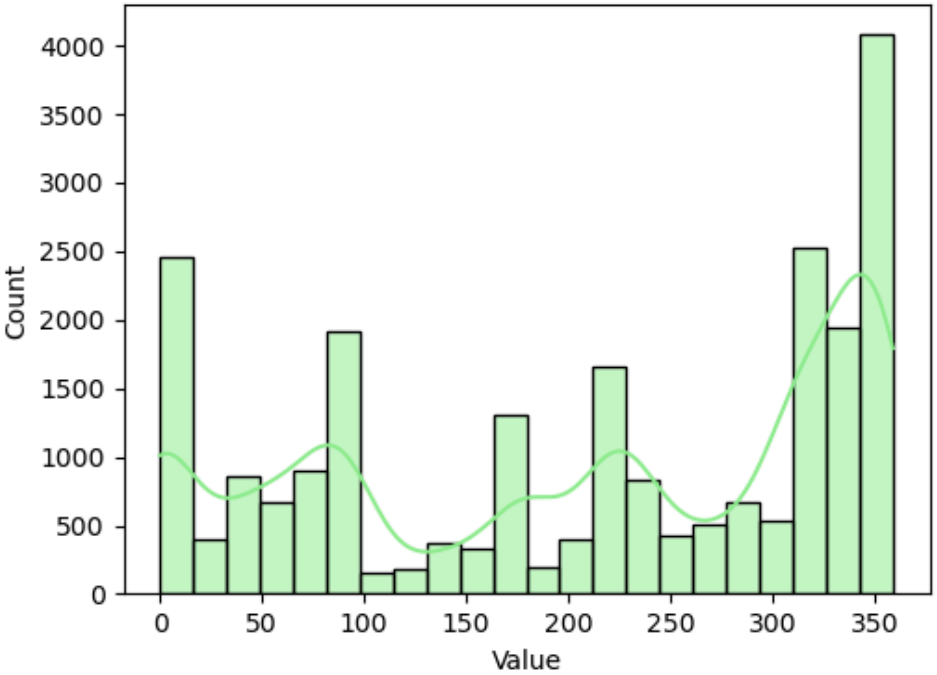




MaximumWindDirection — Boxplot

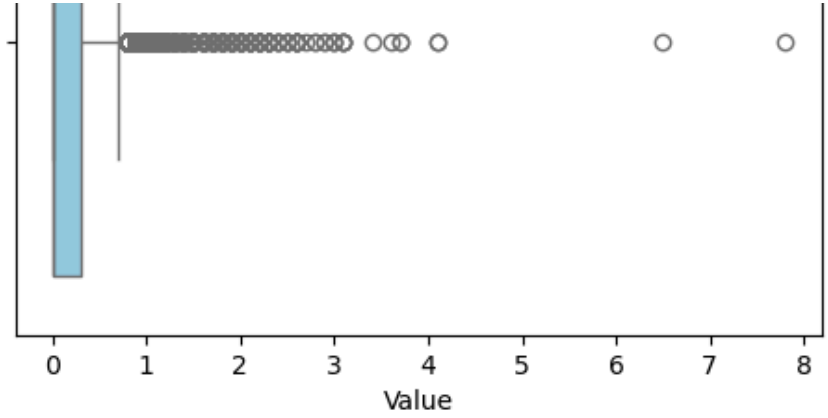


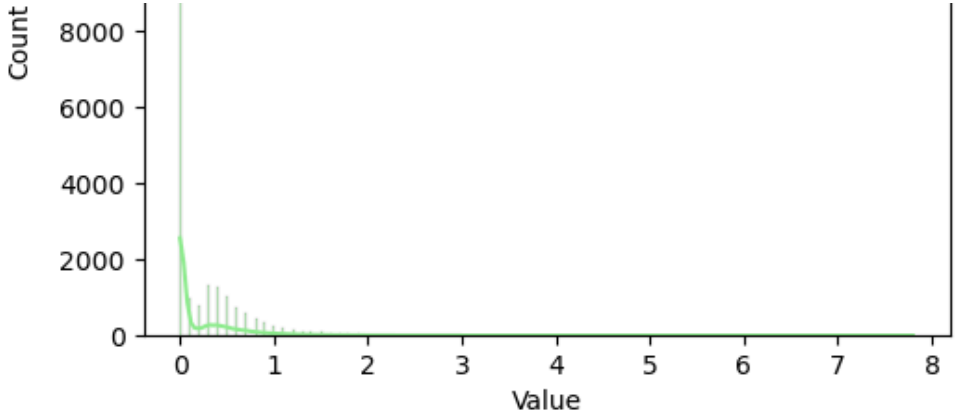
MaximumWindDirection — Distribution



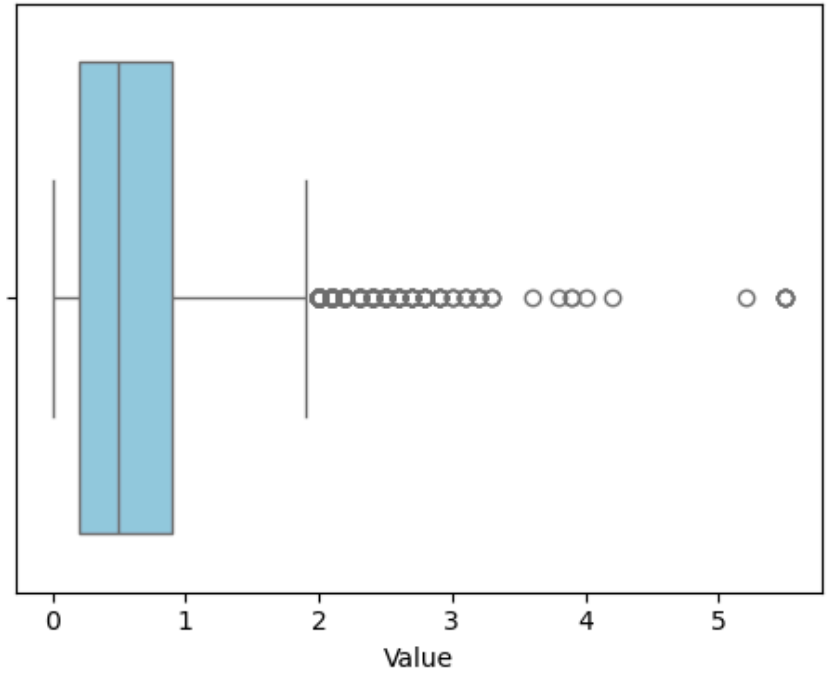
MinimumWindSpeed — Boxplot

MinimumWindSpeed — Distribution

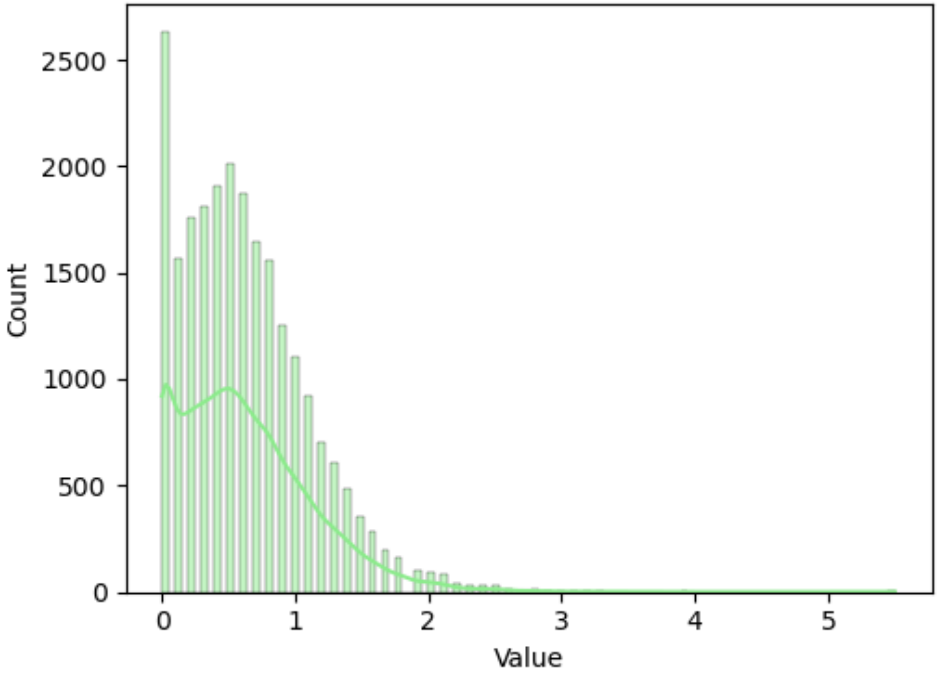




AverageWindSpeed — Boxplot

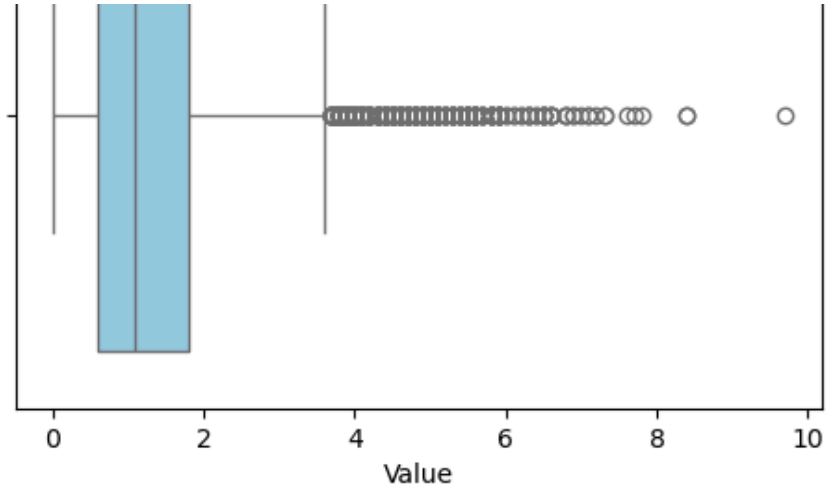


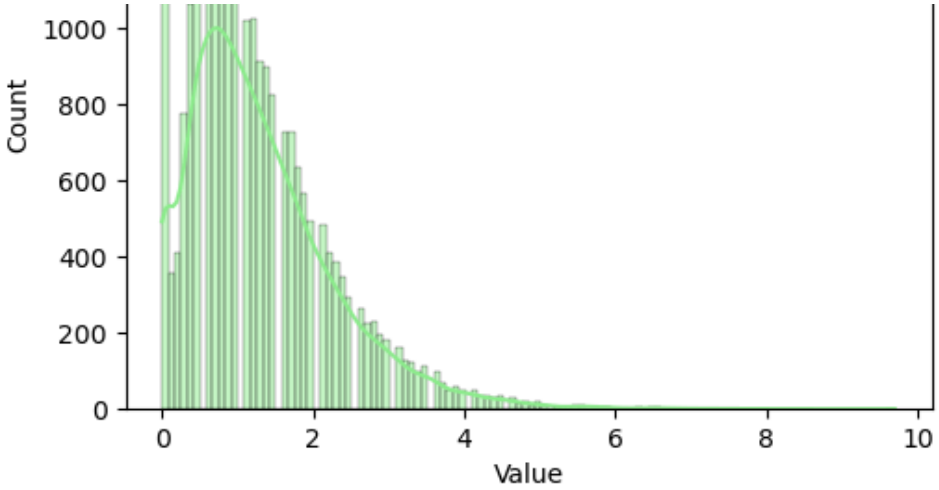
AverageWindSpeed — Distribution



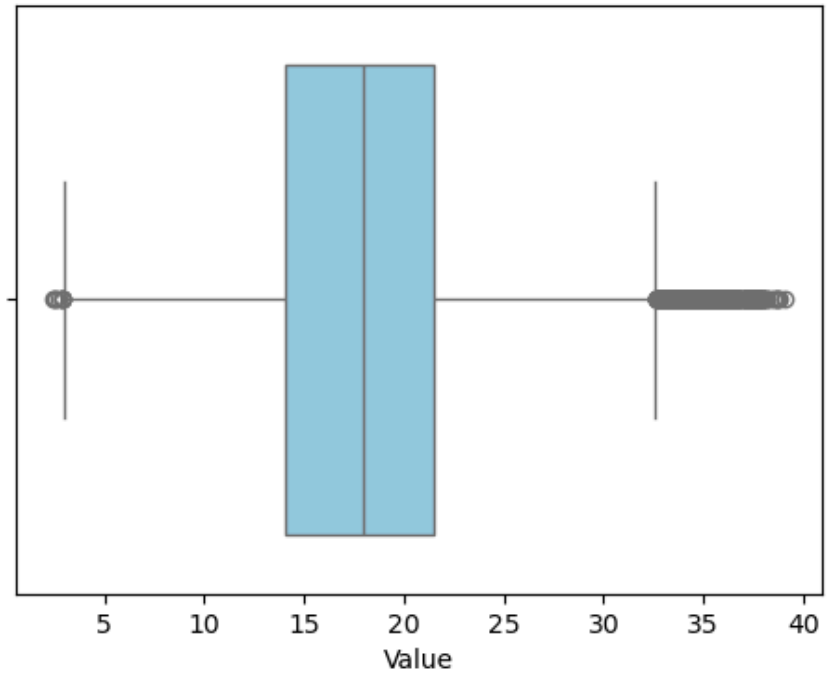
GustWindSpeed — Boxplot

GustWindSpeed — Distribution

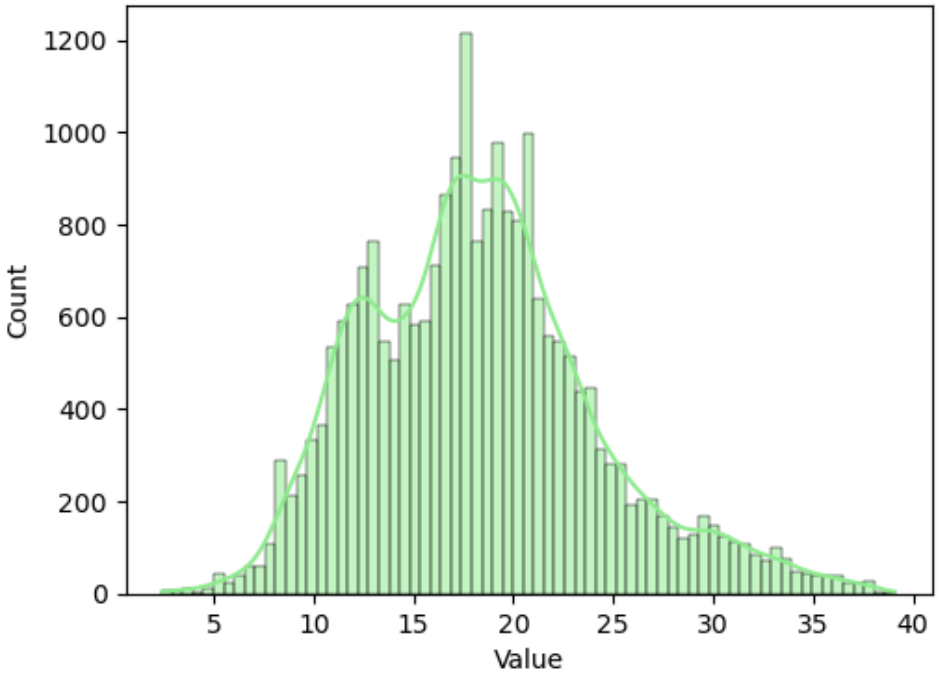




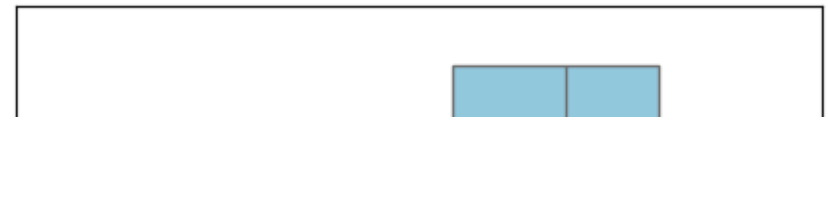
AirTemperature — Boxplot



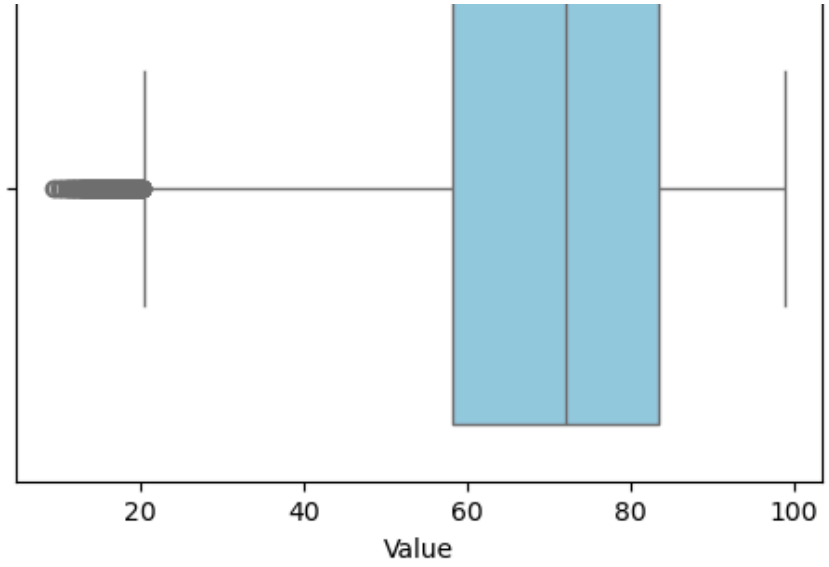
AirTemperature — Distribution

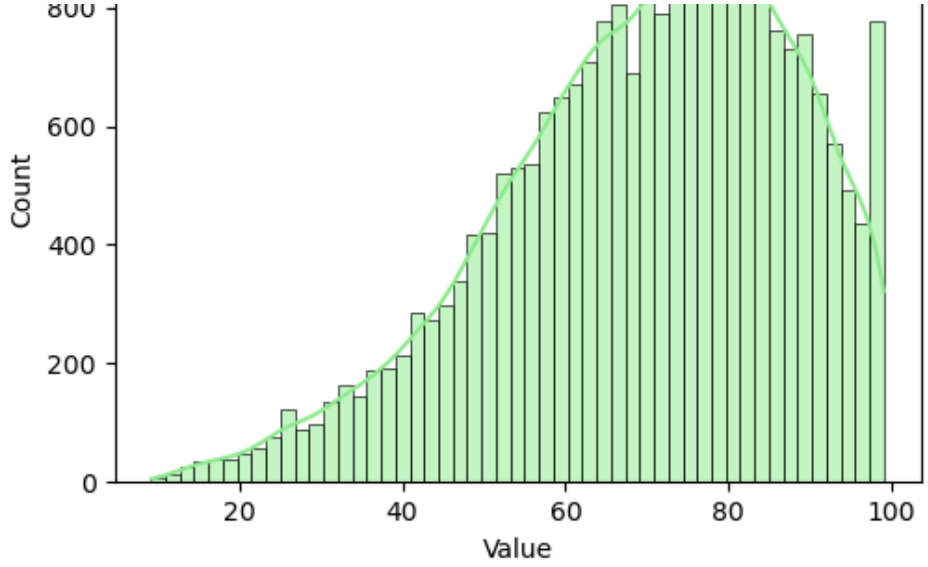


RelativeHumidity — Boxplot

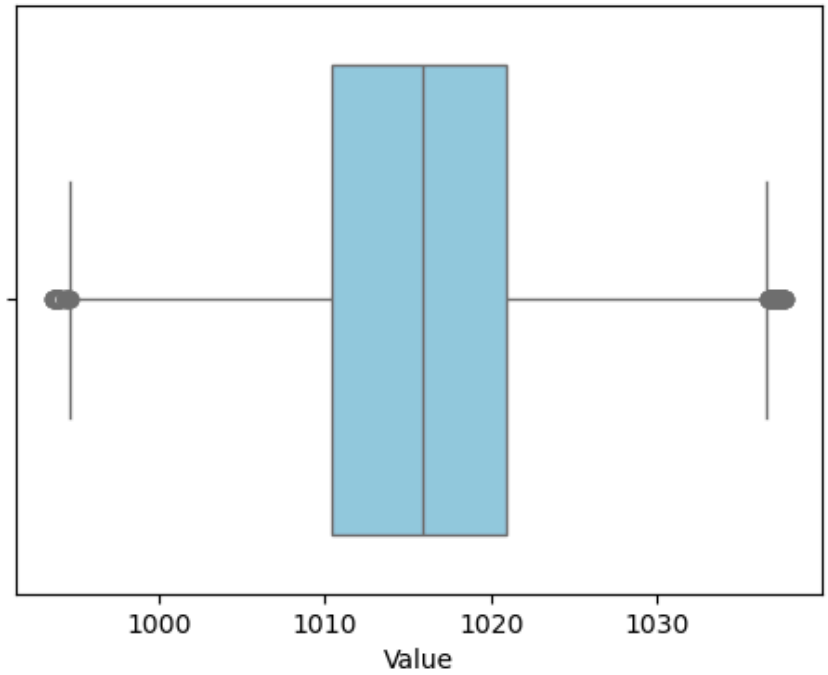


RelativeHumidity — Distribution

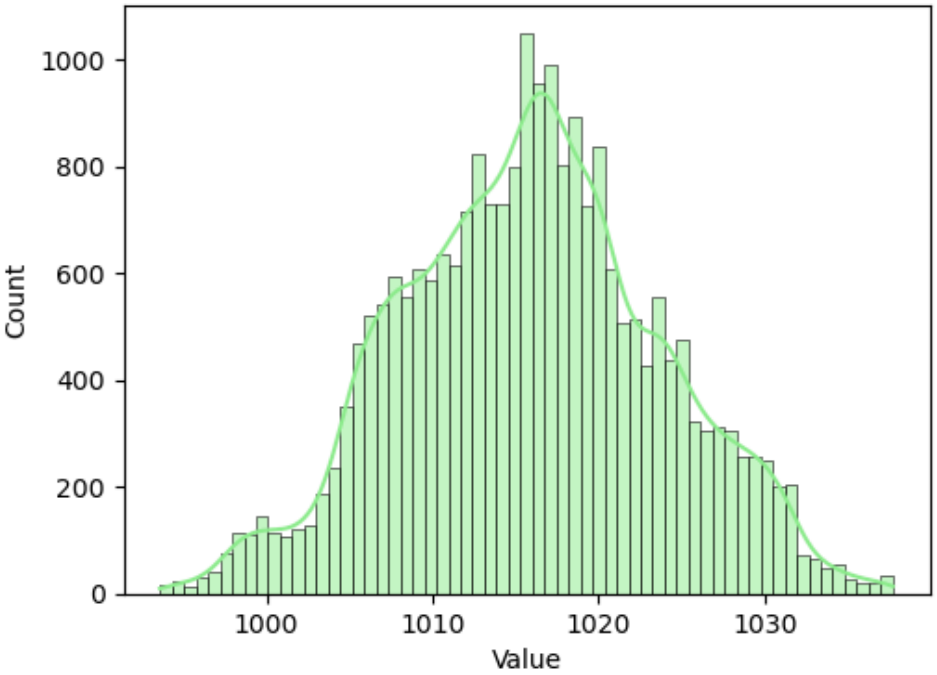




AtmosphericPressure — Boxplot

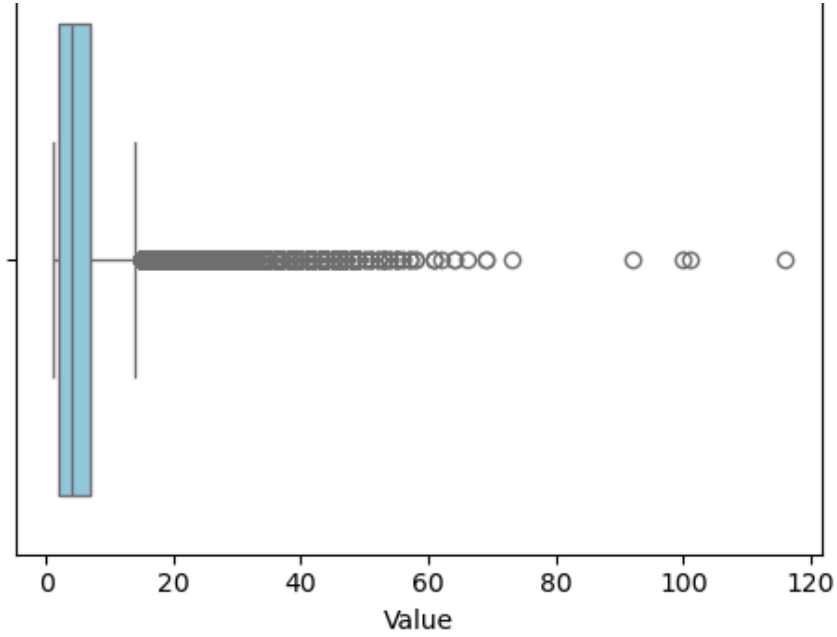


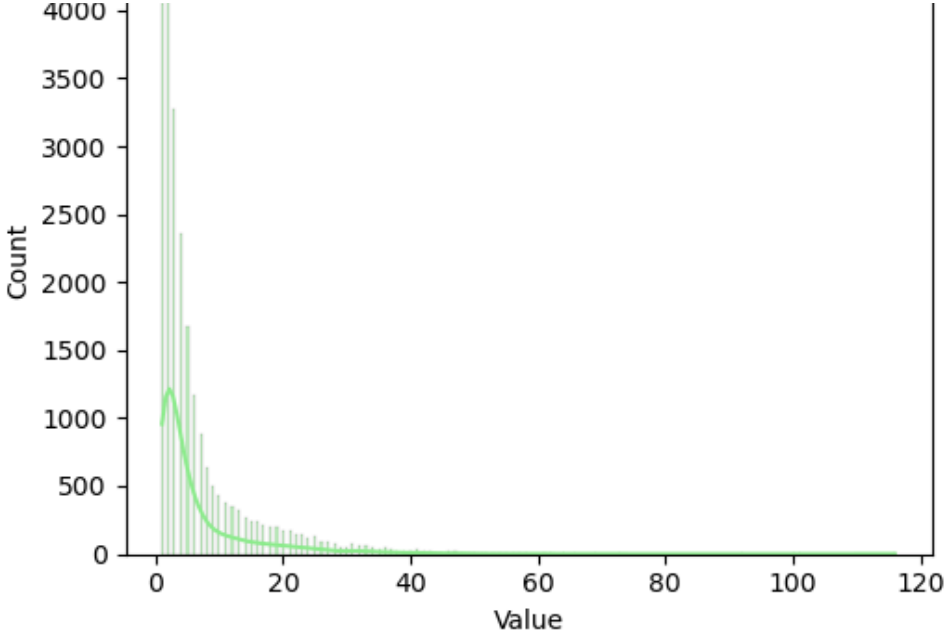
AtmosphericPressure — Distribution



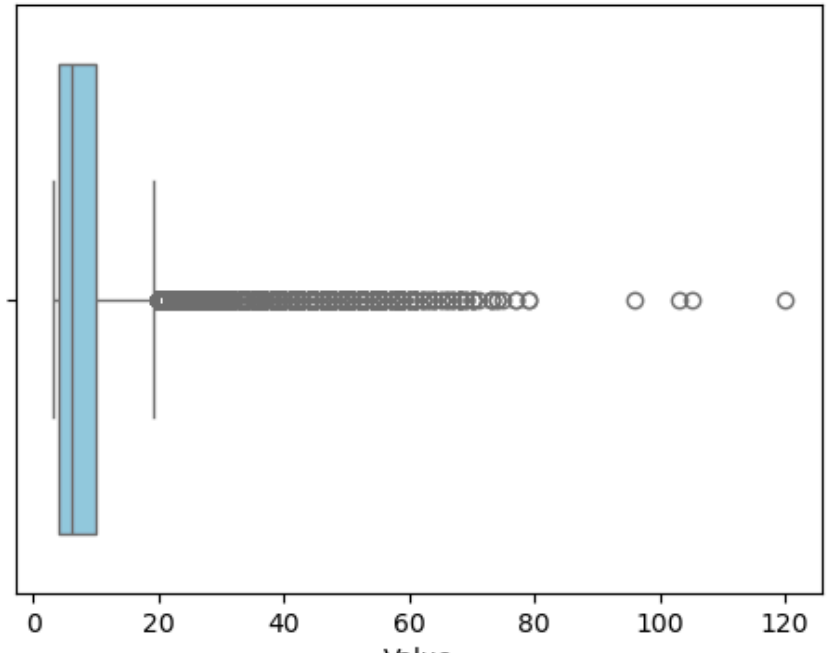
PM25 — Boxplot

PM25 — Distribution

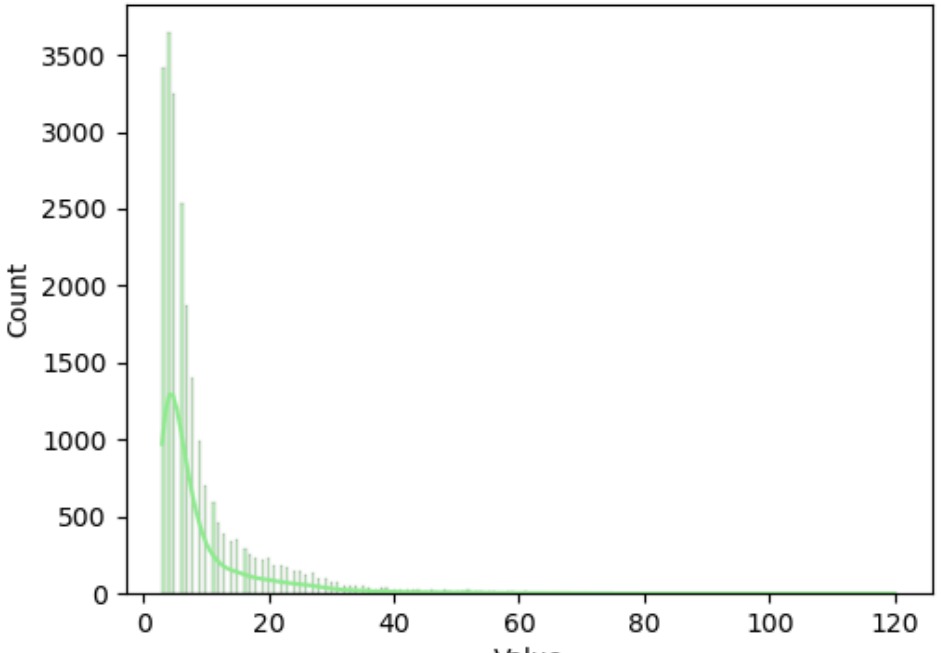


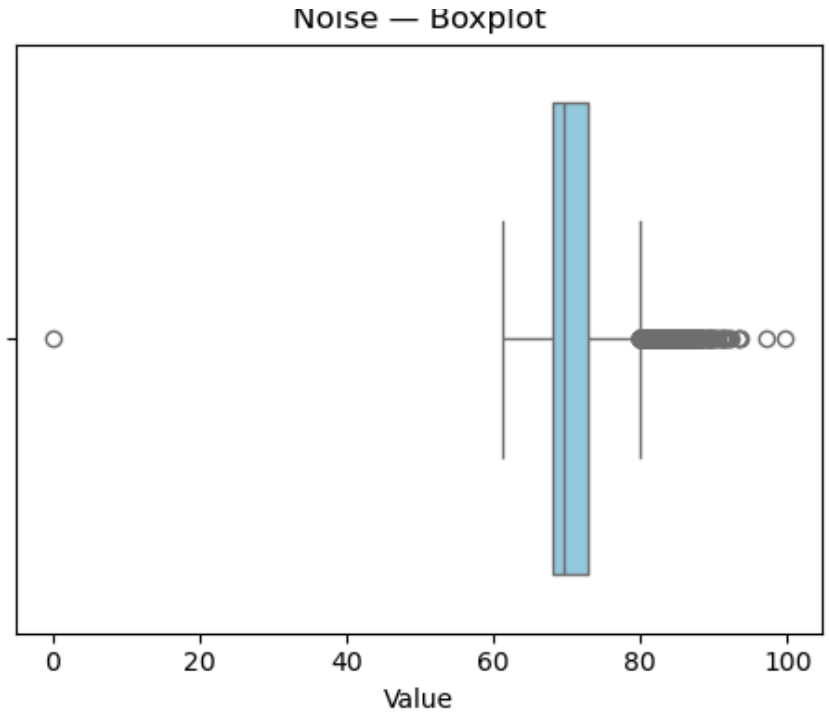


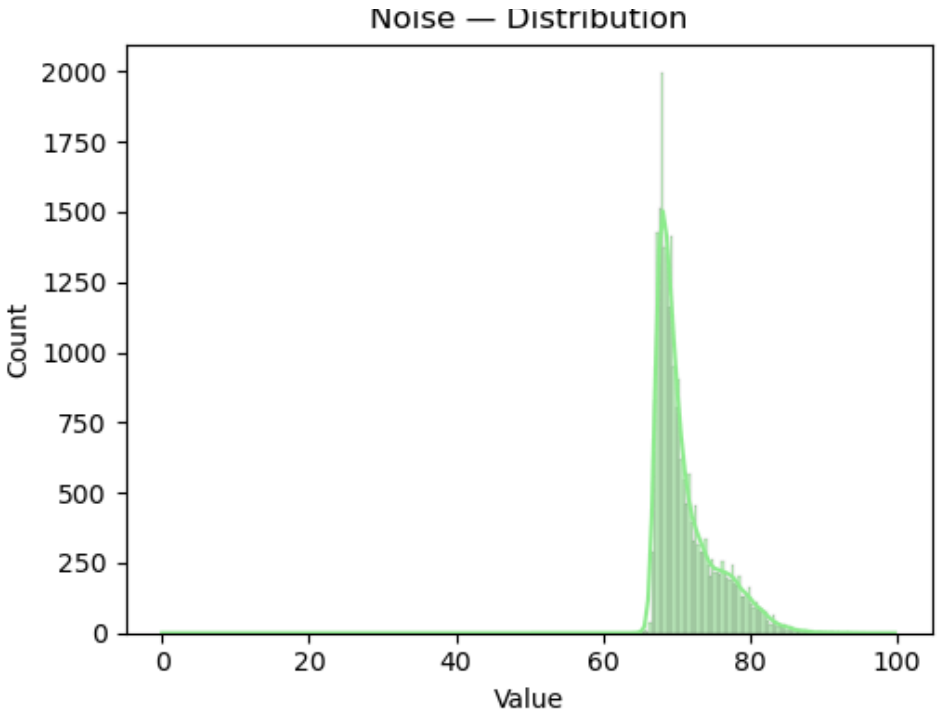
PM10 — Boxplot



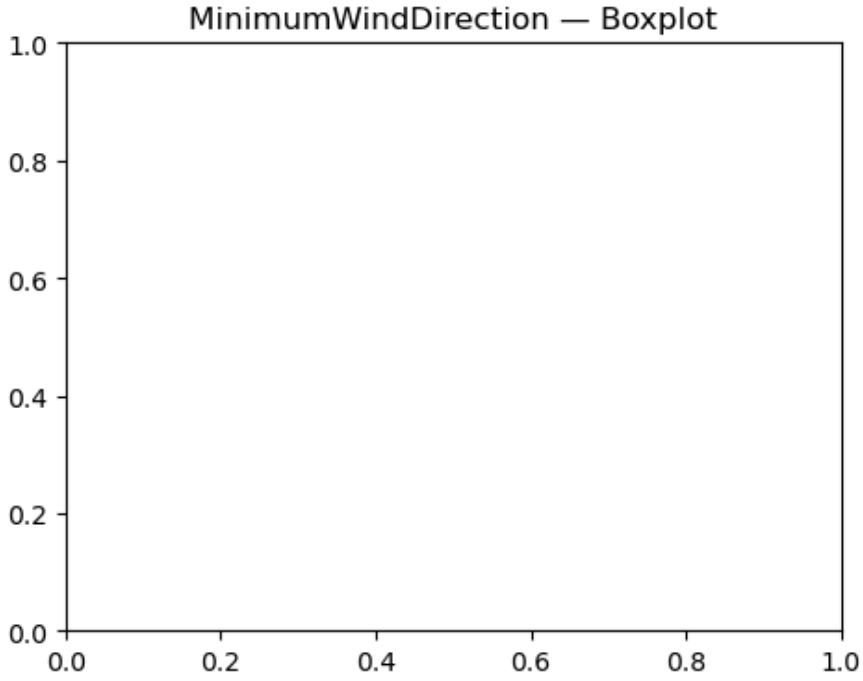
PM10 — Distribution

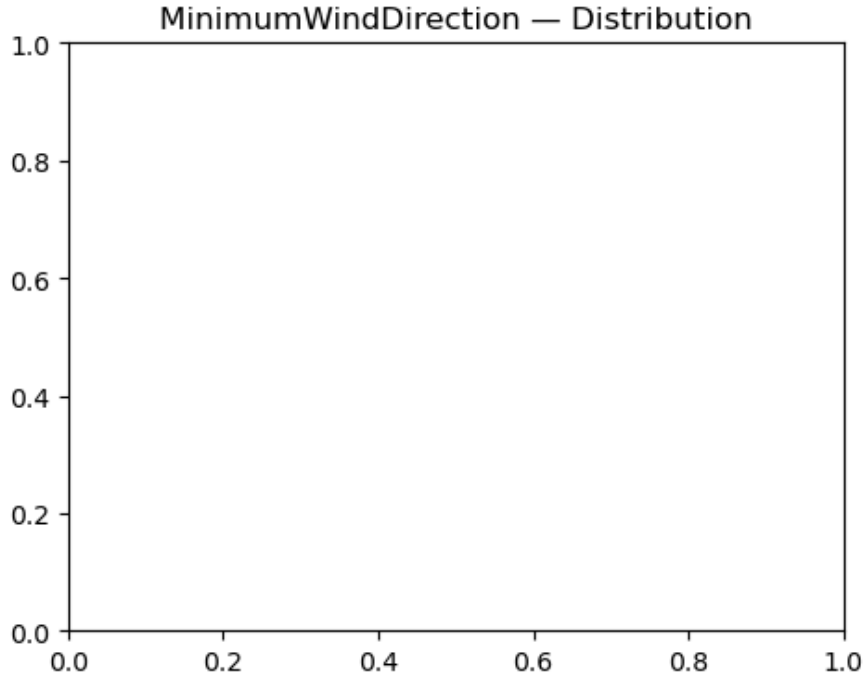


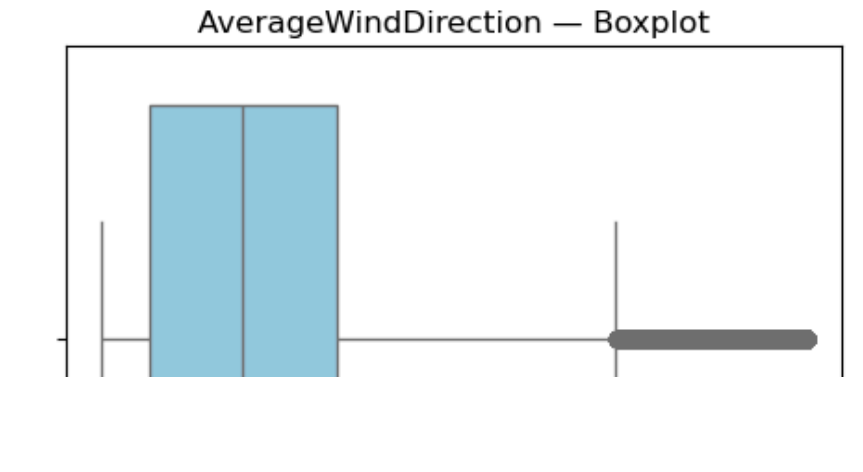

Noise — Boxplot
0
20
40
60
80
100
Value

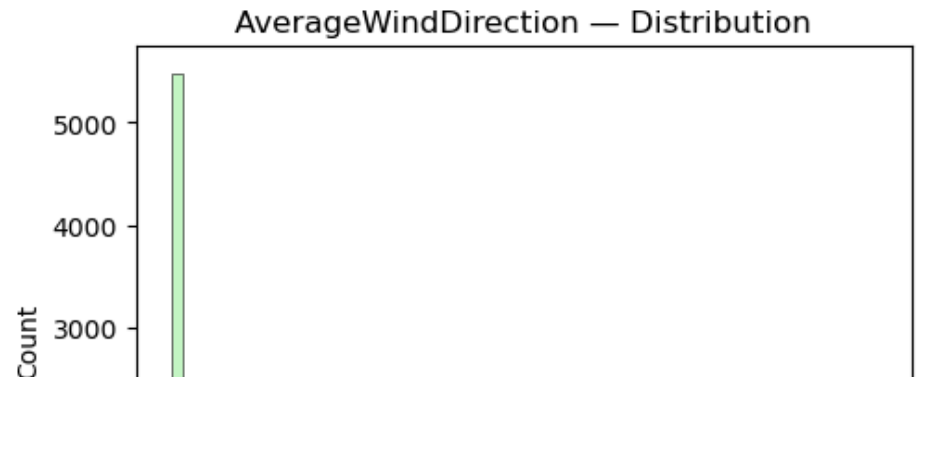

Noise — Distribution
2000
1750
1500
1250
1000
750
500
250
0
Count
0
20
40
60
80
100
Value

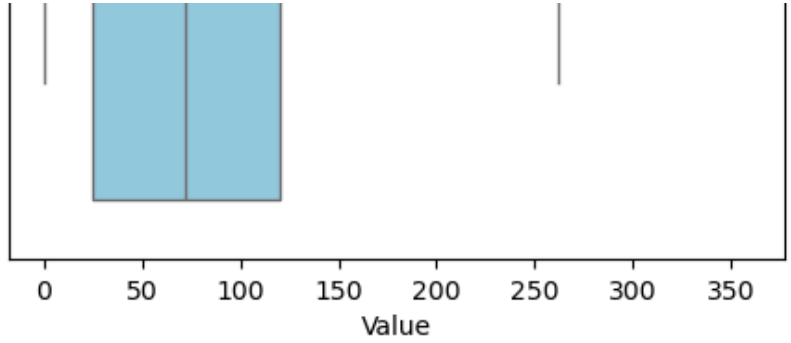
Device: ICTMicroclimate-01

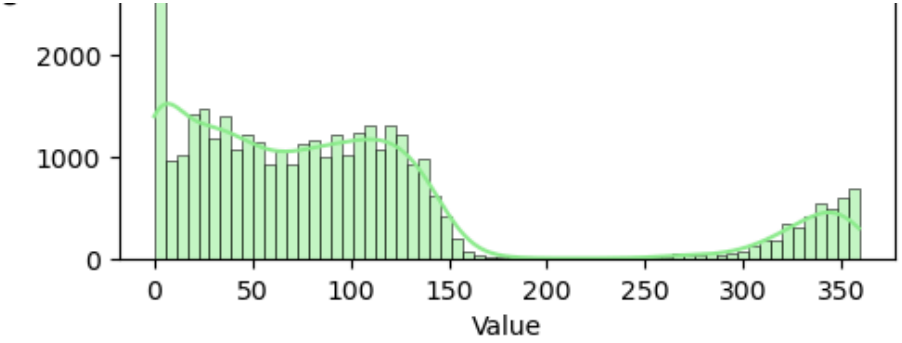


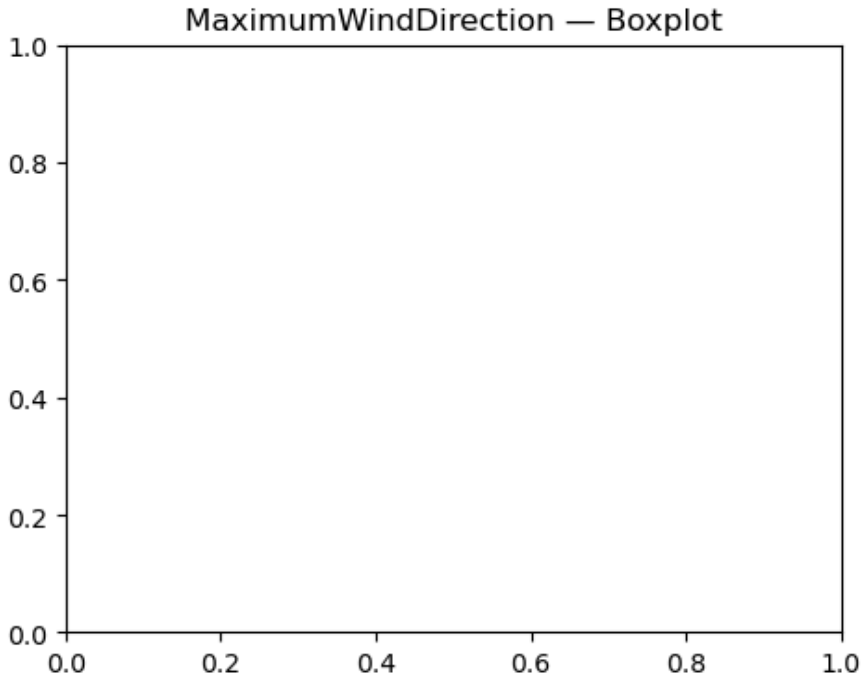


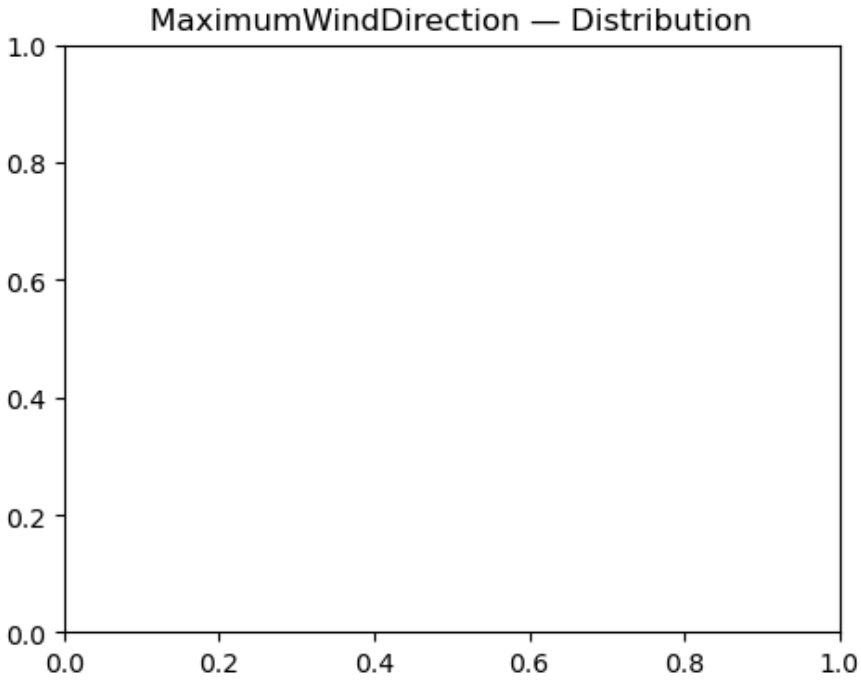


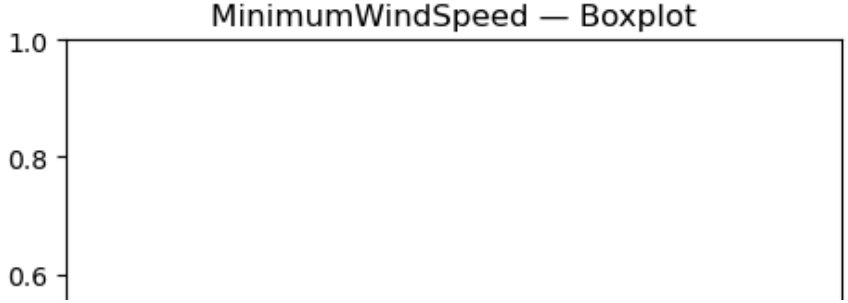


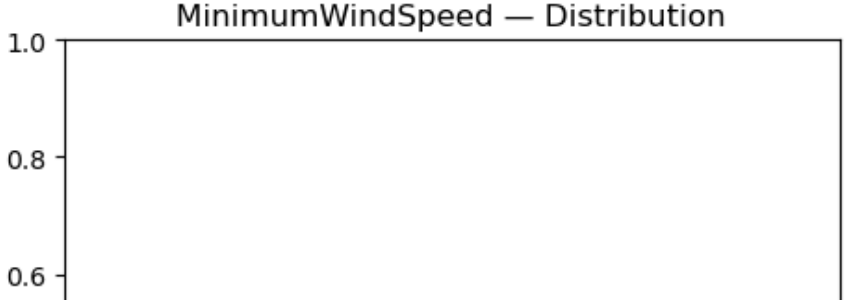

0
50
100
150
200
250
300
350
Value

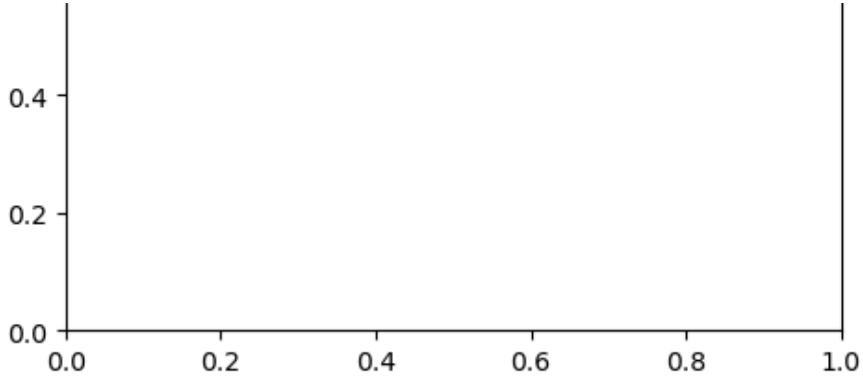

2000
1000
0
0
50
100
150
200
250
300
350
Value

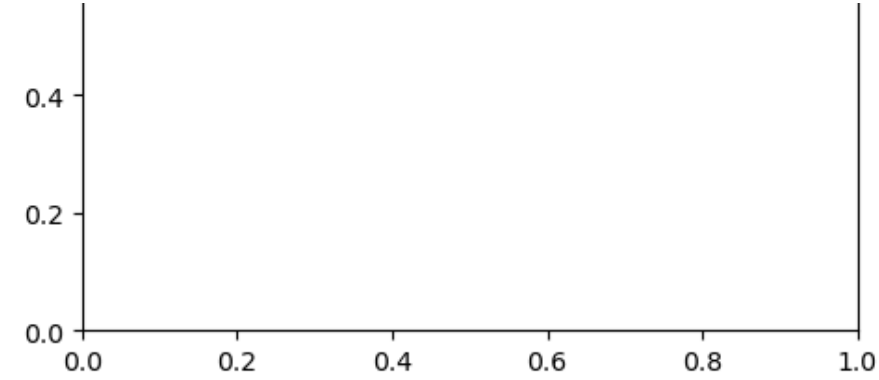

MaximumWindDirection — Boxplot
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

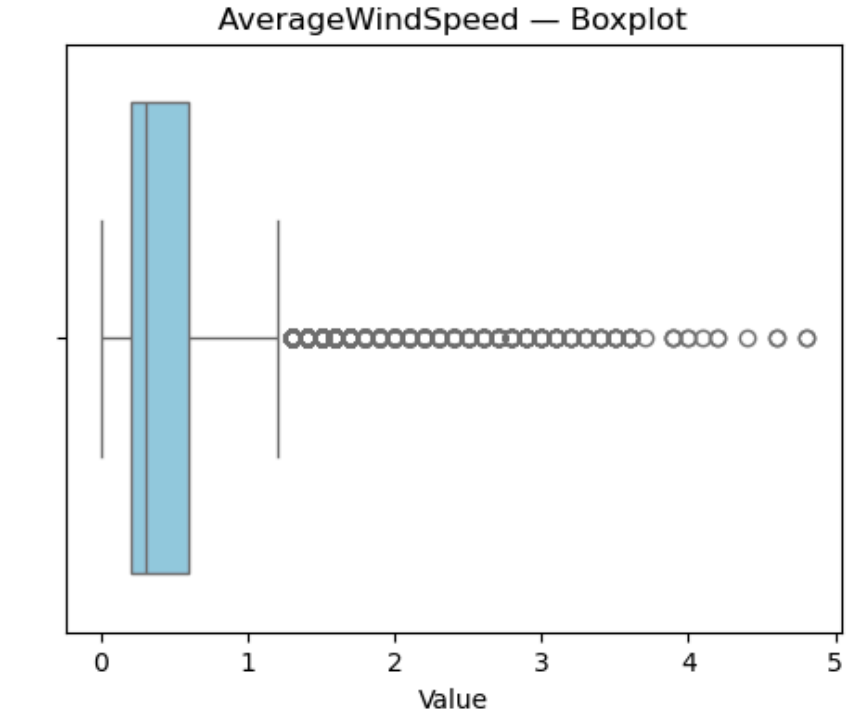

MaximumWindDirection — Distribution
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

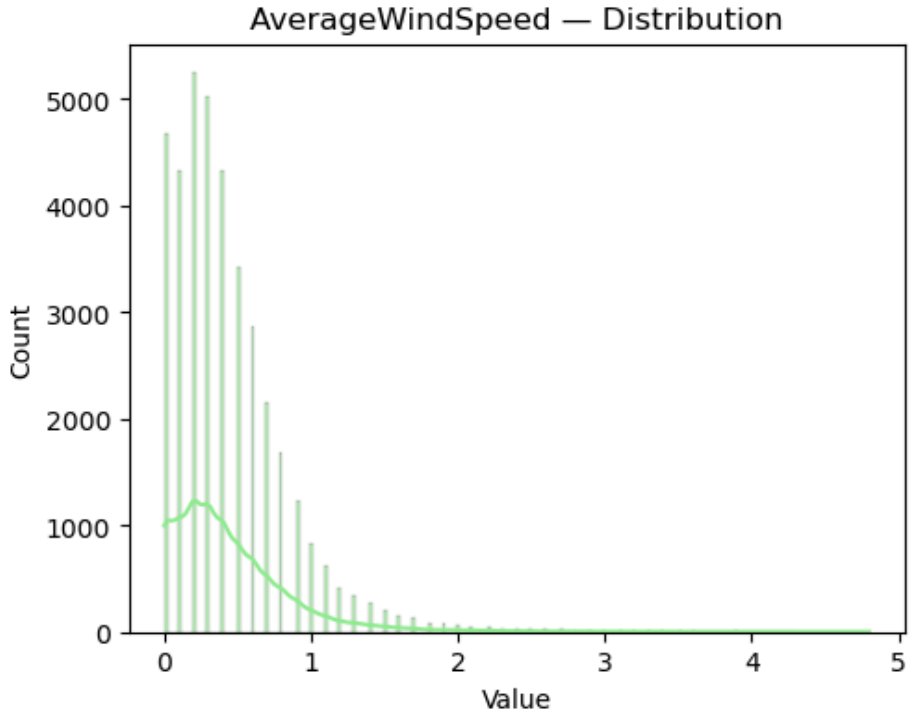

MinimumWindSpeed — Boxplot
1.0
0.8
0.6


MinimumWindSpeed — Distribution
1.0
0.8
0.6

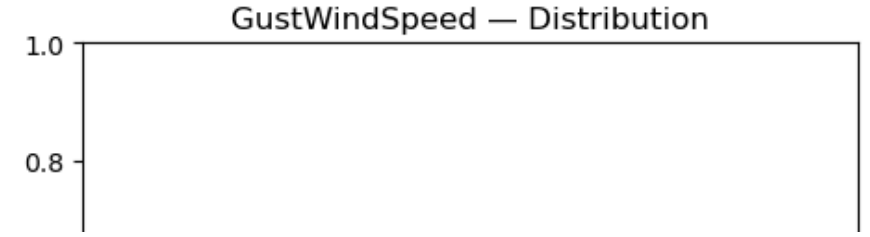

0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

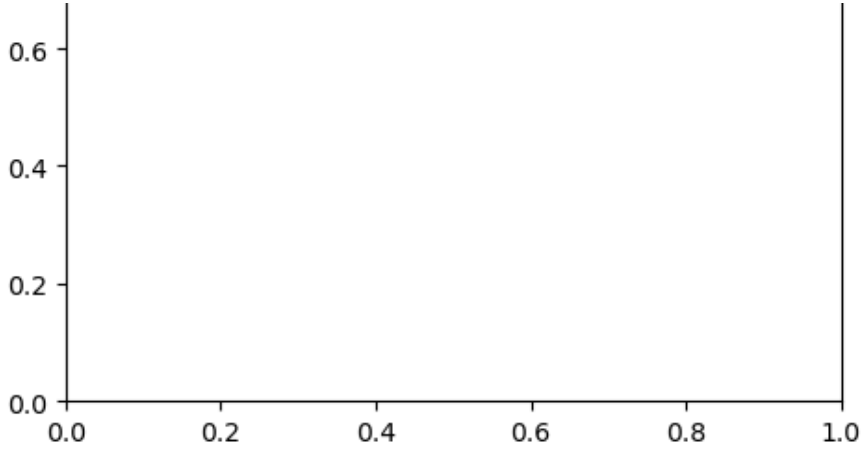

0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

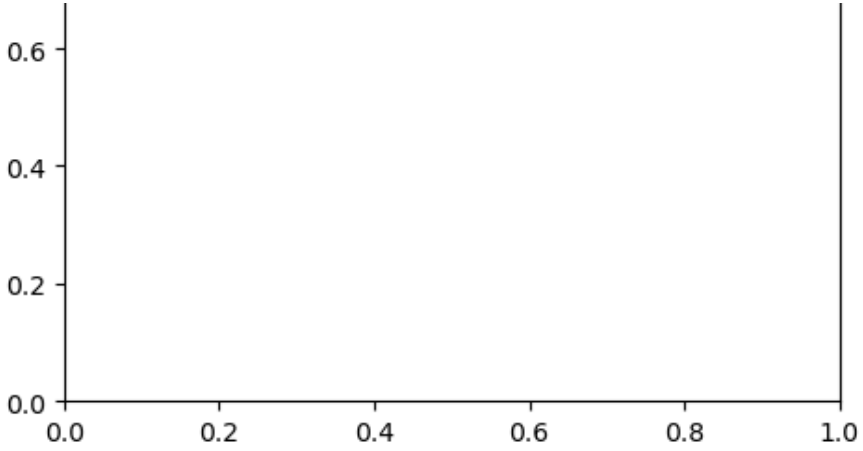

AverageWindSpeed — Boxplot
0
1
2
3
4
5
Value

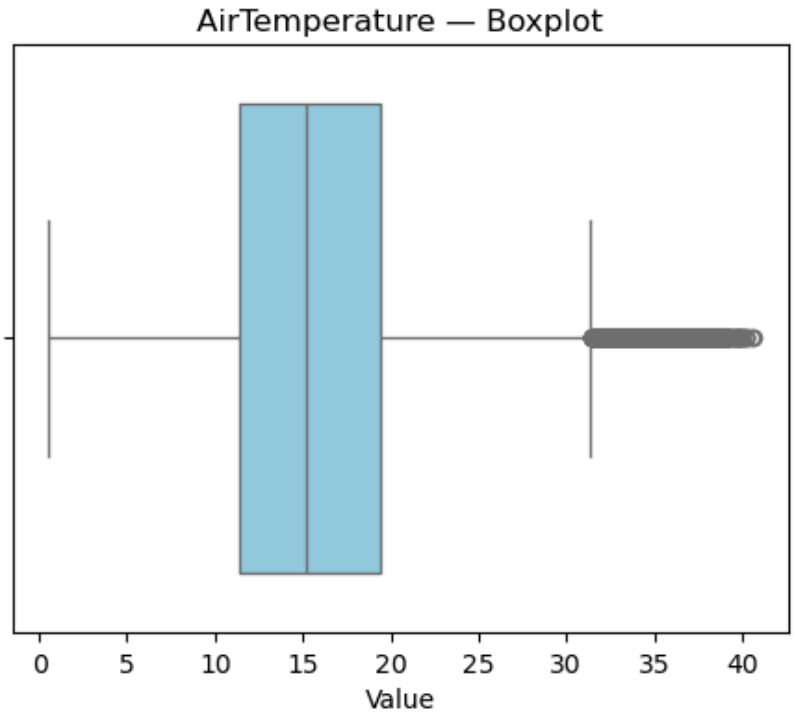

AverageWindSpeed — Distribution
5000
4000
3000
2000
1000
0
Count
0
1
2
3
4
5
Value

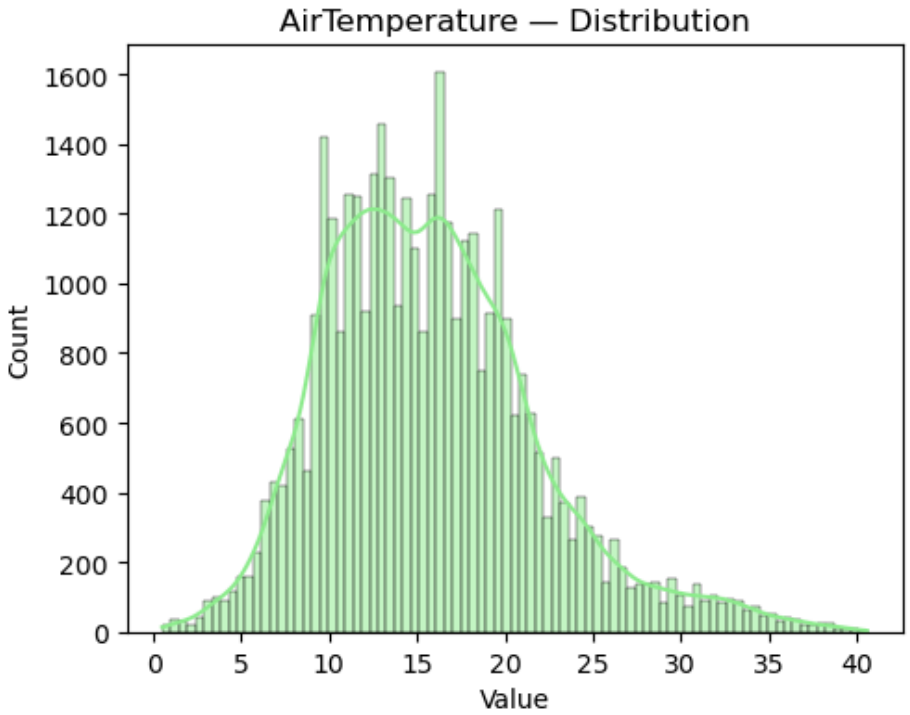

GustWindSpeed — Boxplot
1.0
0.8

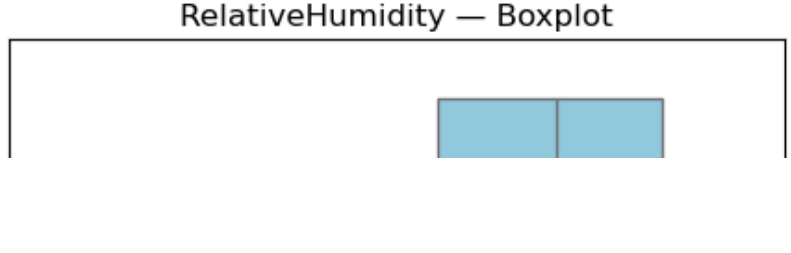

GustWindSpeed — Distribution
1.0
0.8


0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

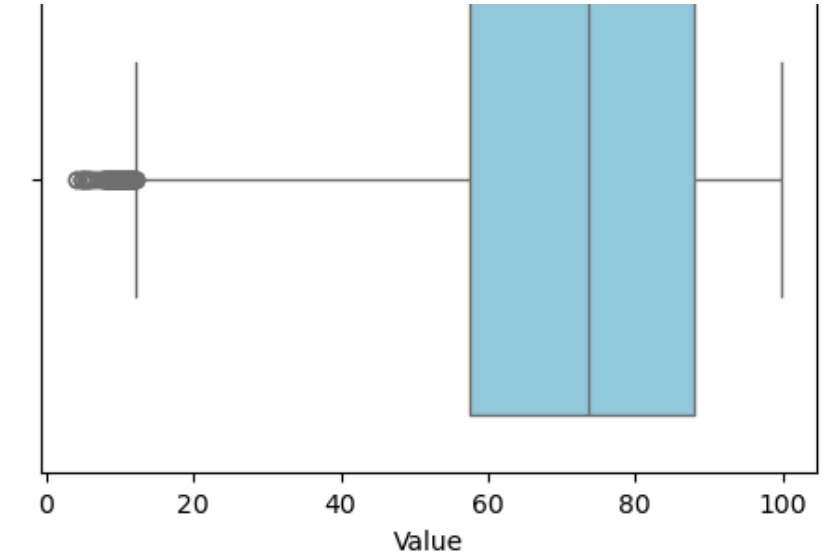

0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

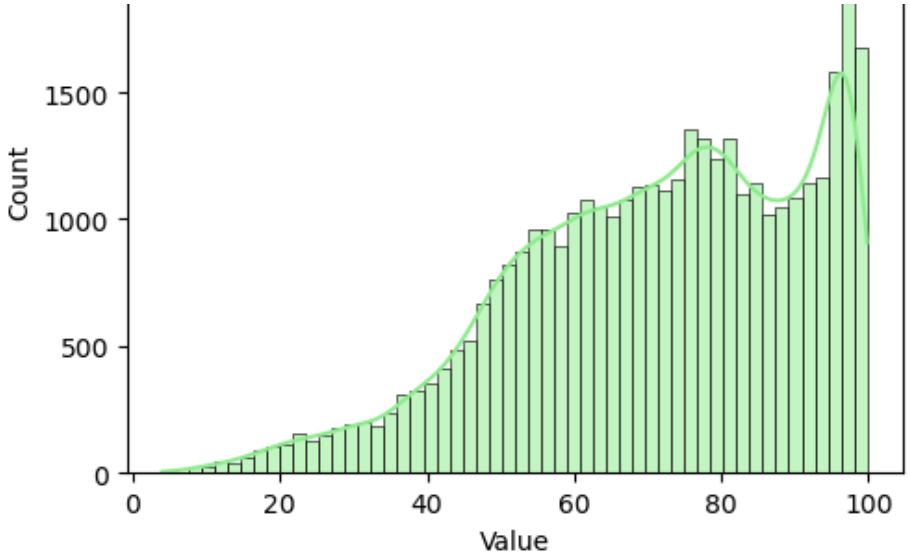

AirTemperature — Boxplot
0
5
10
15
20
25
30
35
40
Value


AirTemperature — Distribution
1600
1400
1200
1000
800
600
400
200
0
Count
0
5
10
15
20
25
30
35
40
Value

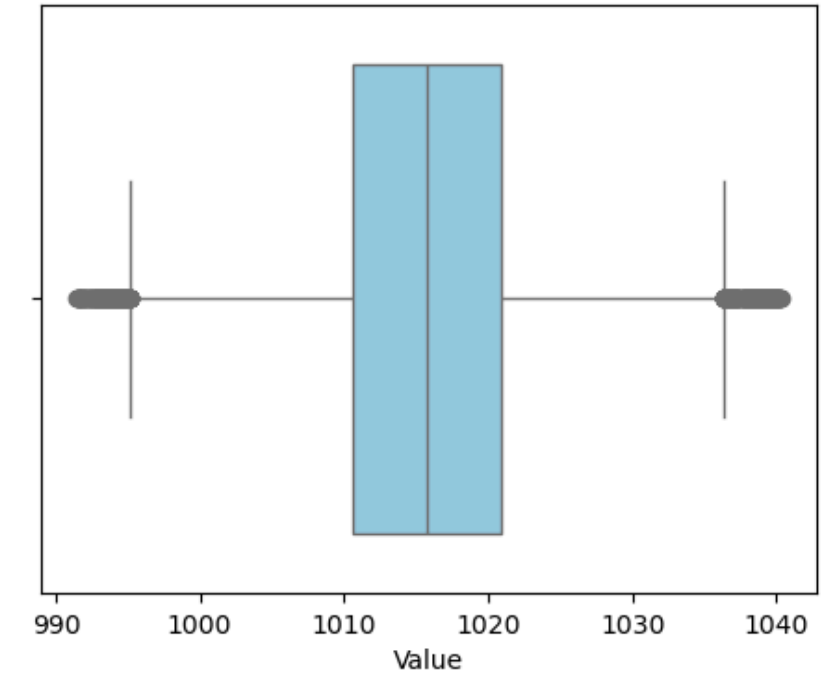

RelativeHumidity — Boxplot


RelativeHumidity — Distribution
2000

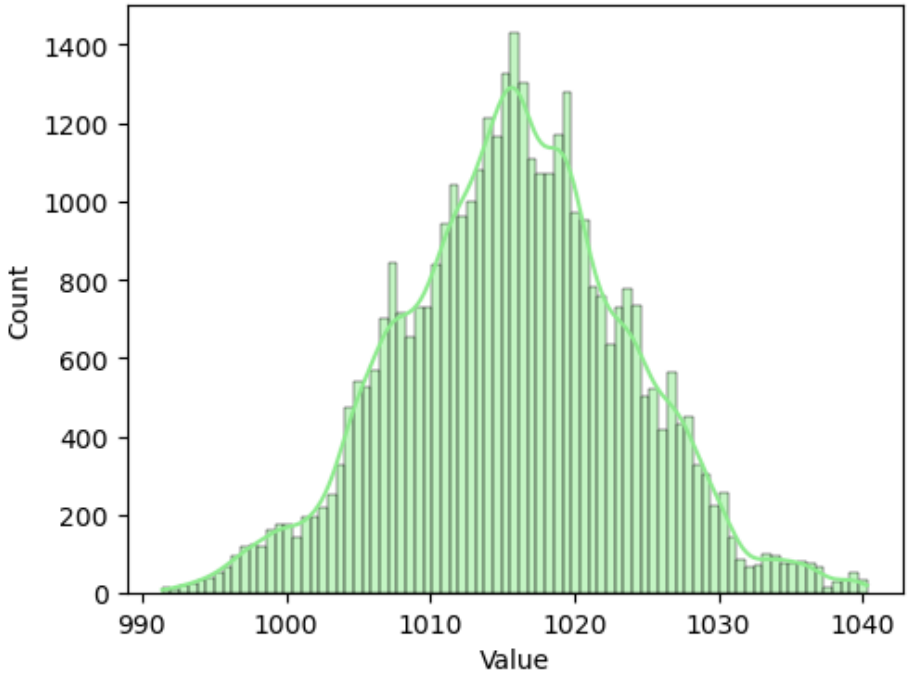




AtmosphericPressure — Boxplot

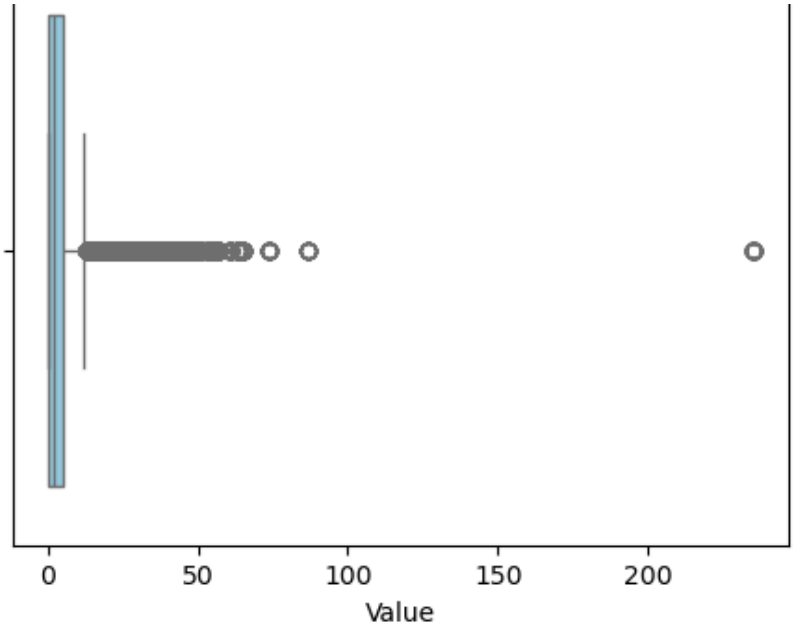


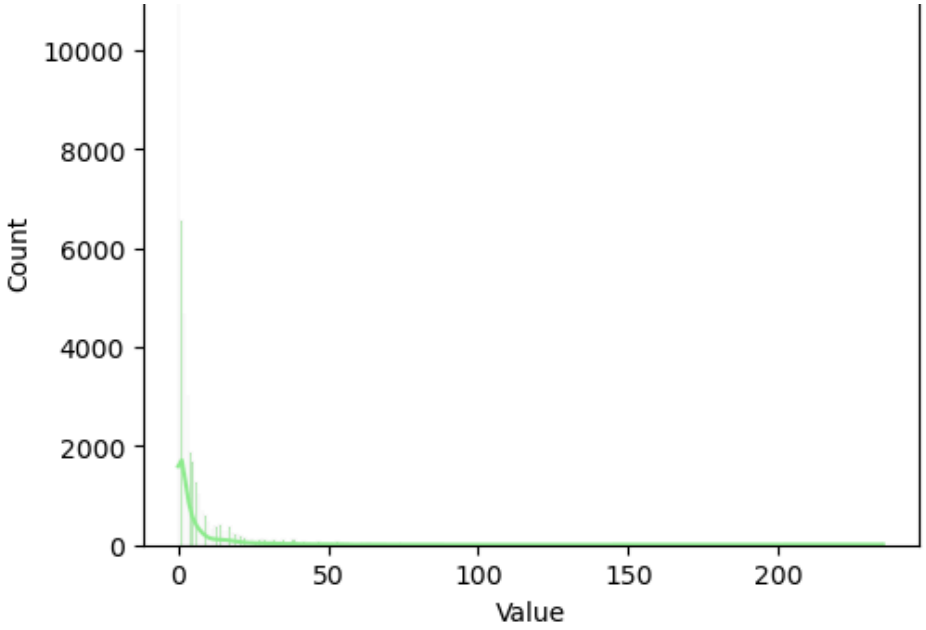
AtmosphericPressure — Distribution



PM25 — Boxplot

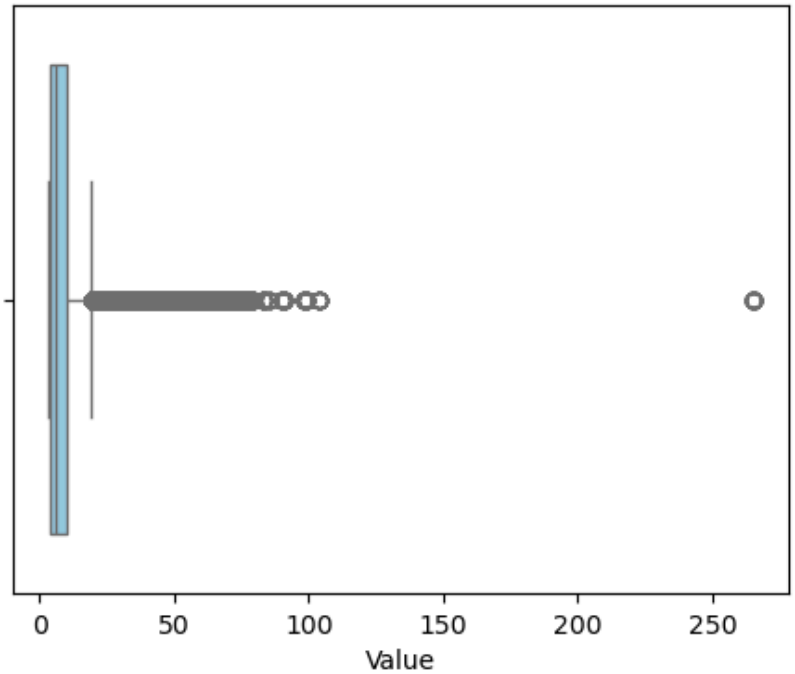
PM25 — Distribution

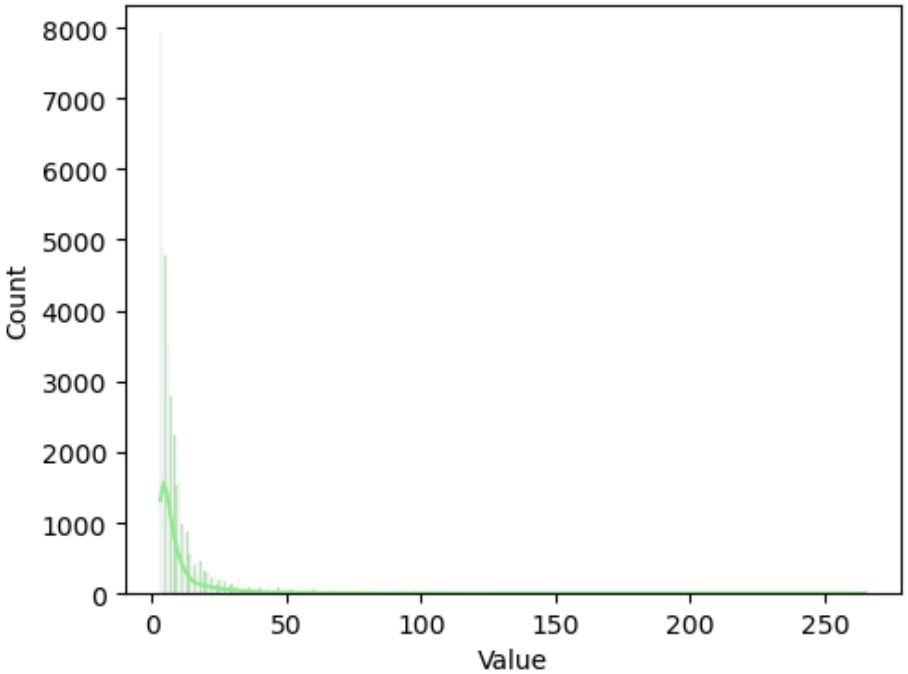




PM10 — Boxplot

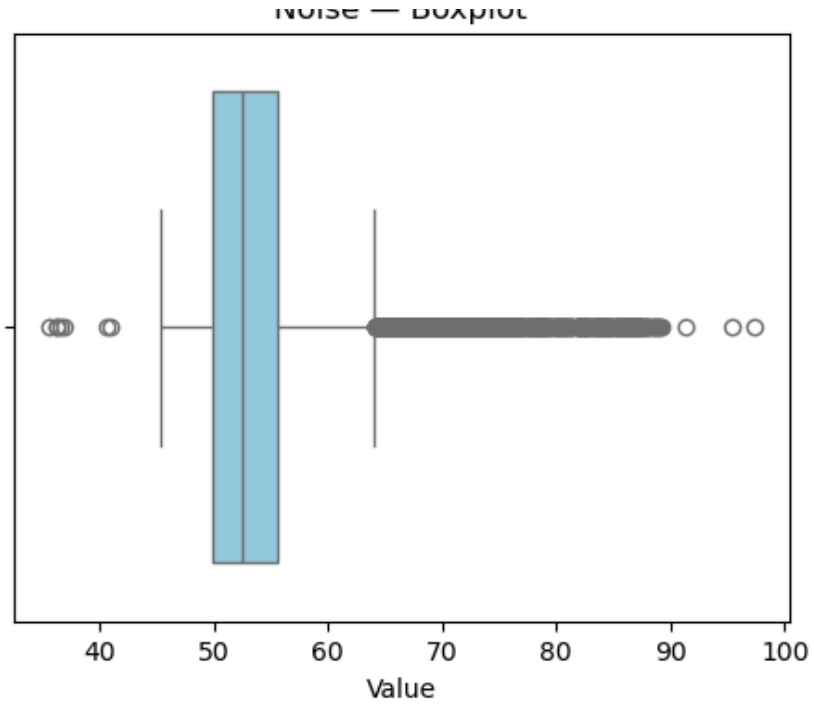
PM10 — Distribution

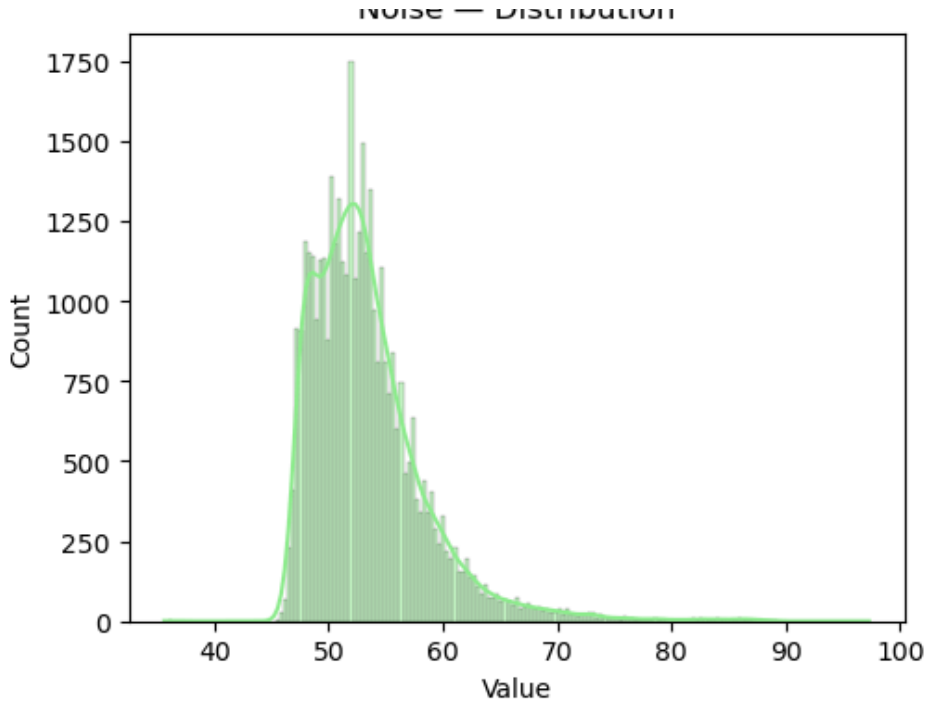




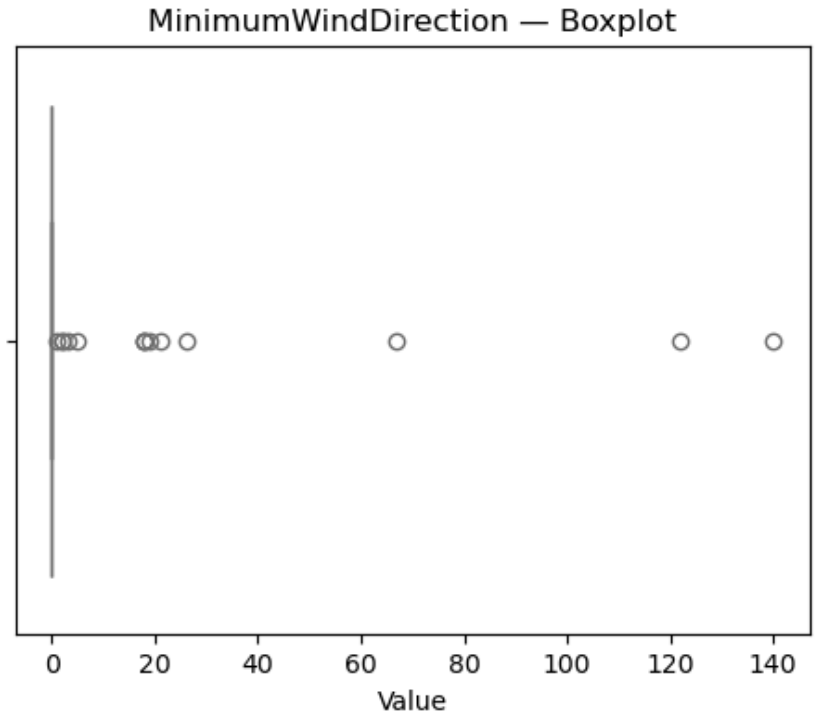
Noise — Boxplot

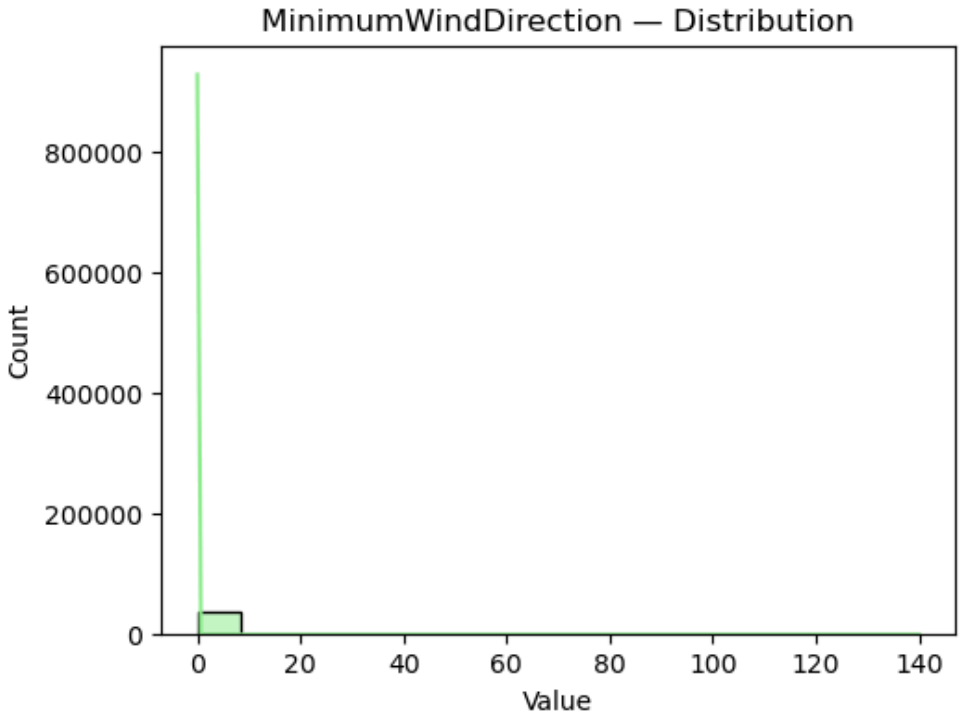
Noise — Distribution

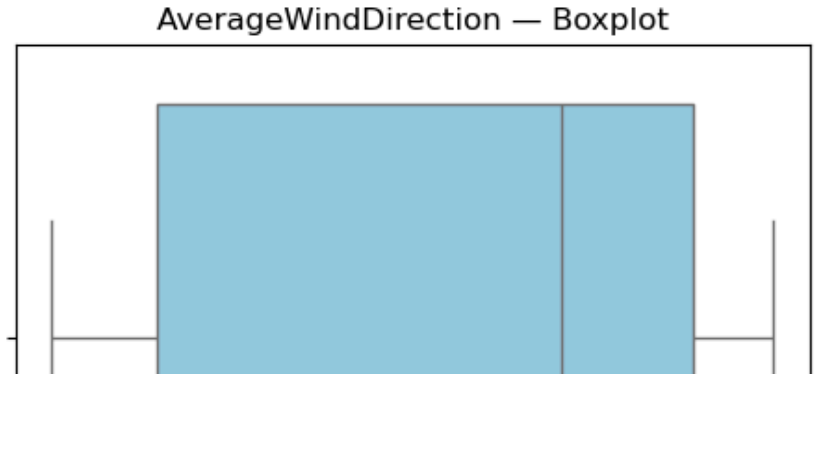

Noise — Boxplot
40
50
60
70
80
90
100
Value

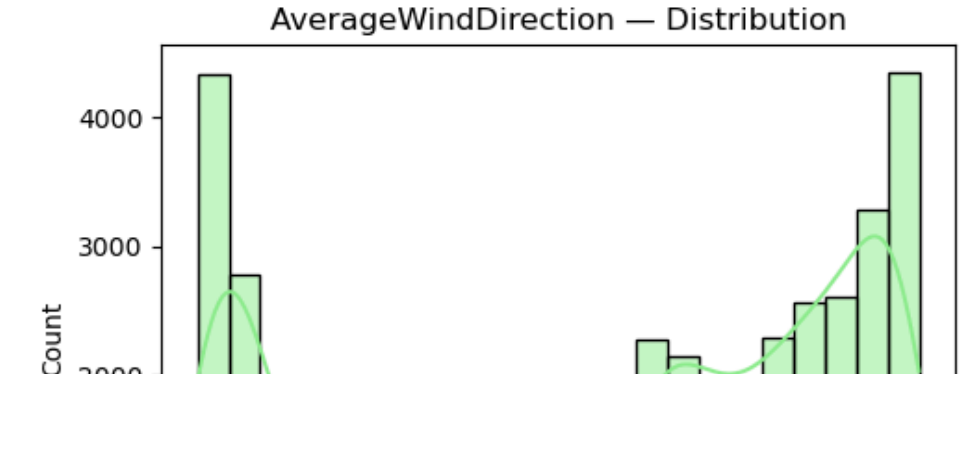

Noise — Distribution
1750
1500
1250
1000
750
500
250
0
Count
40
50
60
70
80
90
100
Value

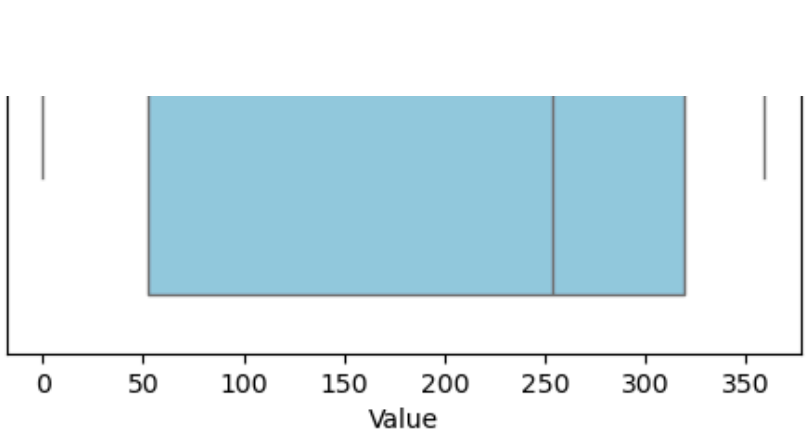
Device: ICTMicroclimate-09

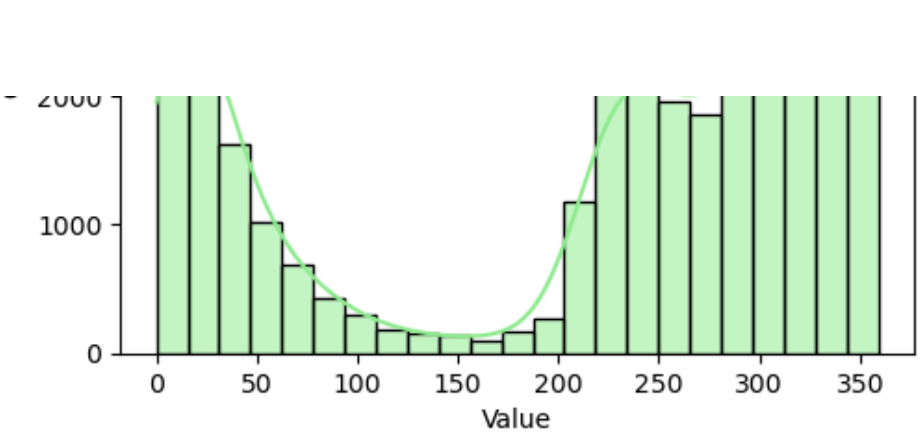




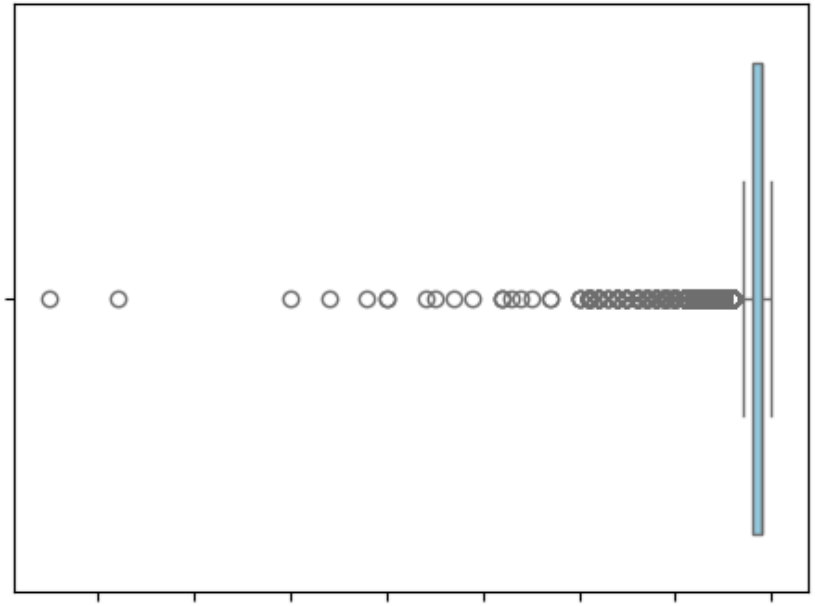




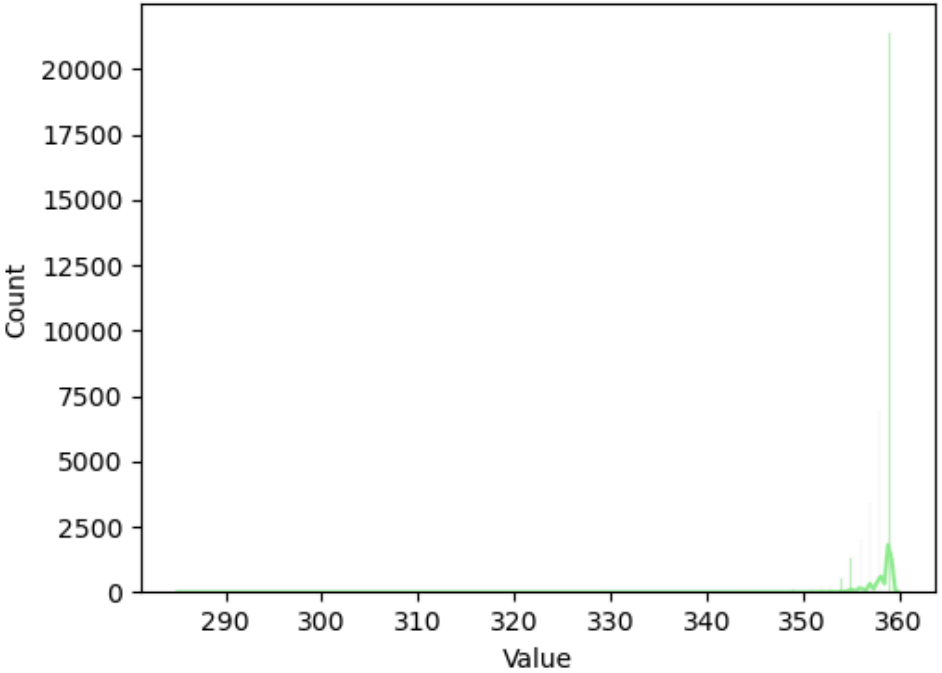




MaximumWindDirection — Boxplot

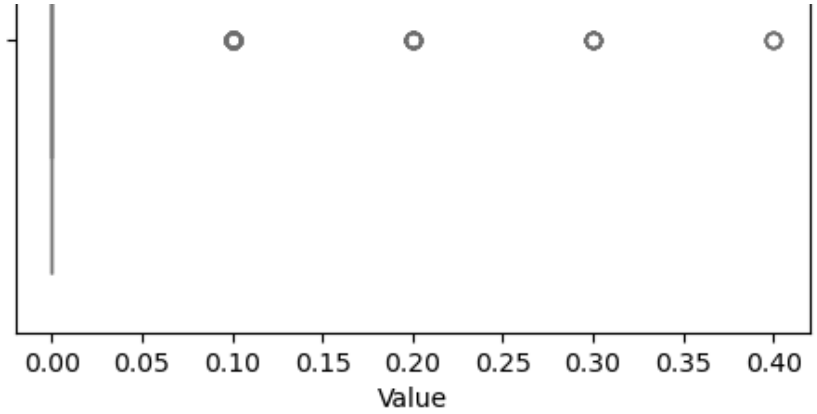


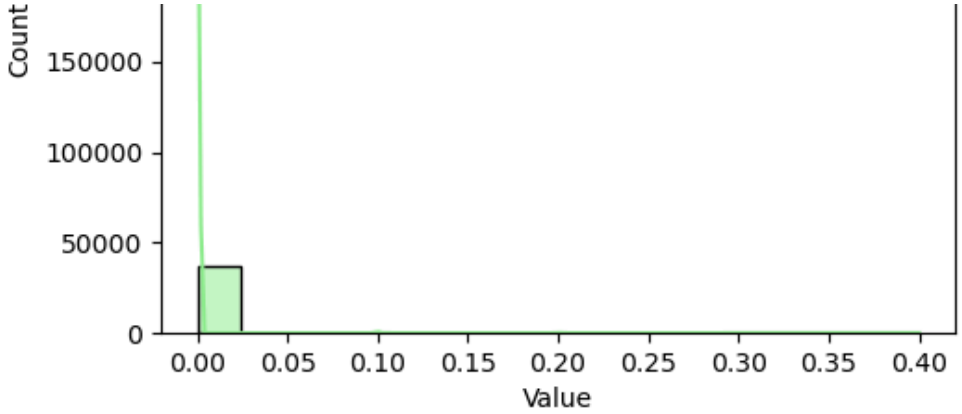
MaximumWindDirection — Distribution



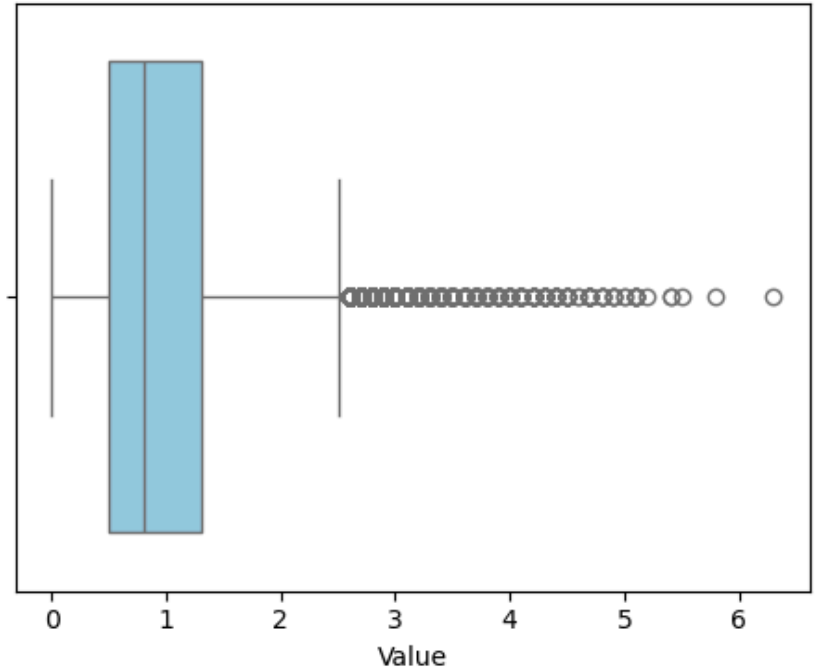
MinimumWindSpeed — Boxplot

MinimumWindSpeed — Distribution

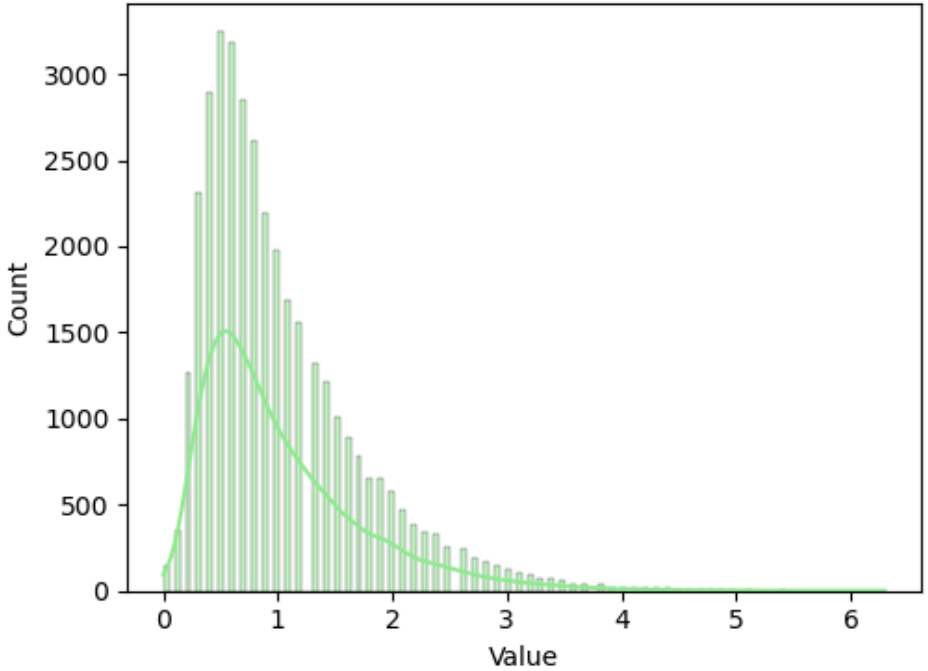




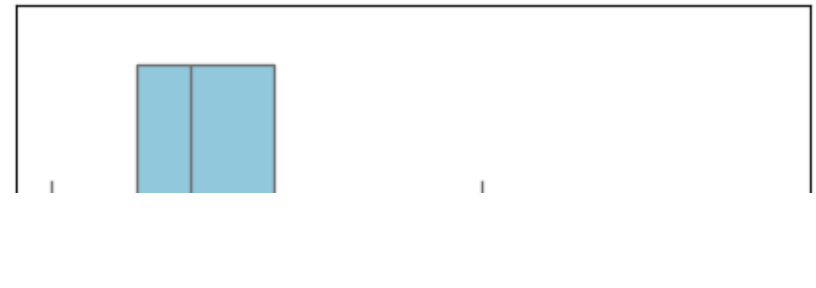
AverageWindSpeed — Boxplot



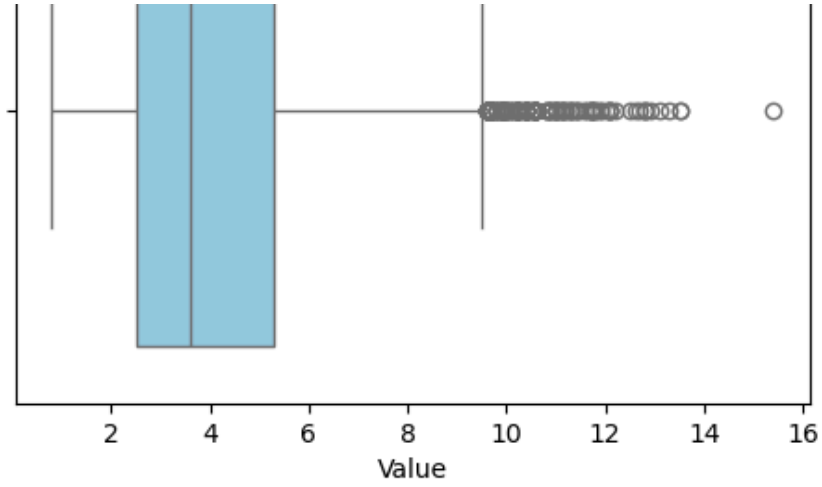
AverageWindSpeed — Distribution

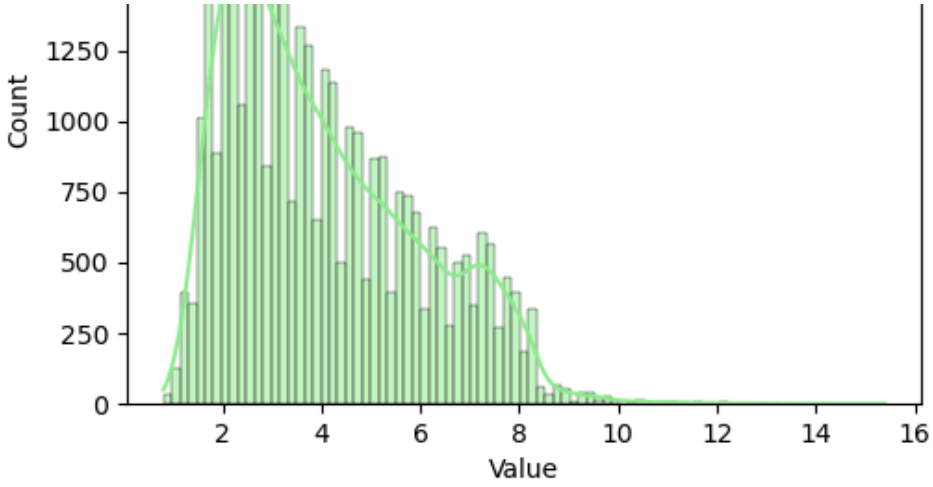


GustWindSpeed — Boxplot

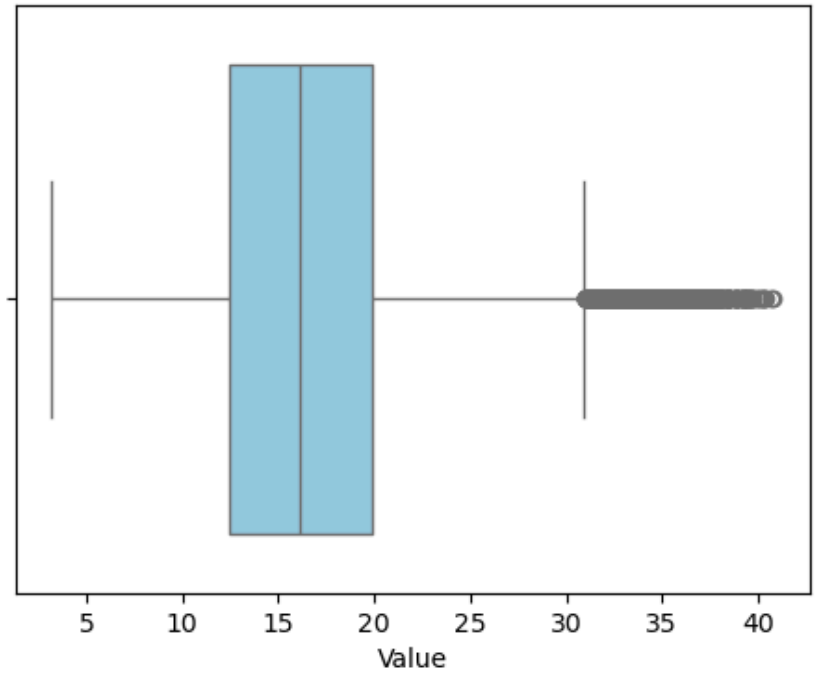


GustWindSpeed — Distribution

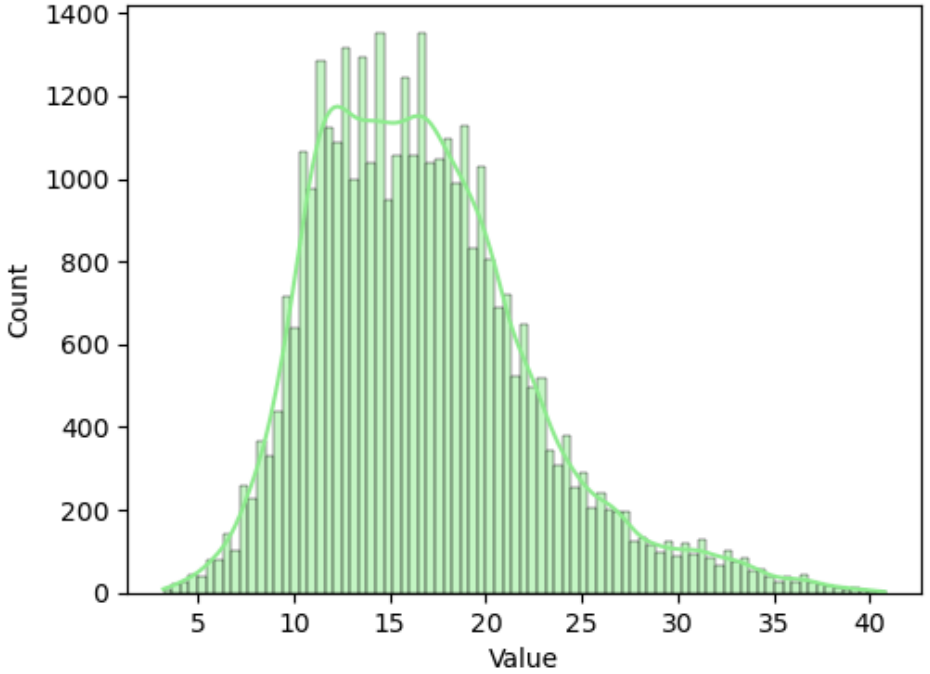




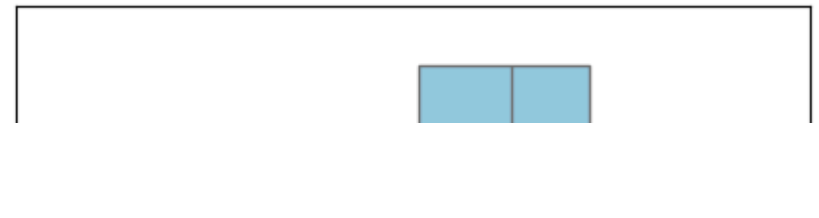
AirTemperature — Boxplot



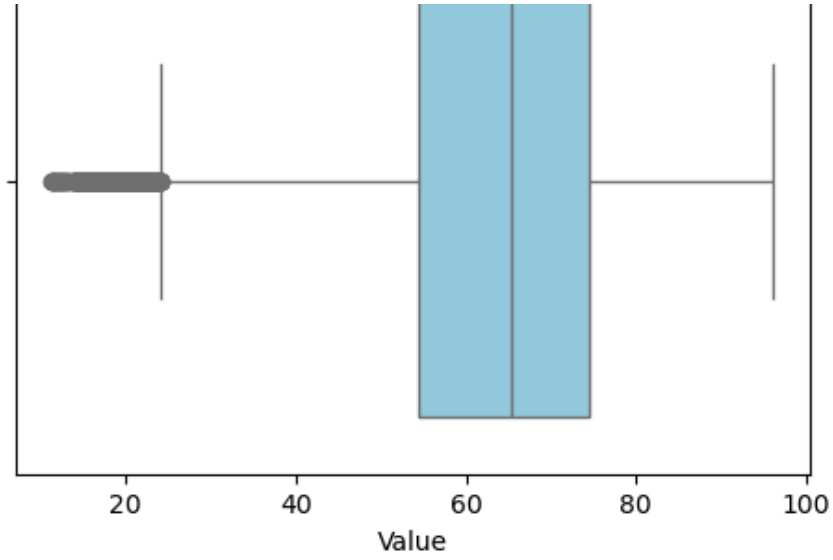
AirTemperature — Distribution

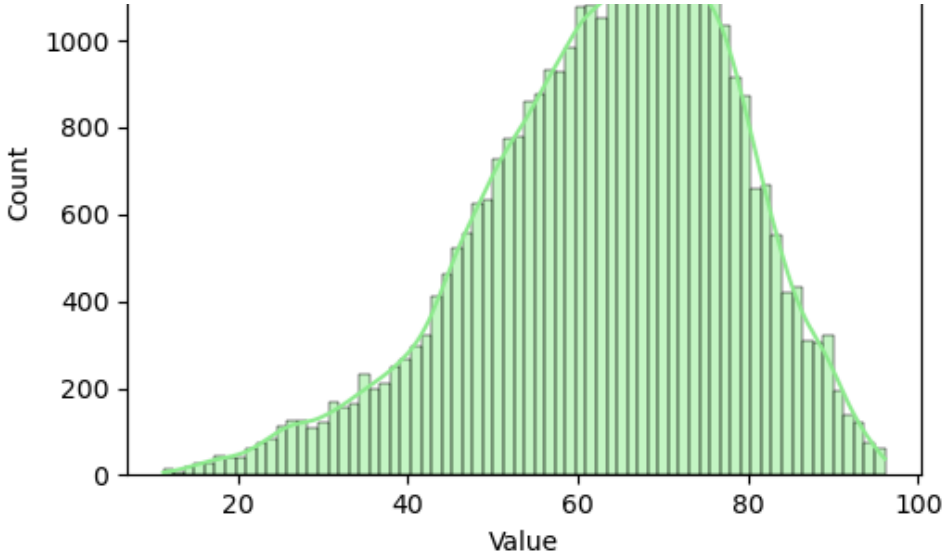


RelativeHumidity — Boxplot

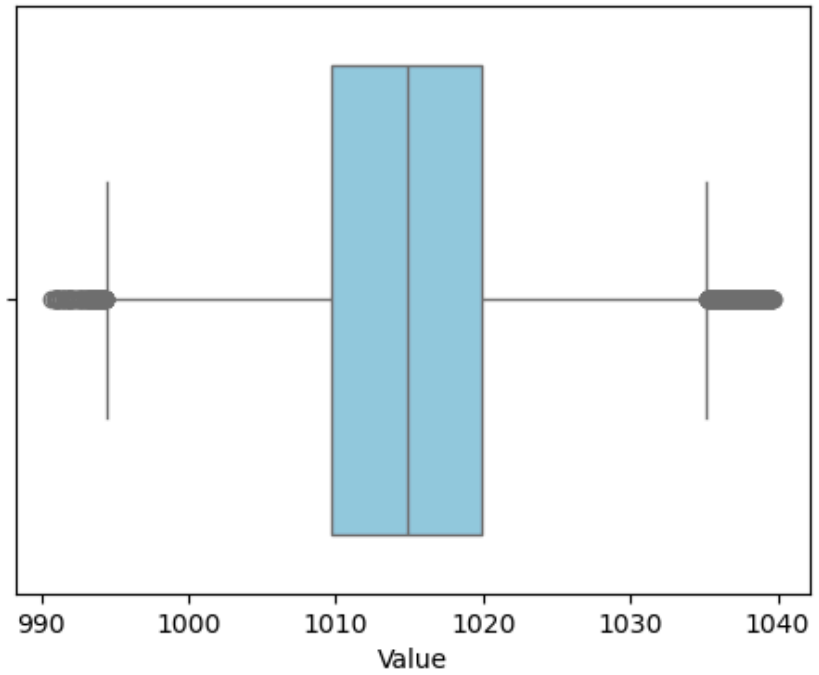


RelativeHumidity — Distribution

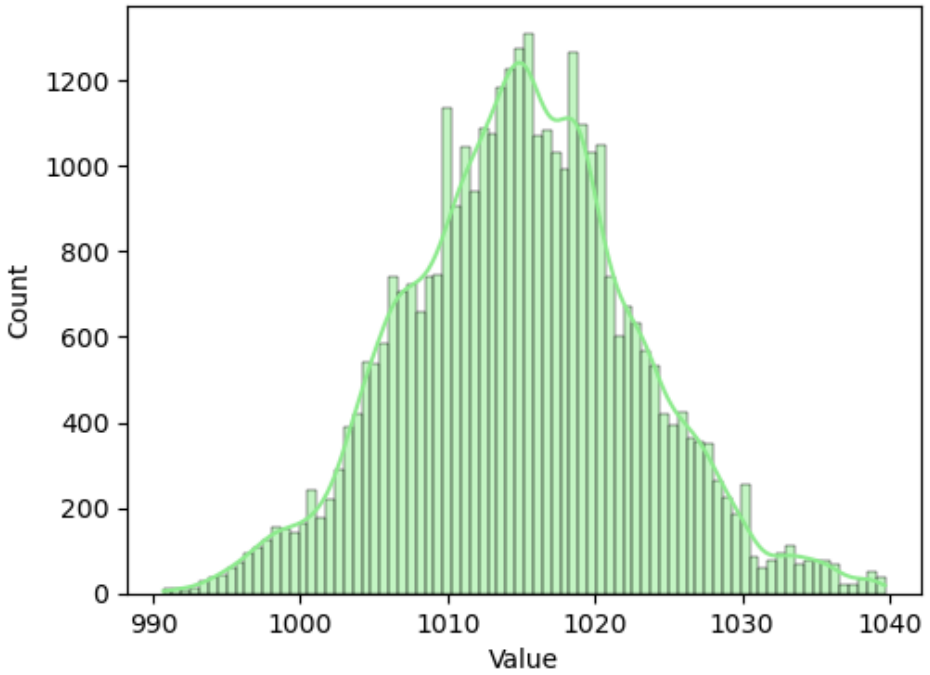




AtmosphericPressure — Boxplot

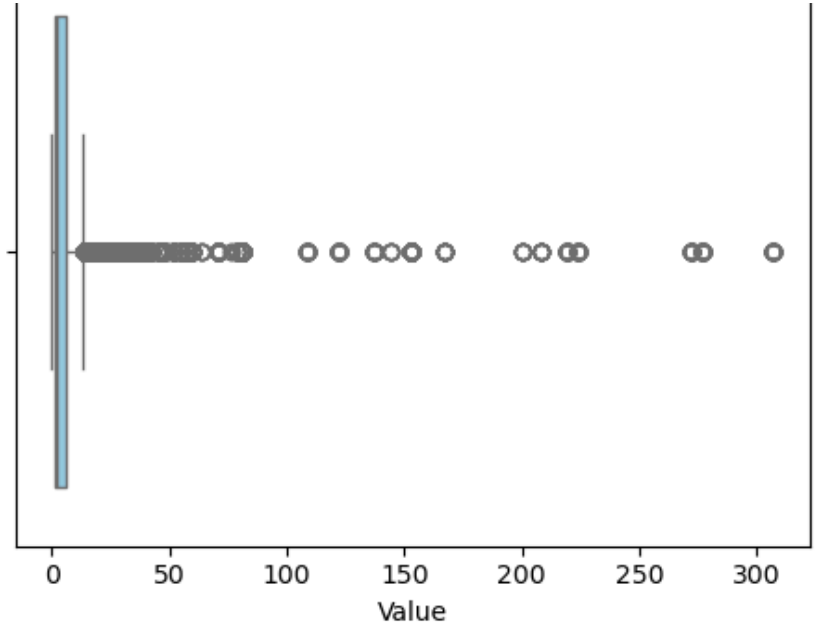


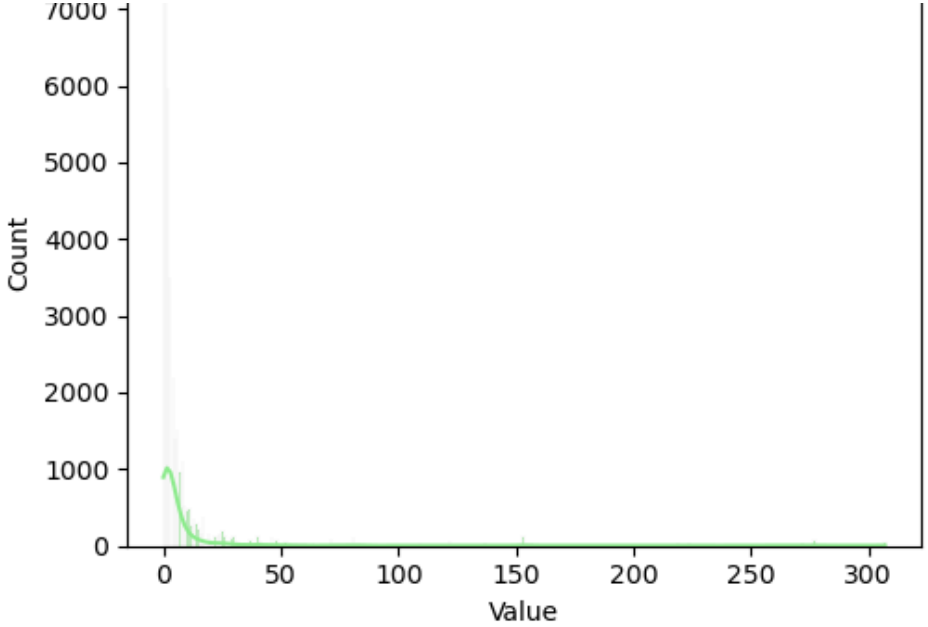
AtmosphericPressure — Distribution



PM25 — Boxplot

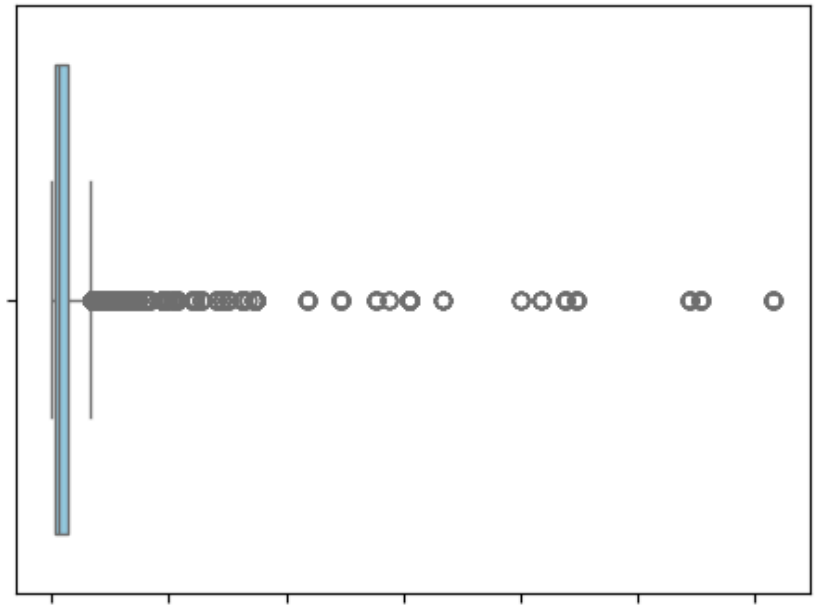
PM25 — Distribution

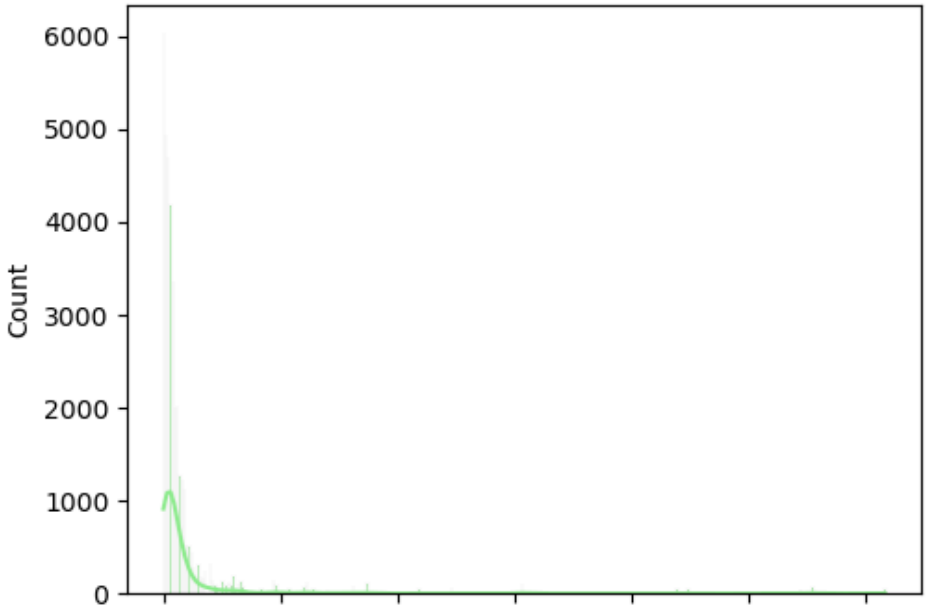




PM10 — Boxplot

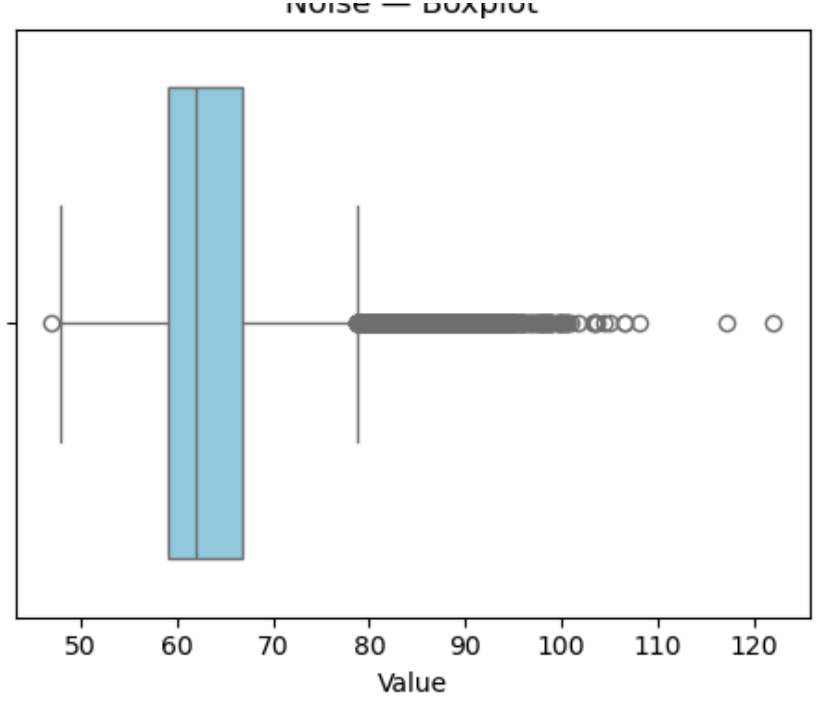
PM10 — Distribution

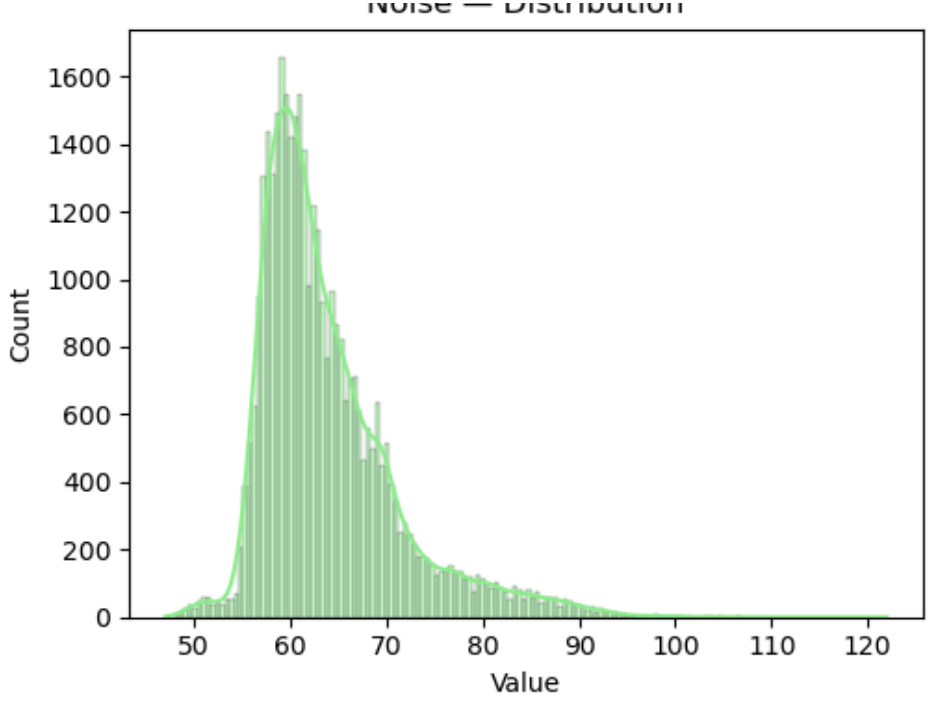




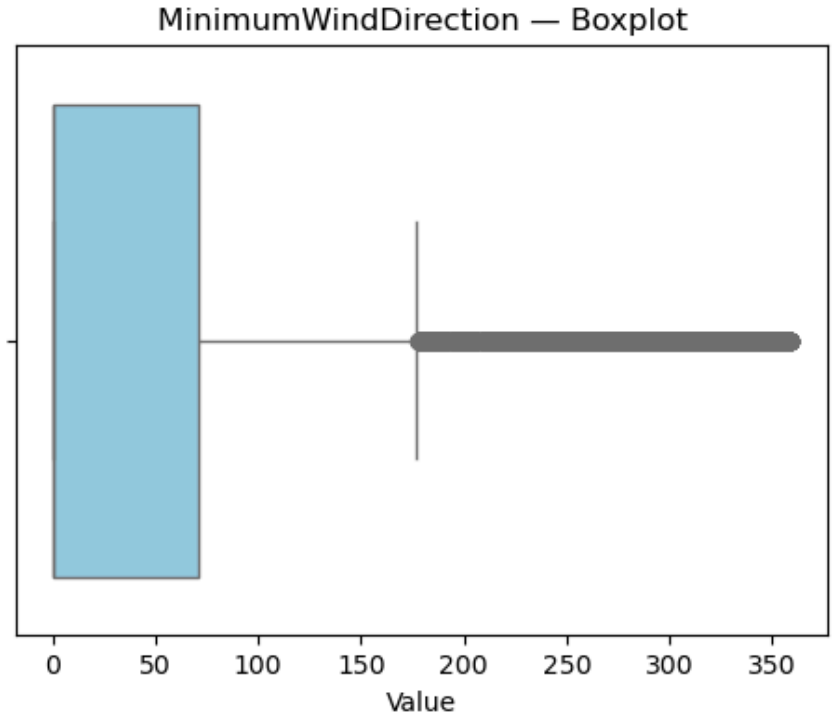
Noise — Boxplot

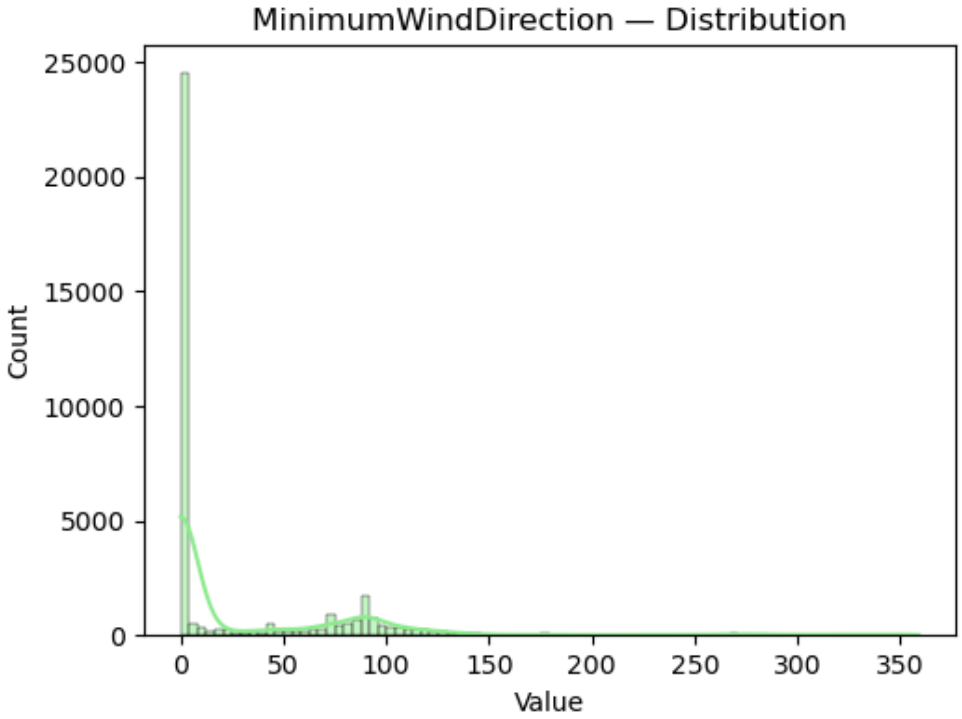
Noise — Distribution

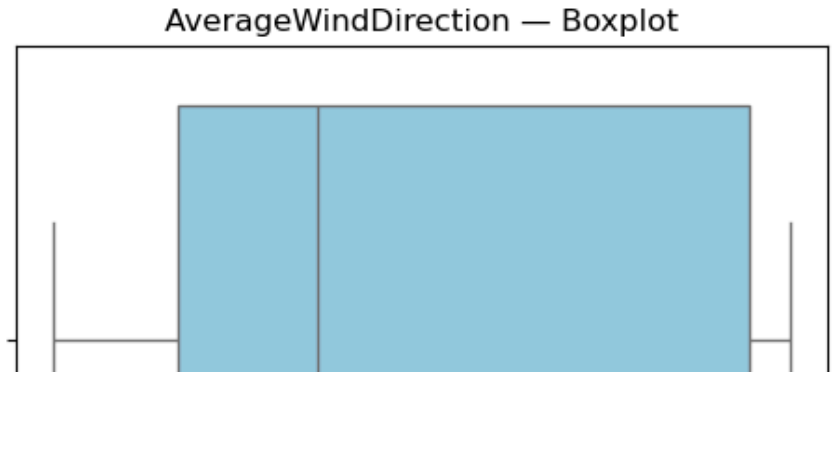

Noise — Boxplot
50
60
70
80
90
100
110
120
Value

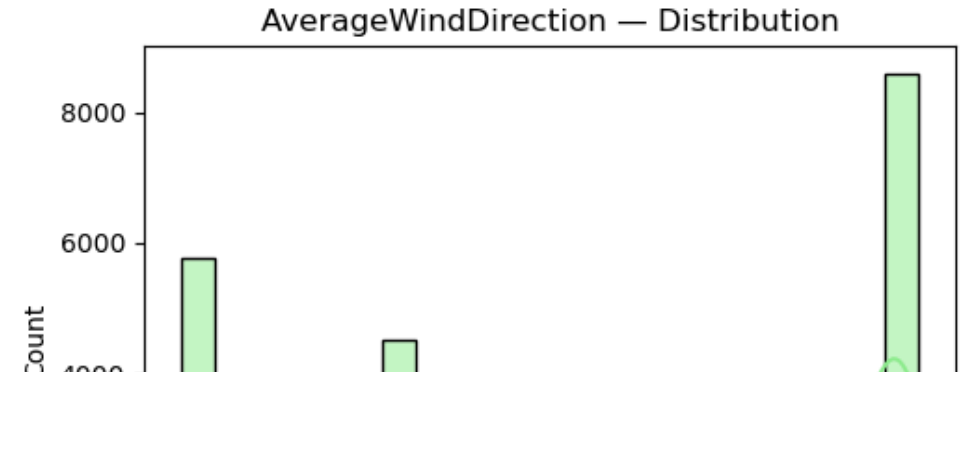

Noise — Distribution
1600
1400
1200
1000
800
600
400
200
0
Count
50
60
70
80
90
100
110
120
Value

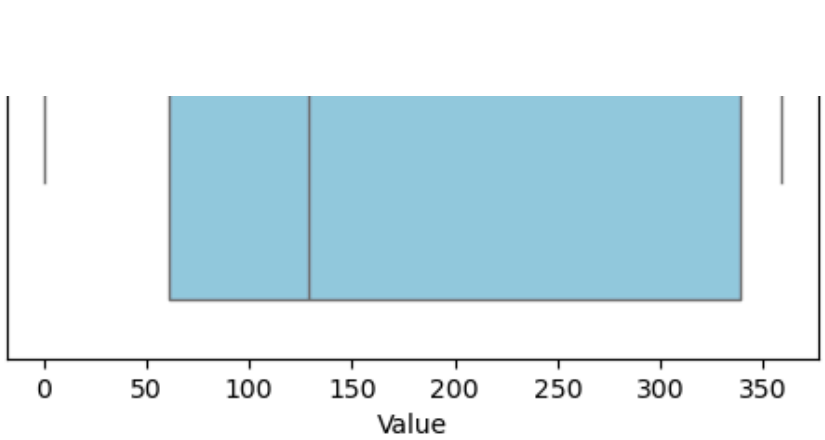
Device: ICTMicroclimate-02

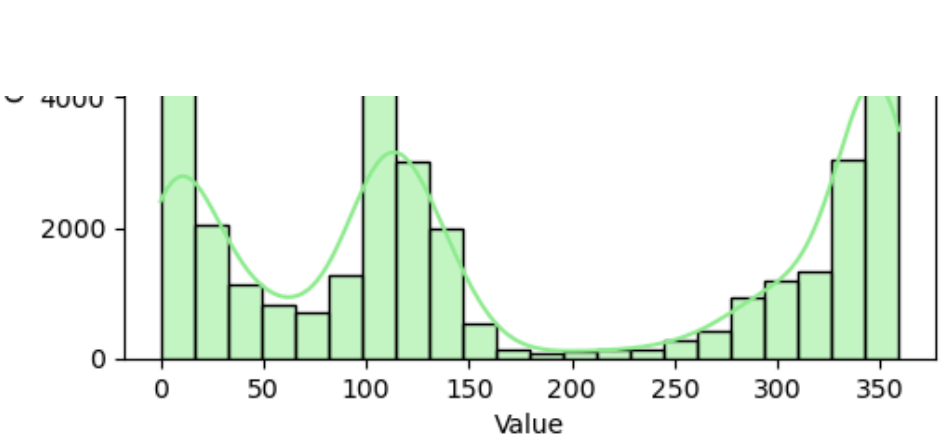




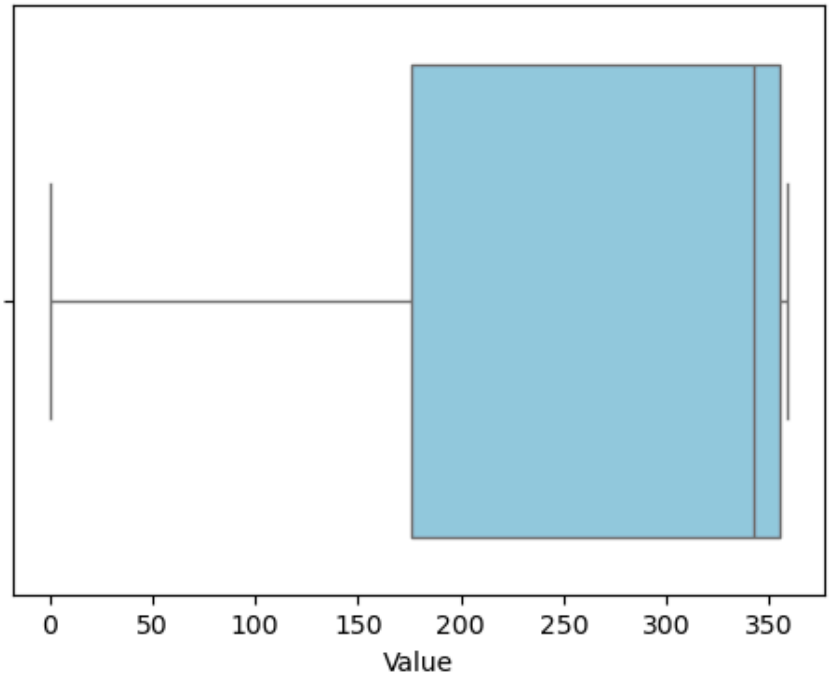




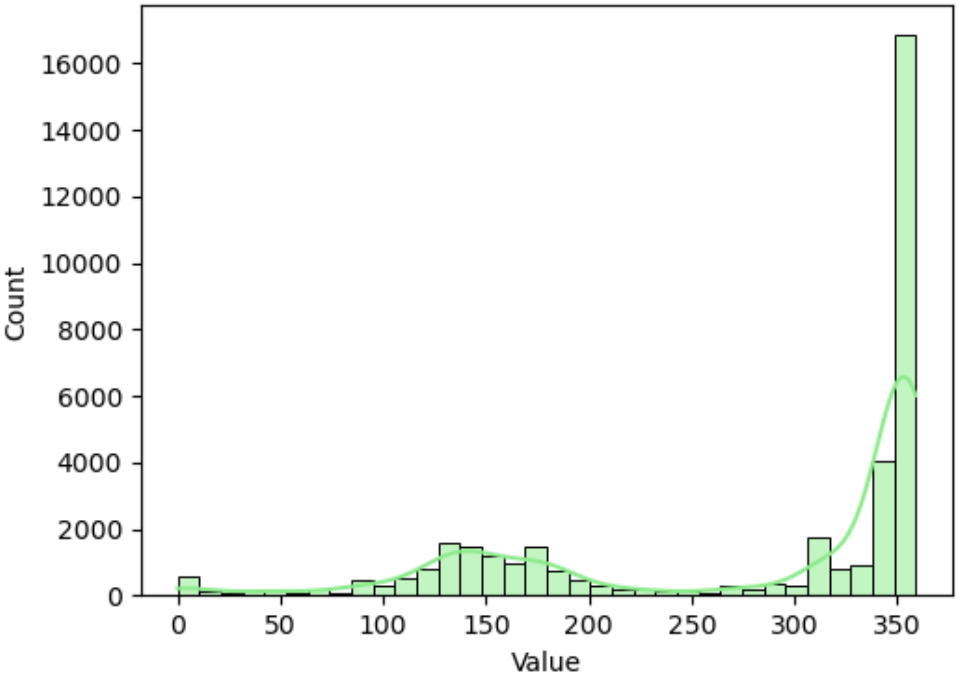




MaximumWindDirection — Boxplot

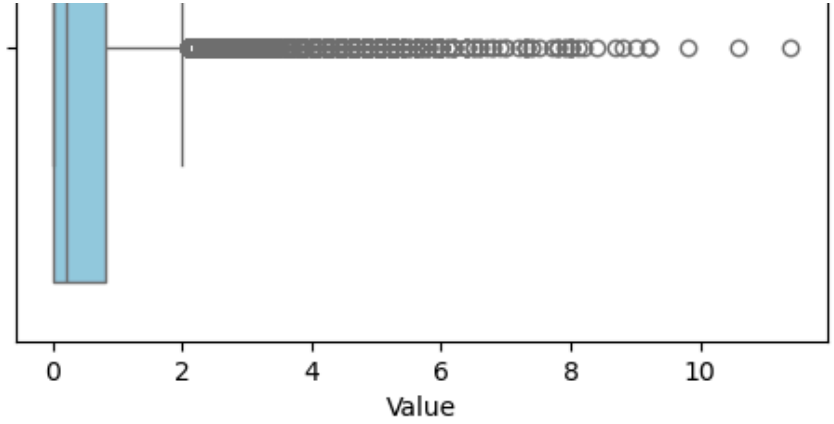


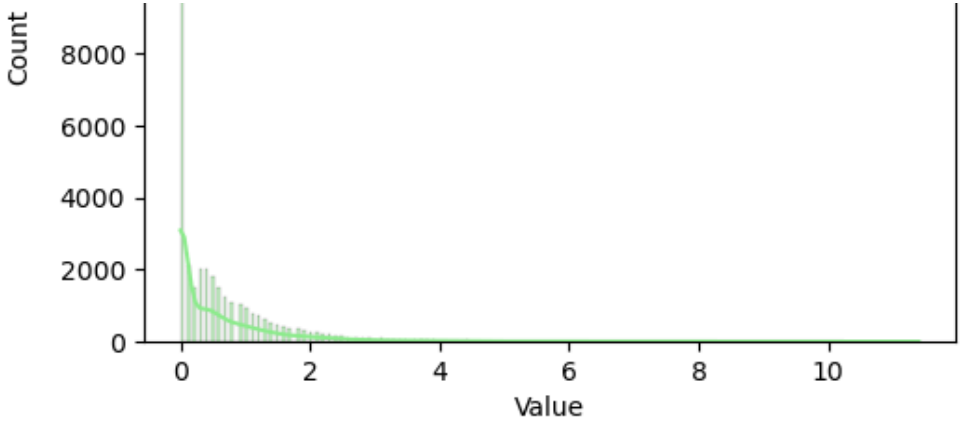
MaximumWindDirection — Distribution



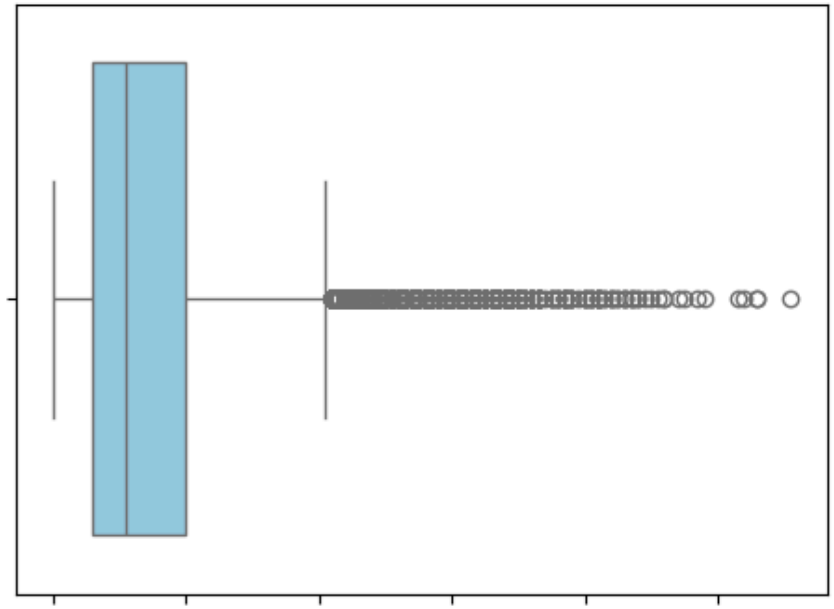
MinimumWindSpeed — Boxplot

MinimumWindSpeed — Distribution

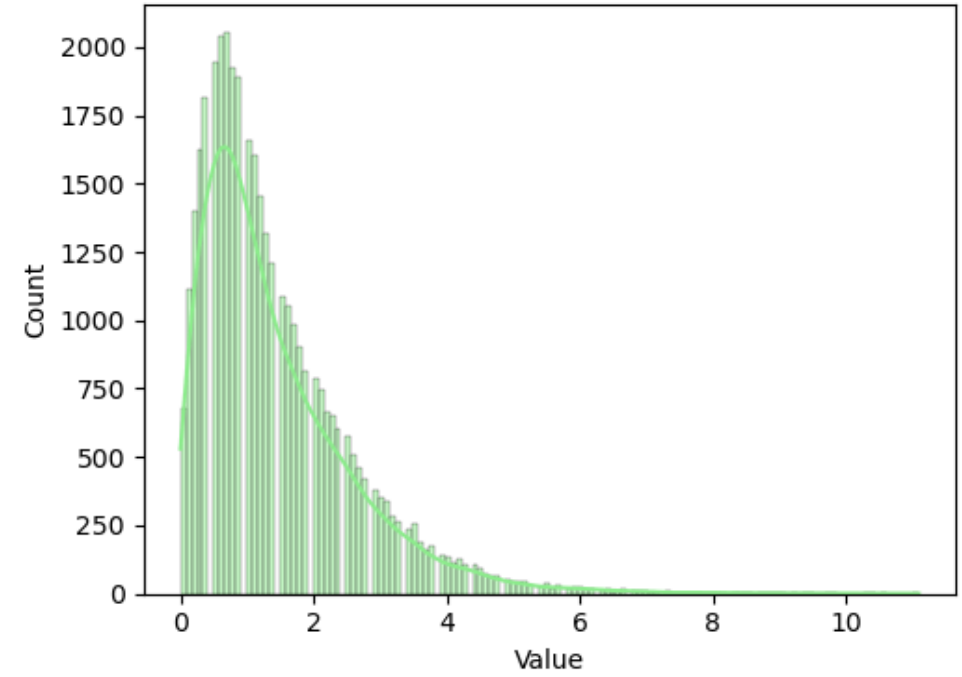




AverageWindSpeed — Boxplot

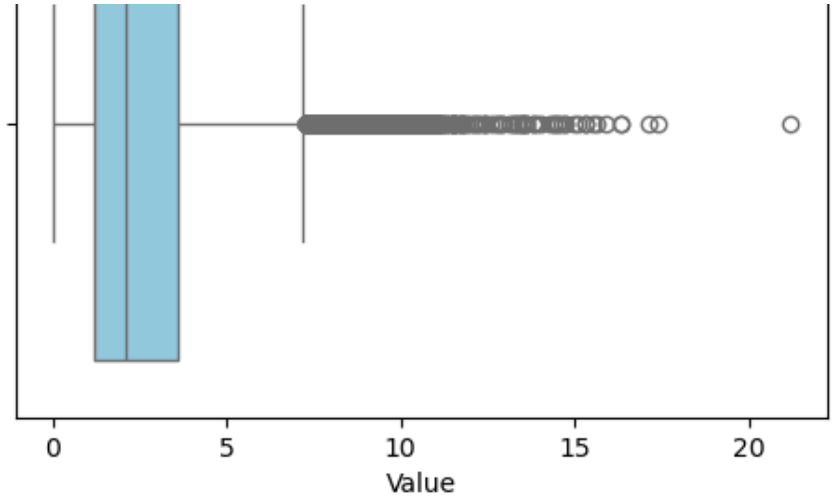


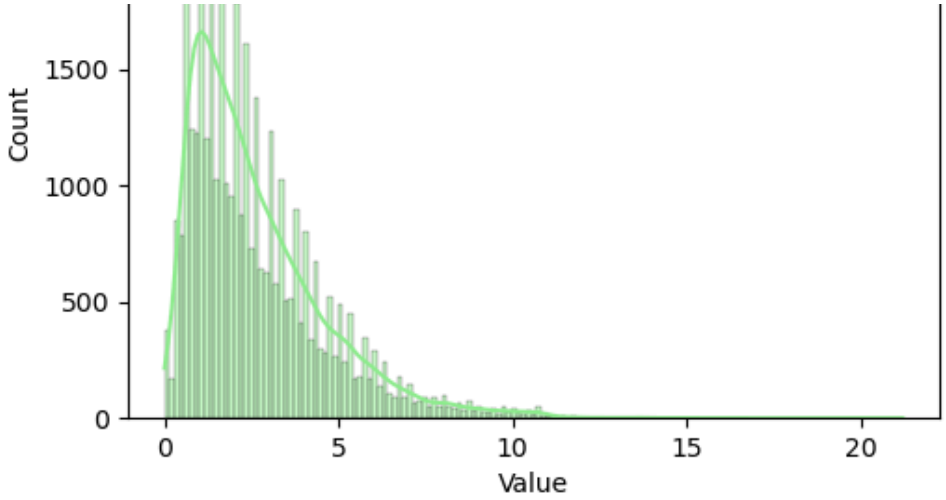
AverageWindSpeed — Distribution



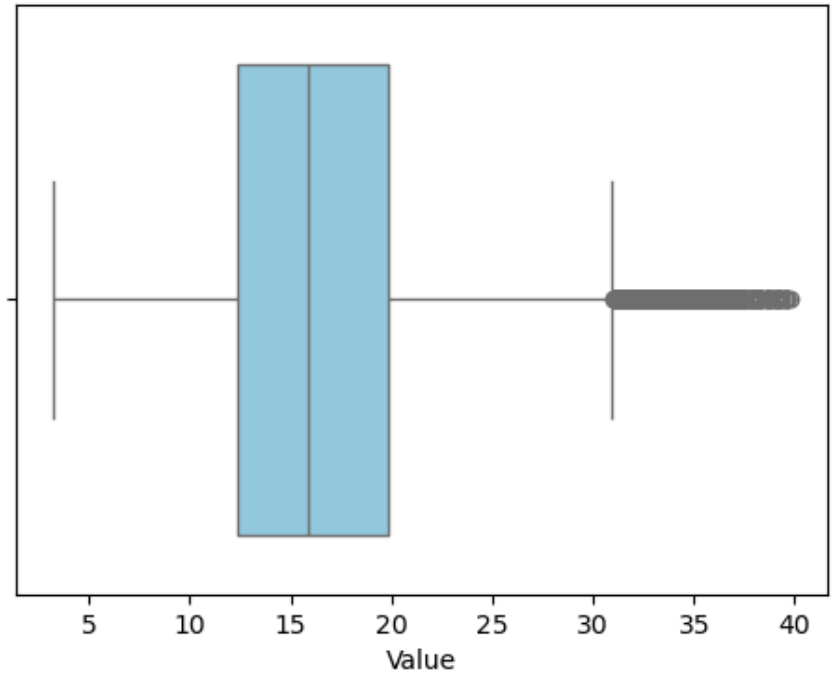
GustWindSpeed — Boxplot

GustWindSpeed — Distribution

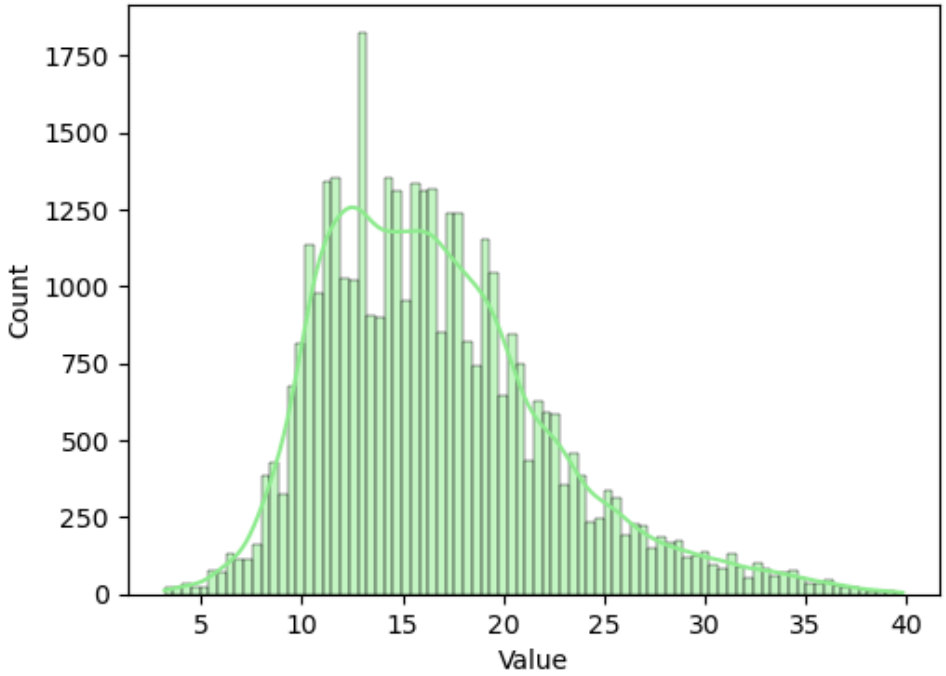




AirTemperature — Boxplot

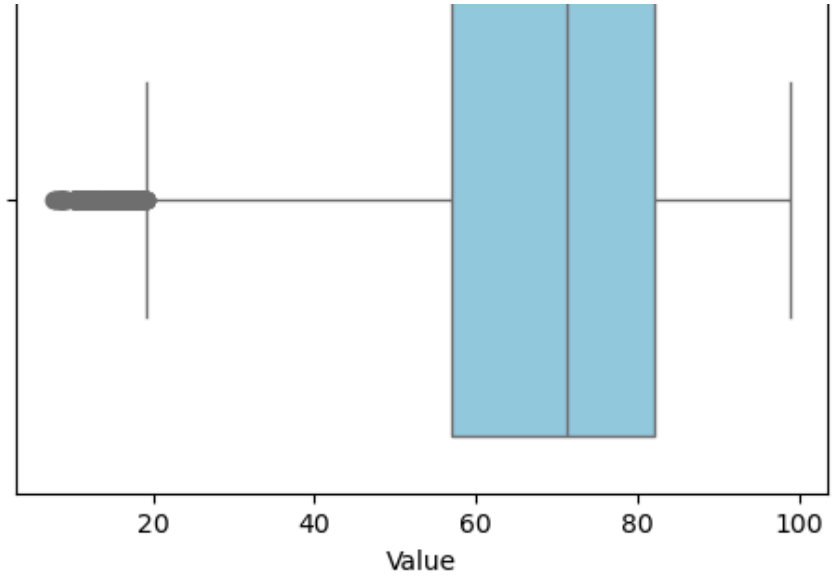


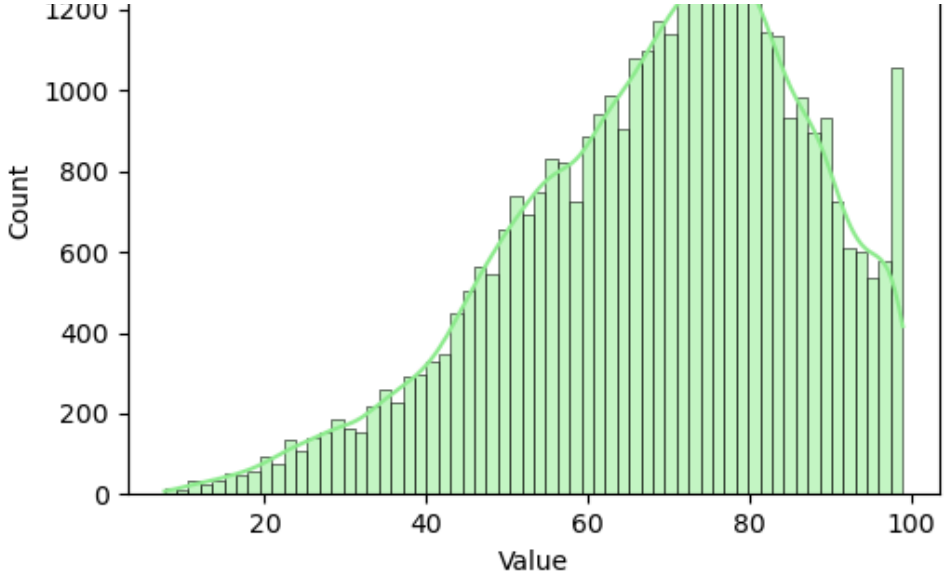
AirTemperature — Distribution



RelativeHumidity — Boxplot

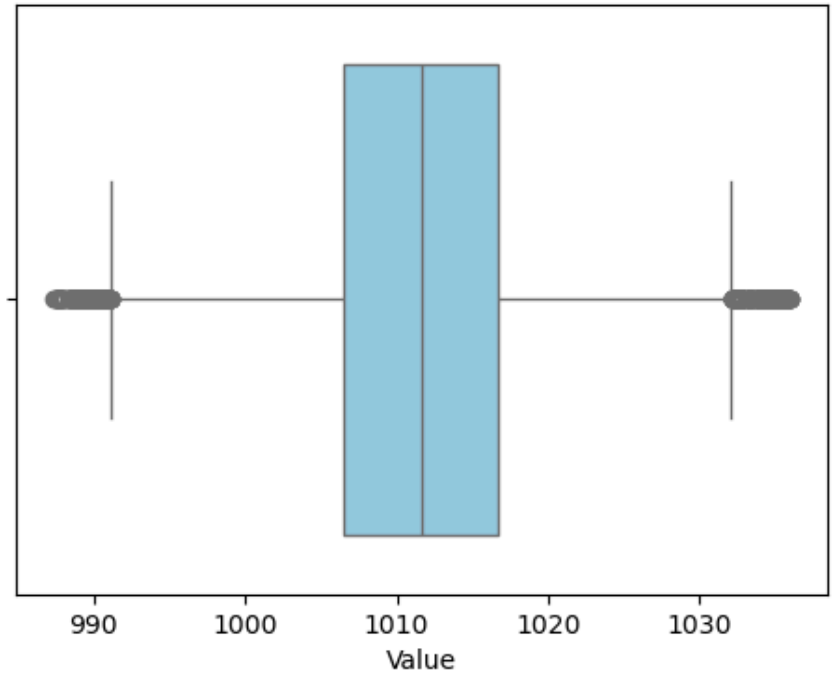
RelativeHumidity — Distribution

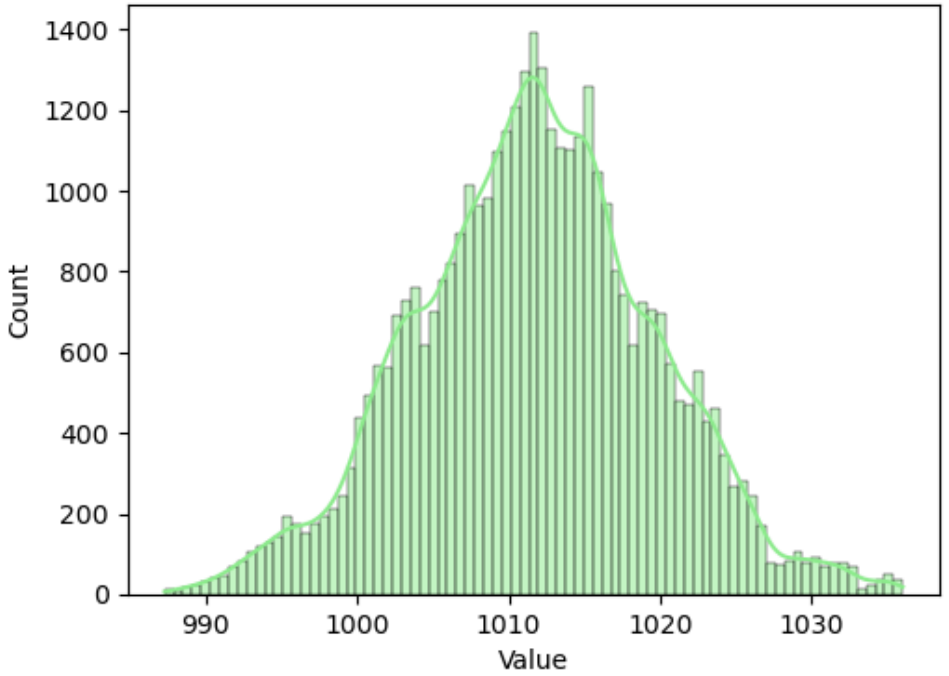




AtmosphericPressure — Boxplot

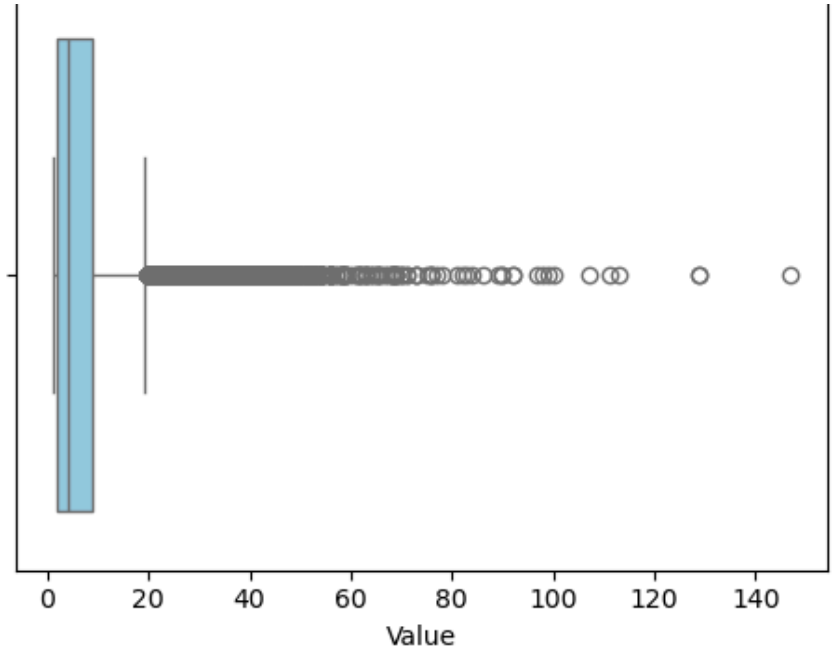
AtmosphericPressure — Distribution

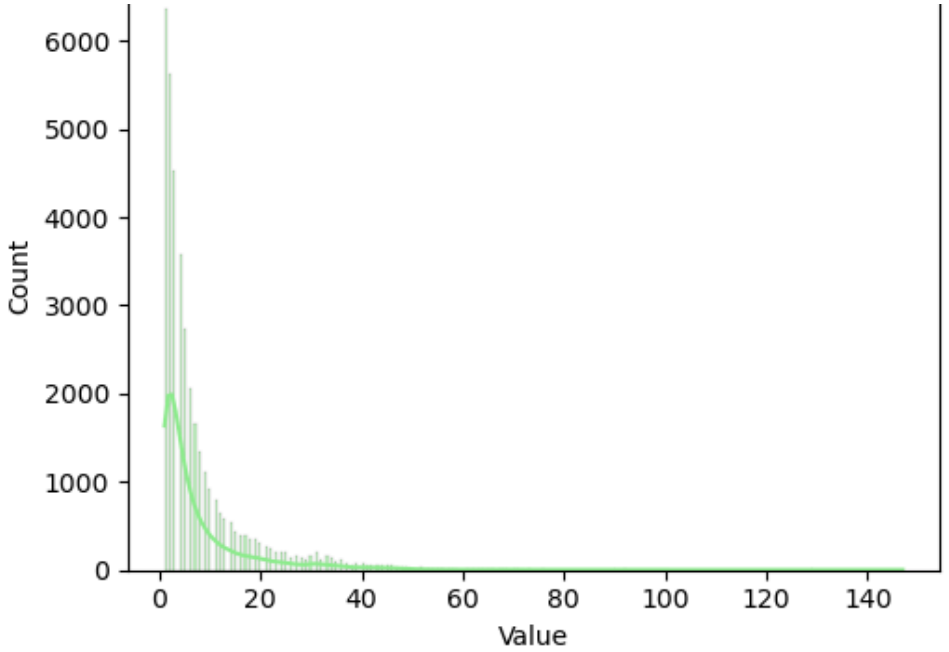




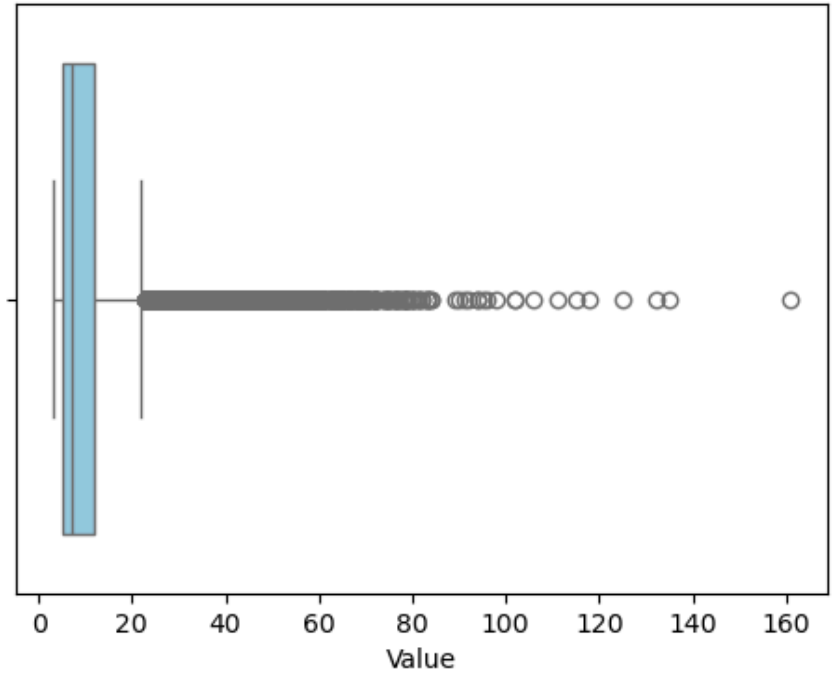
PM25 — Boxplot

PM25 — Distribution

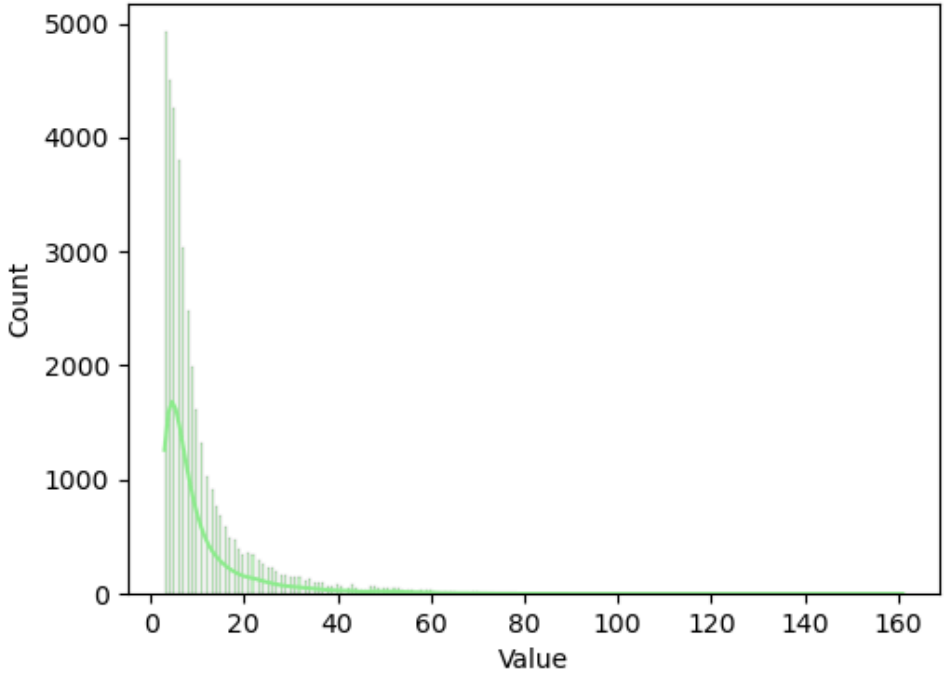


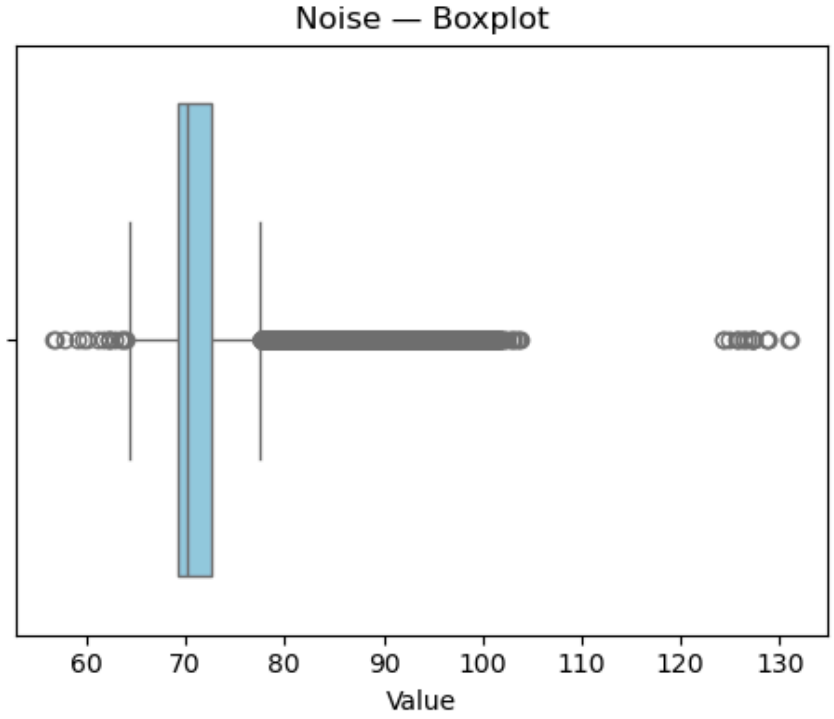


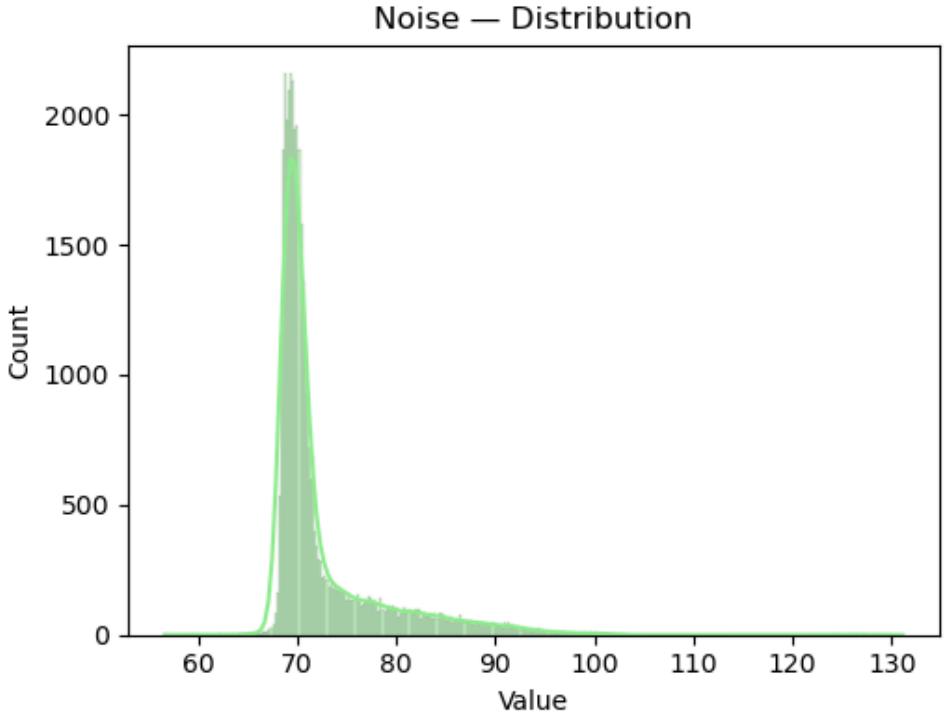
PM10 — Boxplot



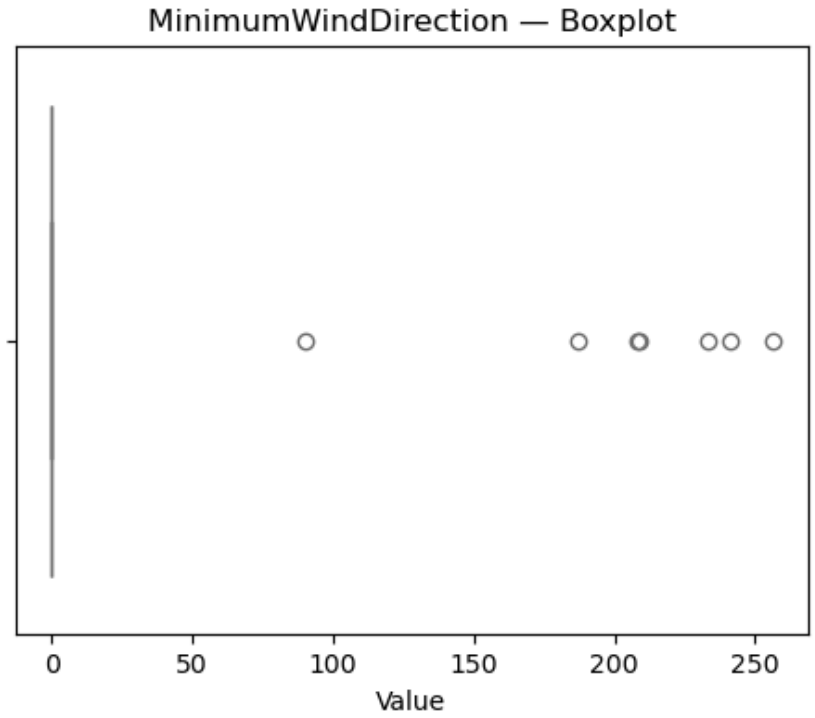
PM10 — Distribution

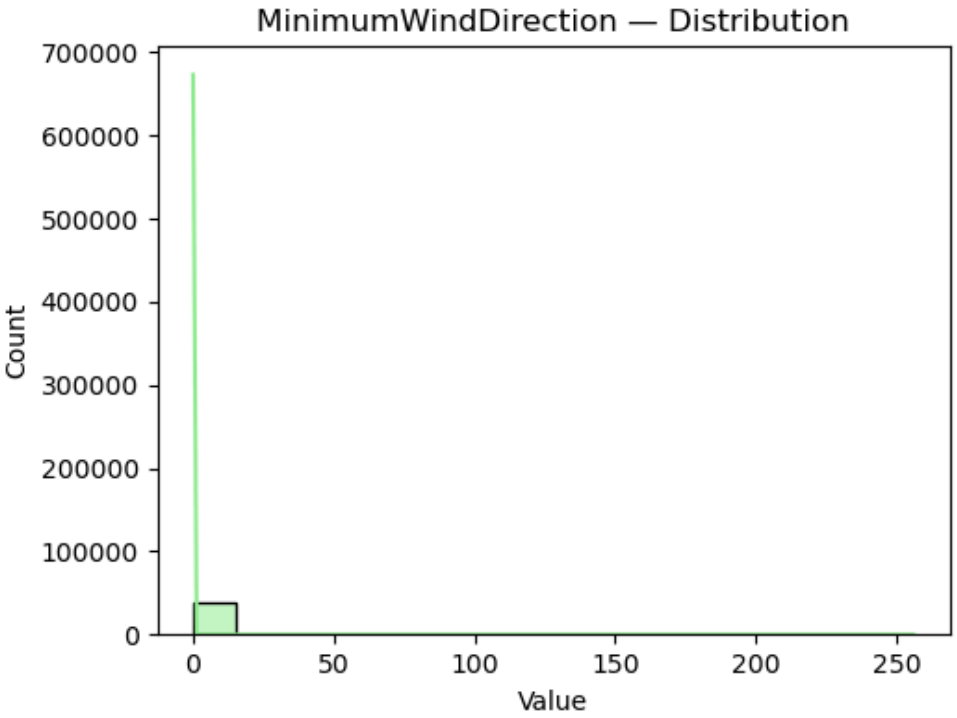


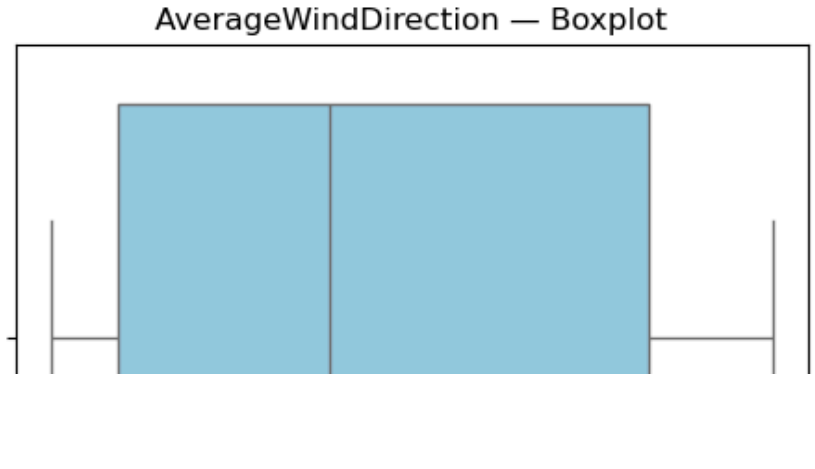

Noise — Boxplot
60
70
80
90
100
110
120
130
Value

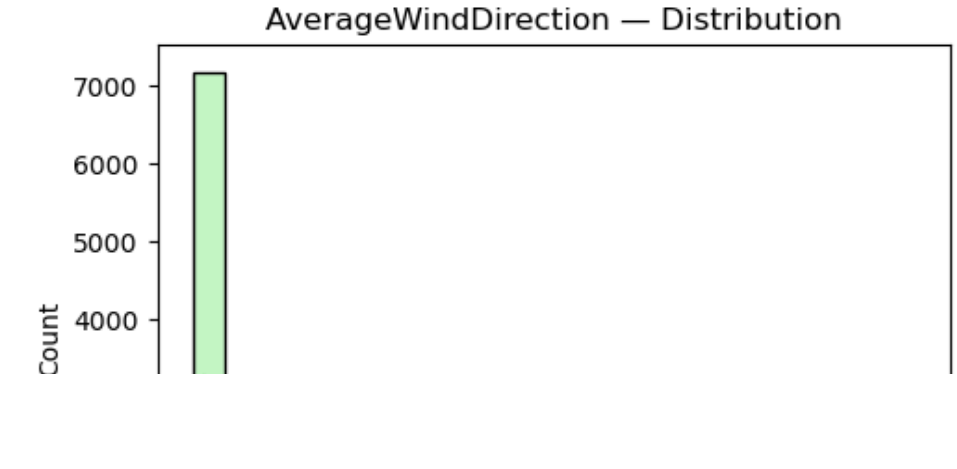

Noise — Distribution
2000
1500
1000
500
0
Count
60
70
80
90
100
110
120
130
Value

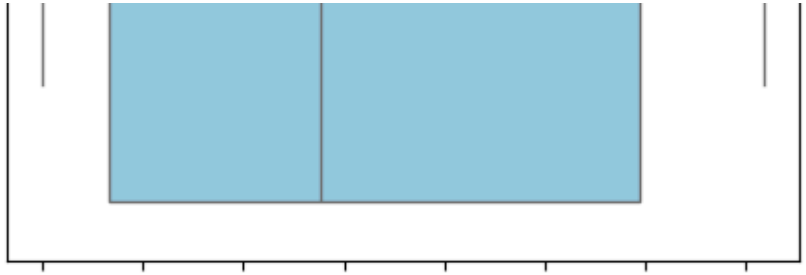
Device: ICTMicroclimate-07

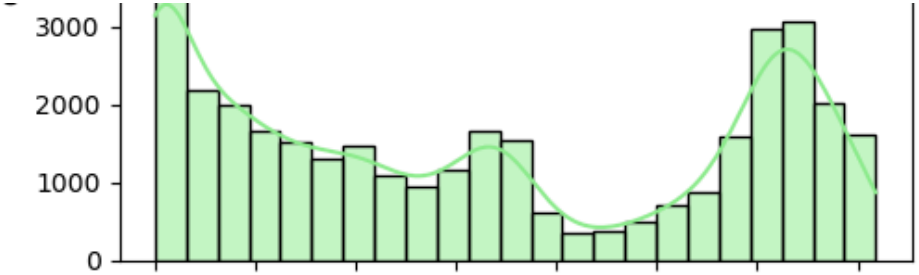




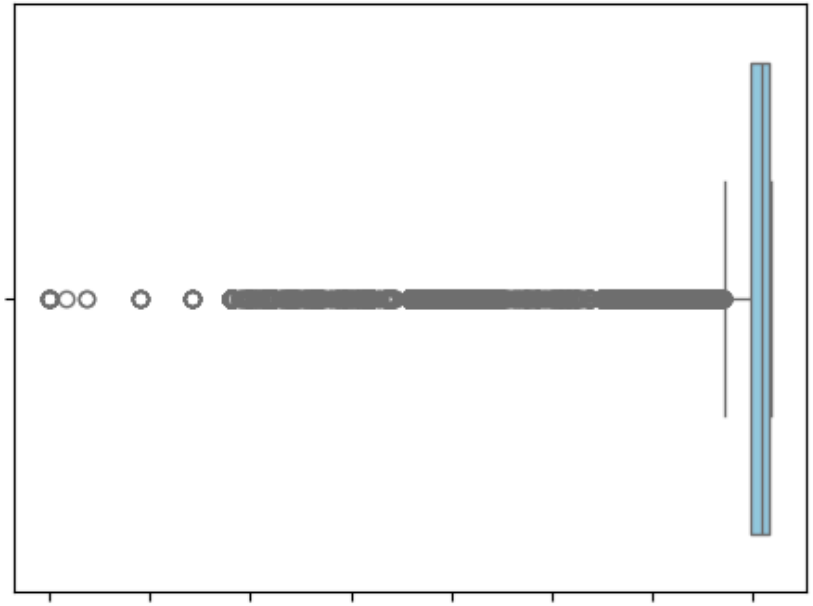




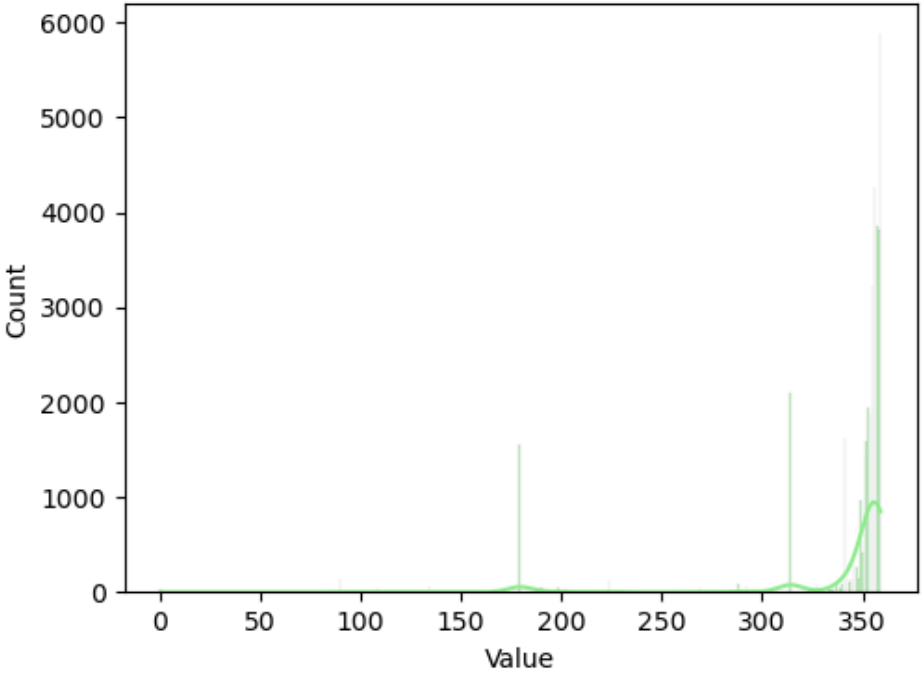




MaximumWindDirection — Boxplot

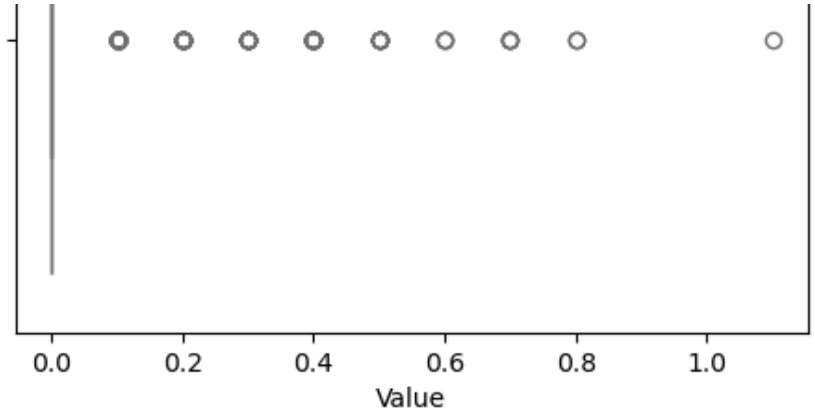


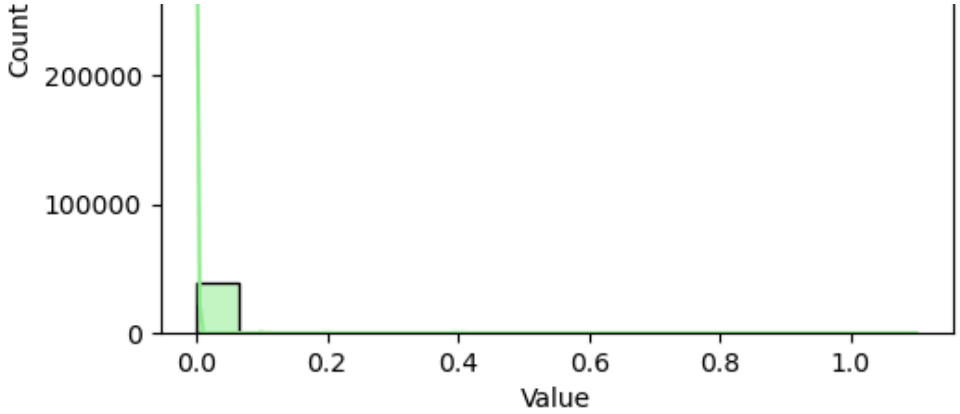
MaximumWindDirection — Distribution



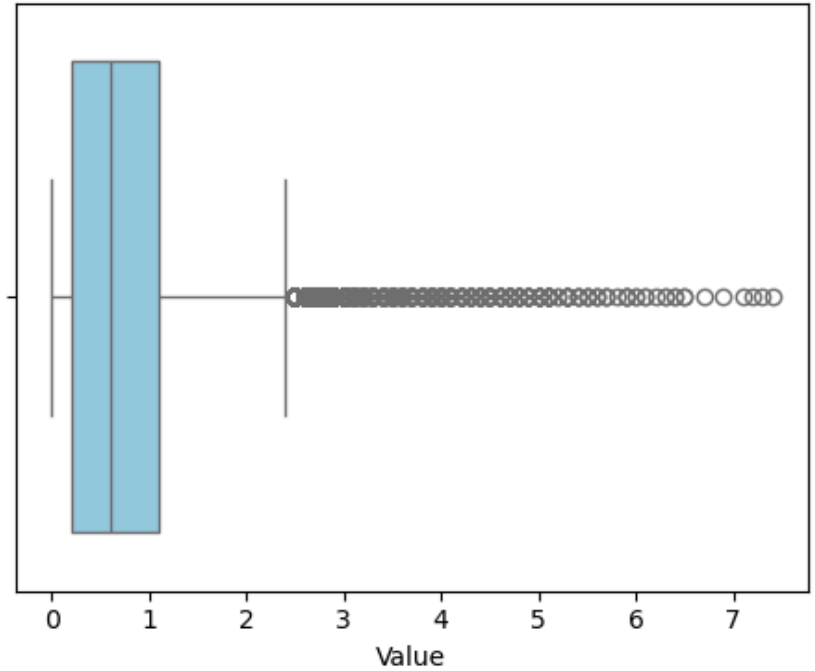
MinimumWindSpeed — Boxplot

MinimumWindSpeed — Distribution

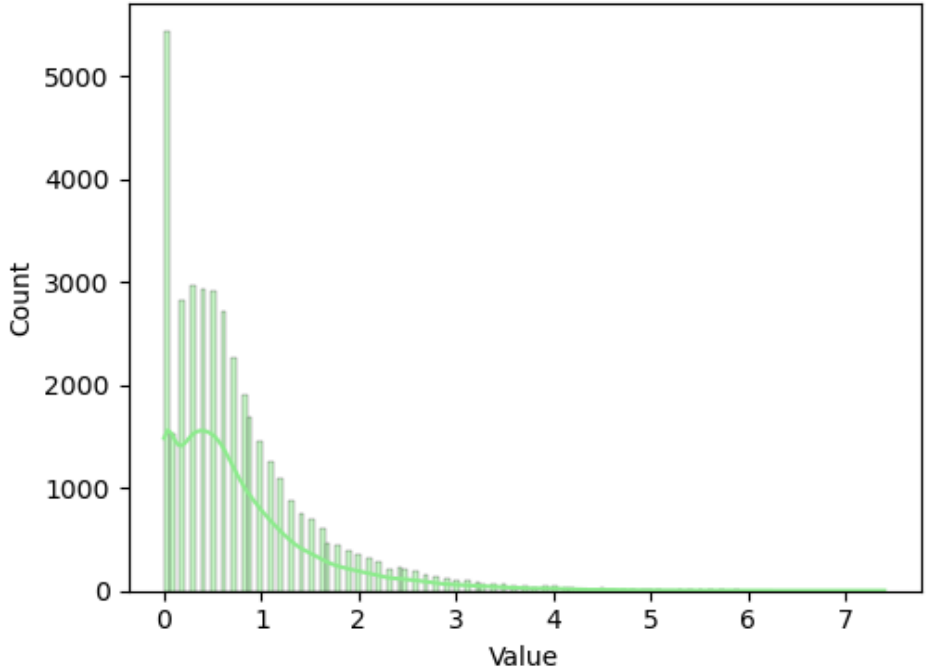




AverageWindSpeed — Boxplot

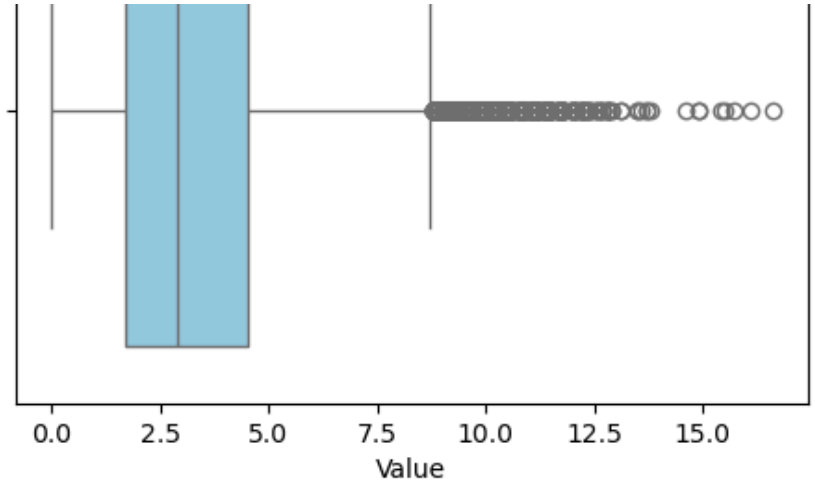


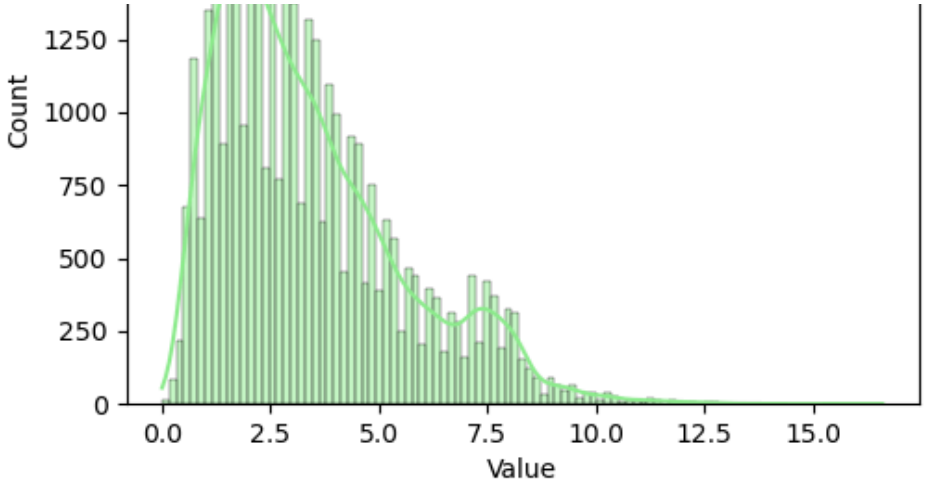
AverageWindSpeed — Distribution



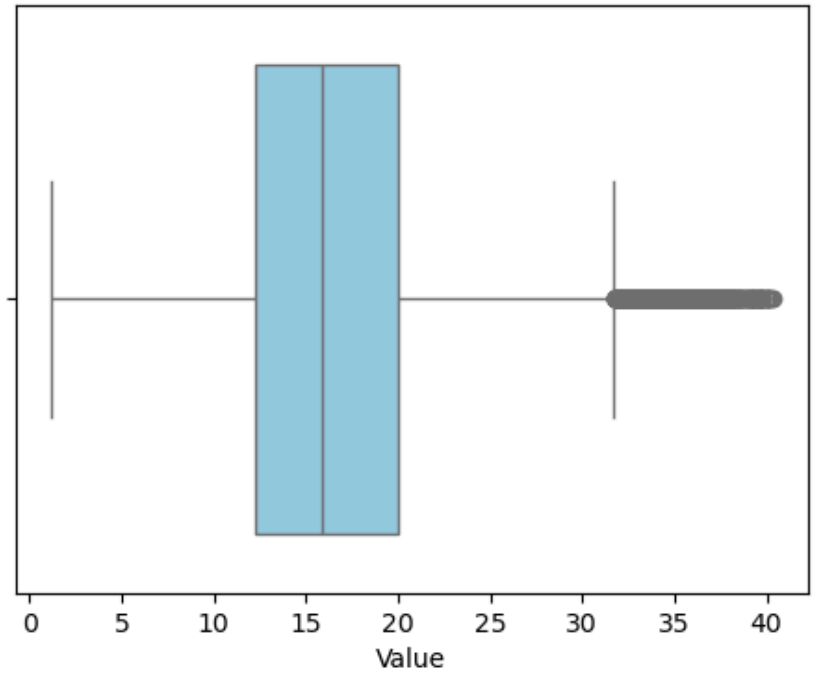
GustWindSpeed — Boxplot

GustWindSpeed — Distribution

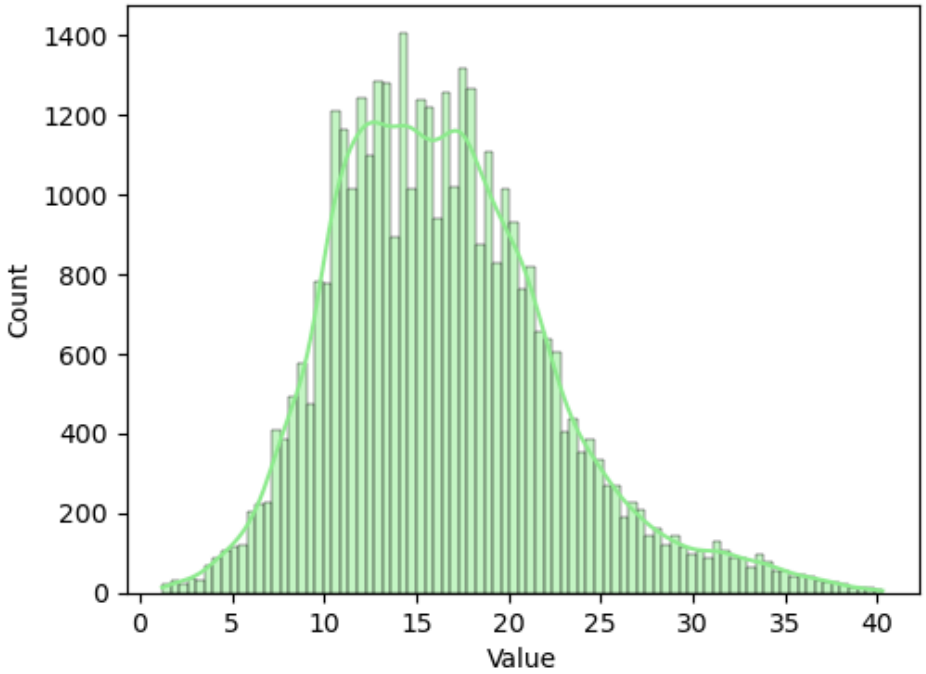




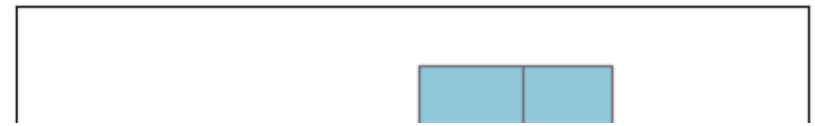
AirTemperature — Boxplot



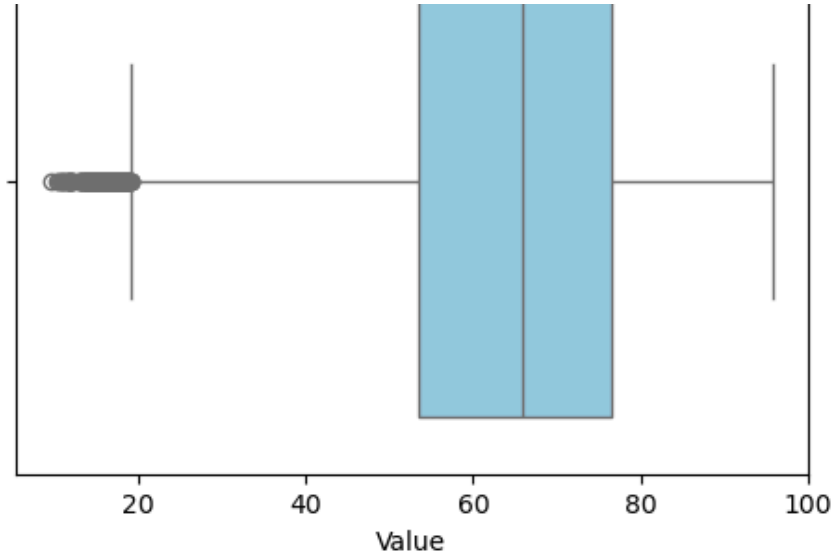
AirTemperature — Distribution

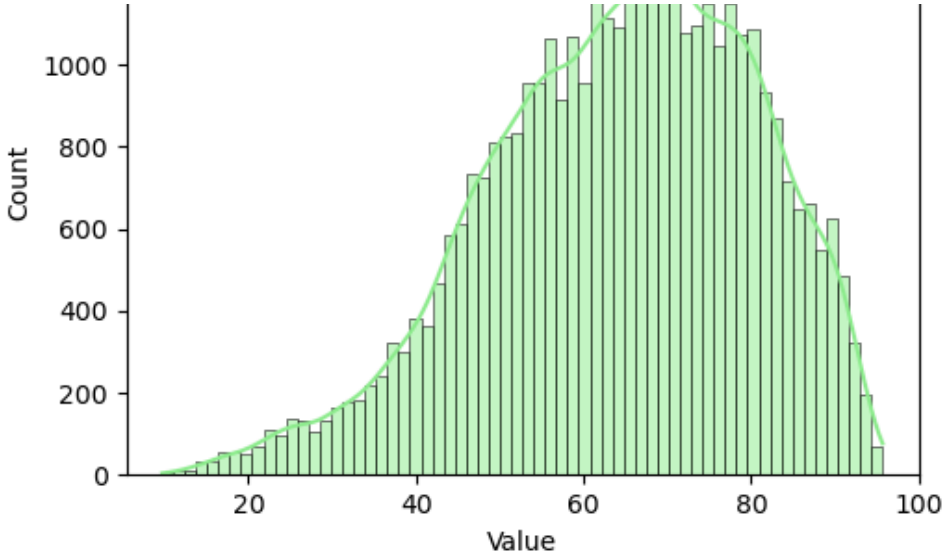


RelativeHumidity — Boxplot

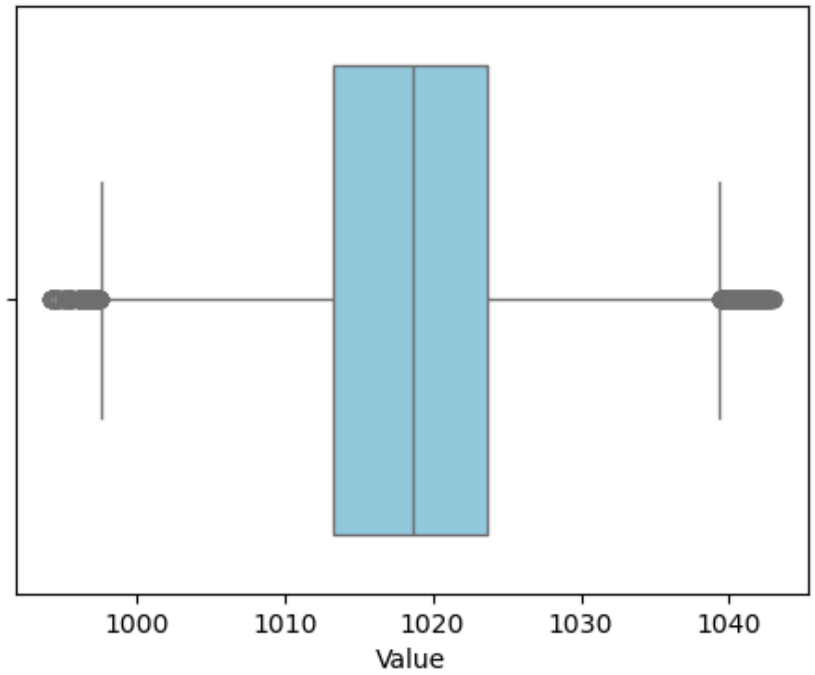


RelativeHumidity — Distribution

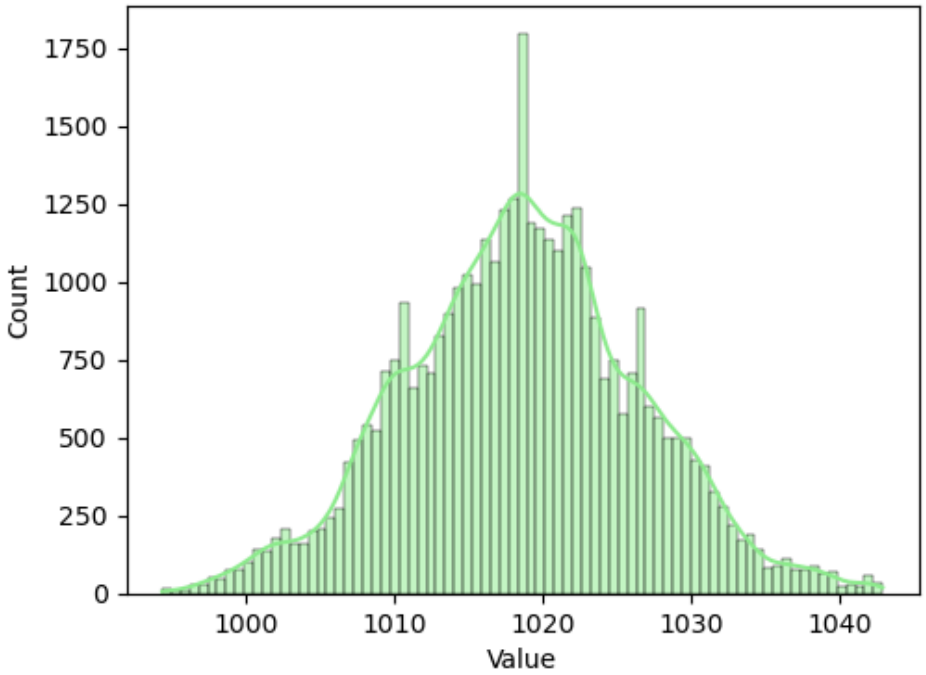




AtmosphericPressure — Boxplot

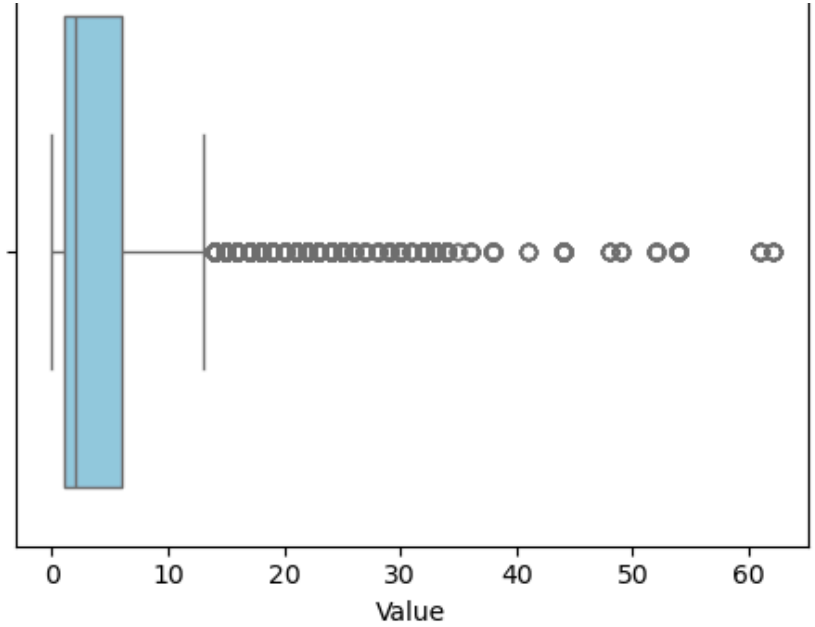


AtmosphericPressure — Distribution

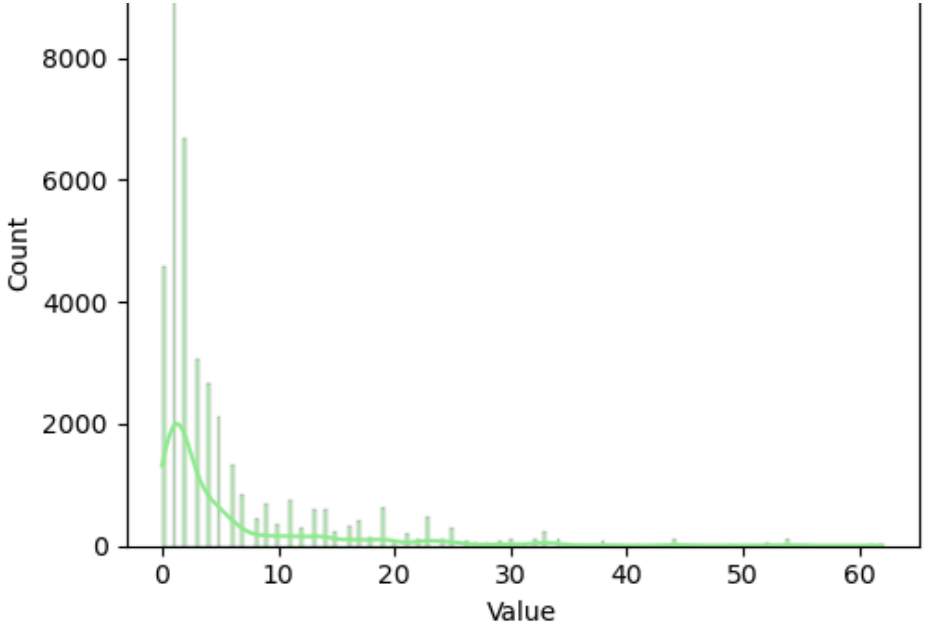


PM25 — Boxplot

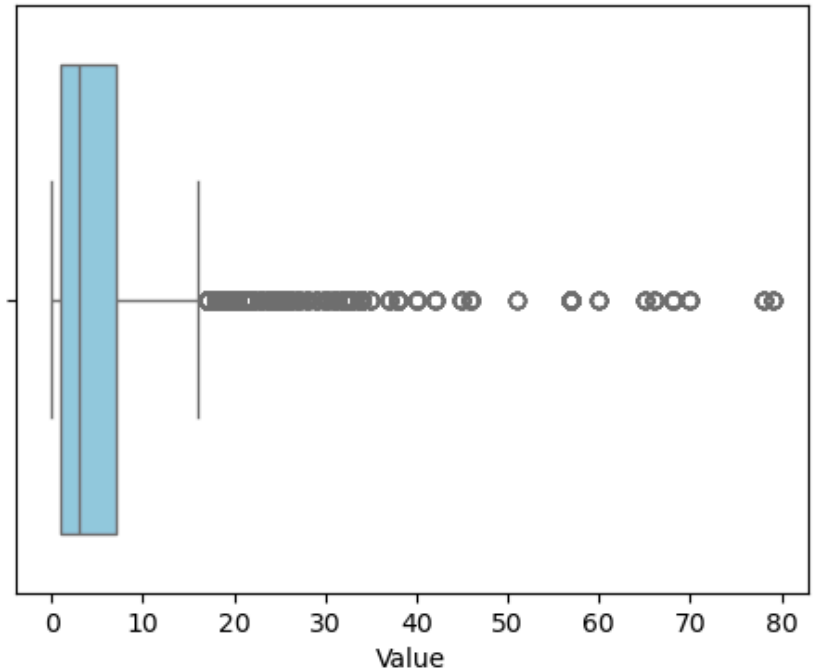
PM25 — Distribution



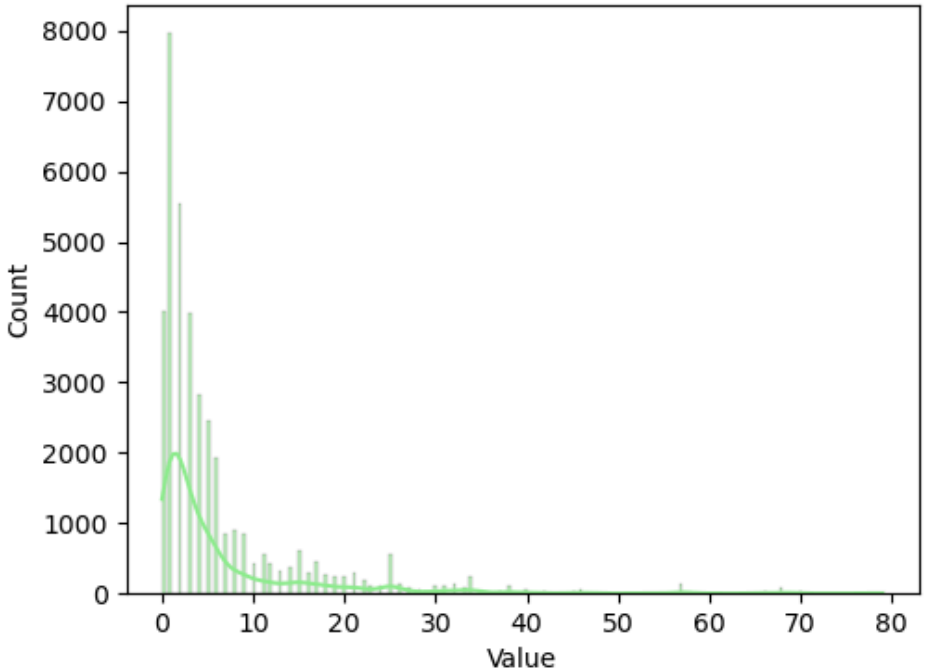
PM10 — Boxplot



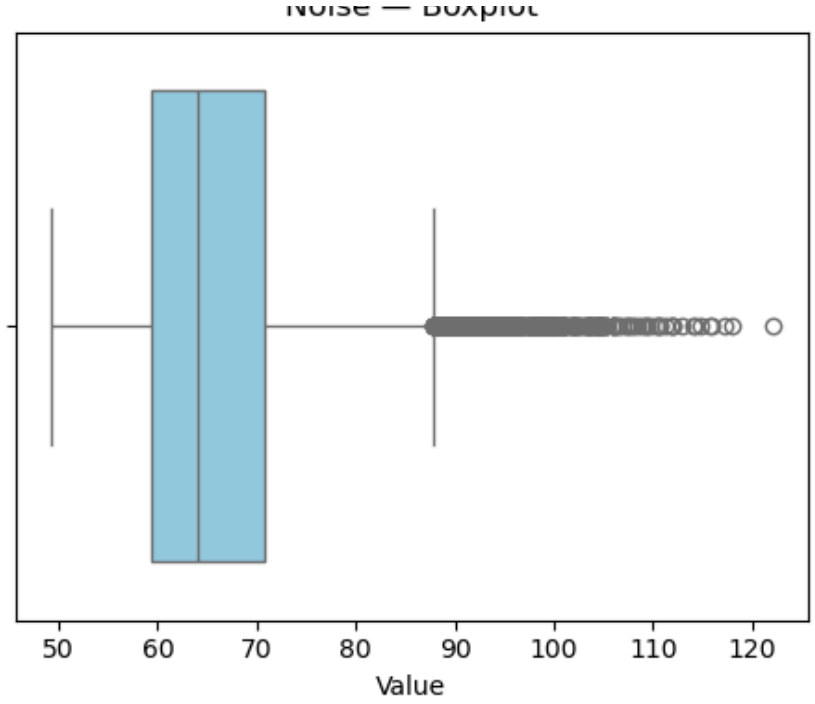
PM10 — Distribution

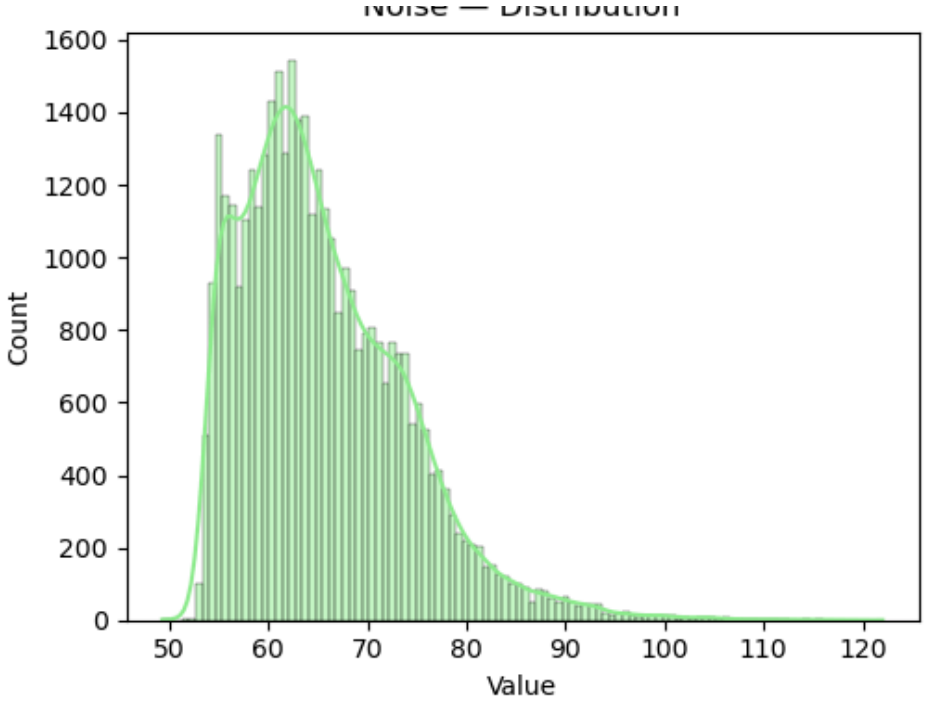


Noise — Boxplot

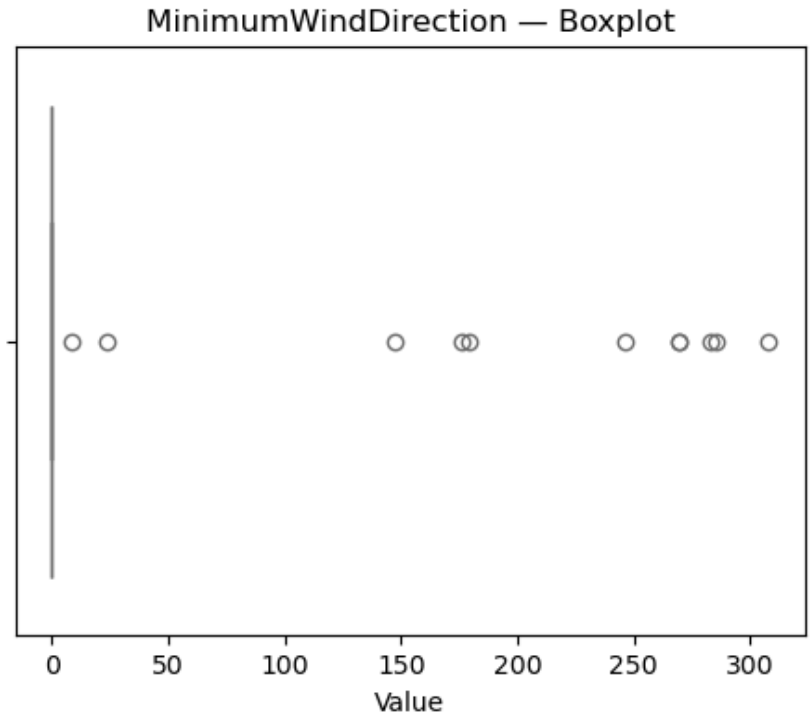


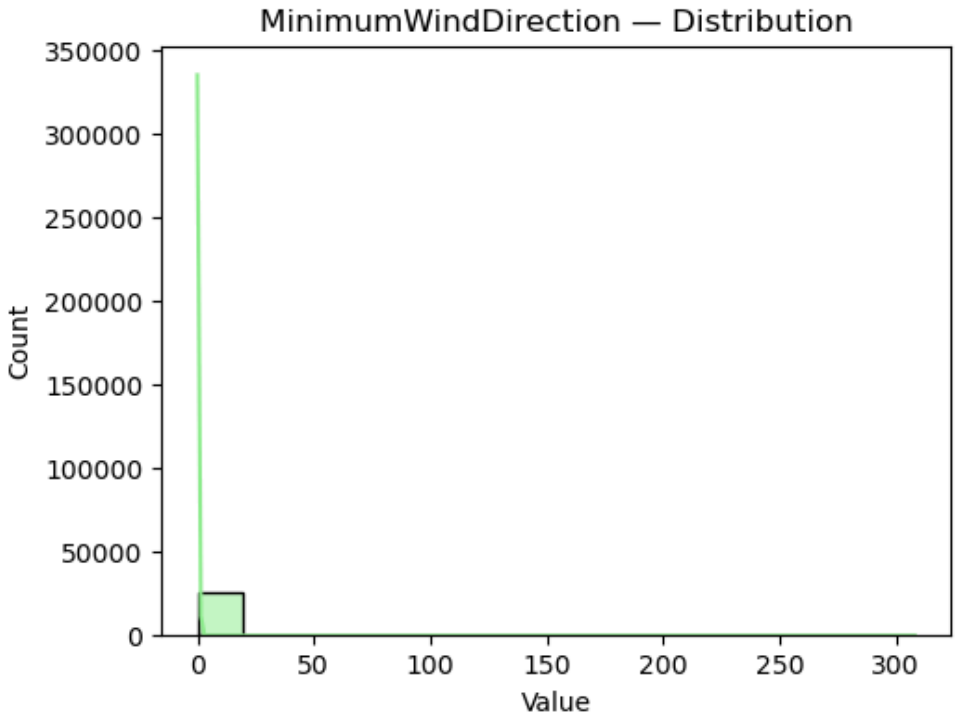
Noise — Distribution

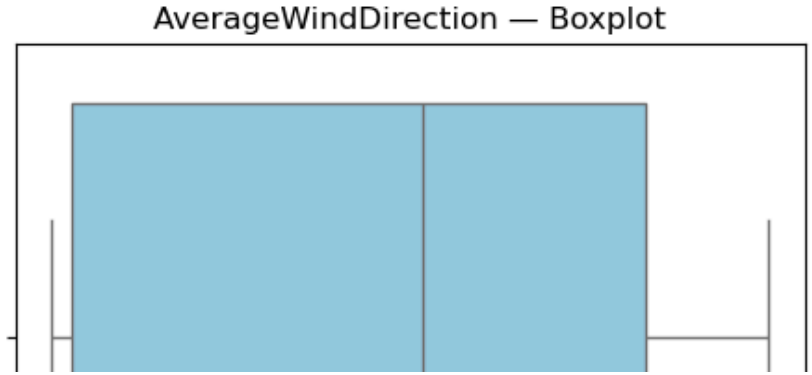

Noise — Boxplot
50
60
70
80
90
100
110
120
Value

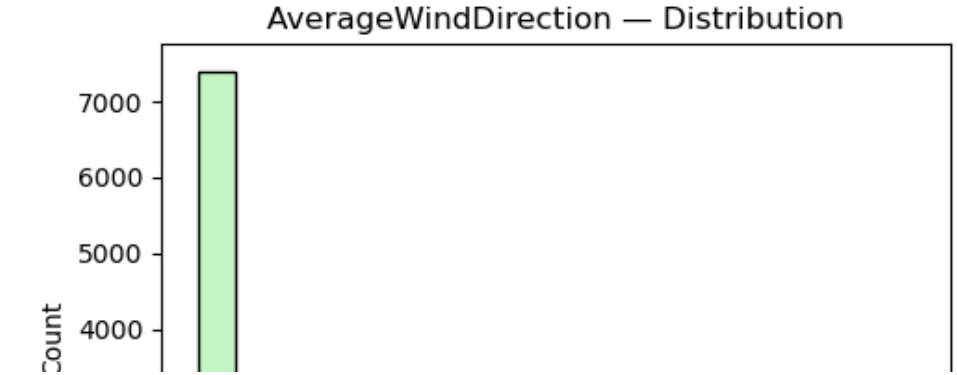

Noise — Distribution
1600
1400
1200
1000
800
600
400
200
0
Count
50
60
70
80
90
100
110
120
Value

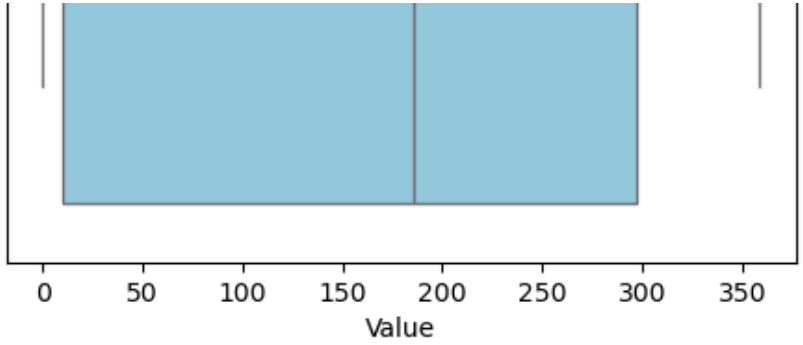
Device: ICTMicroclimate-04

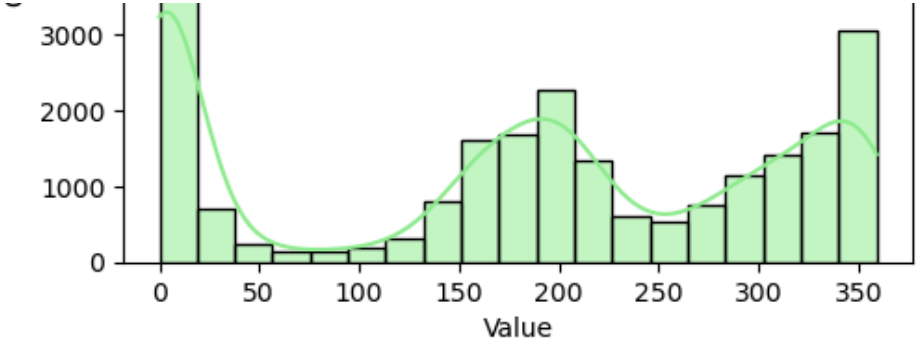




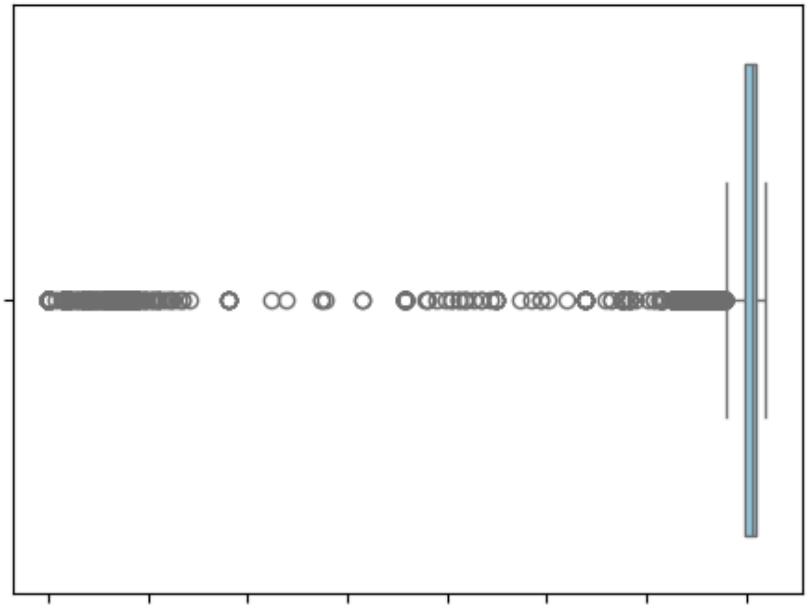




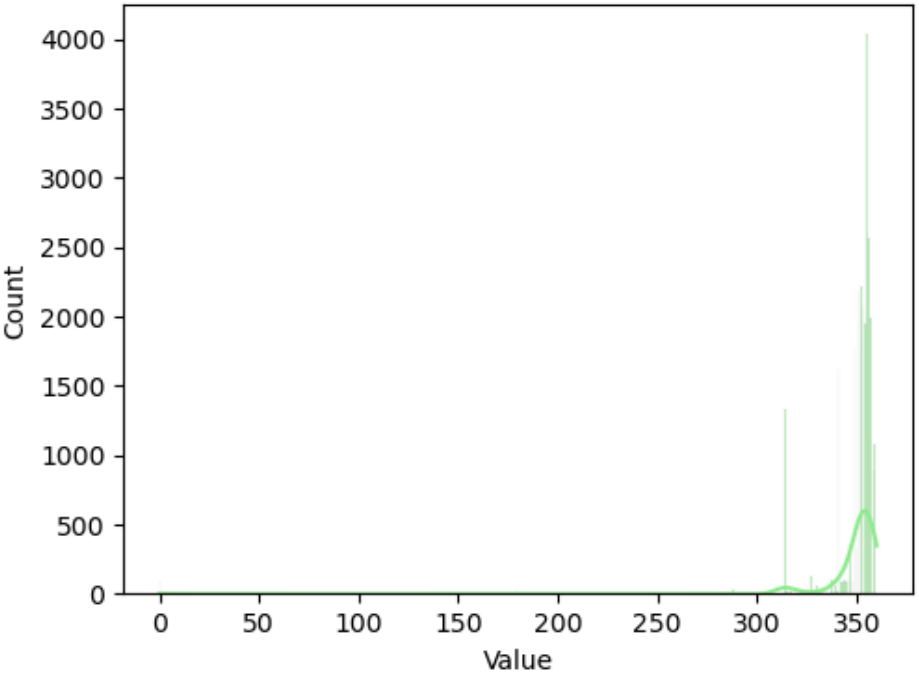




MaximumWindDirection — Boxplot

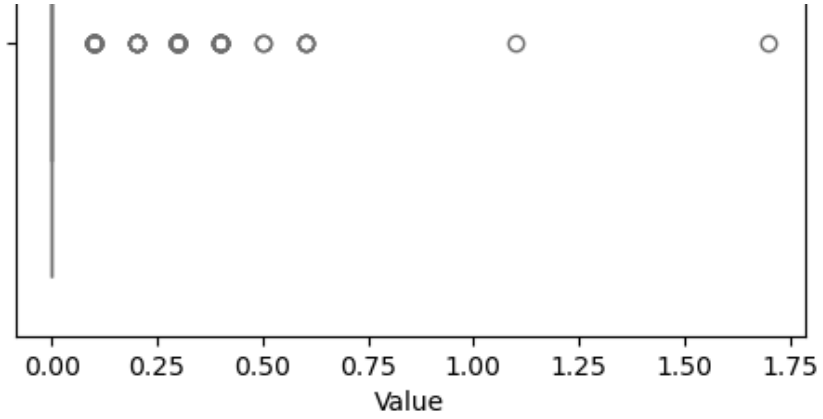


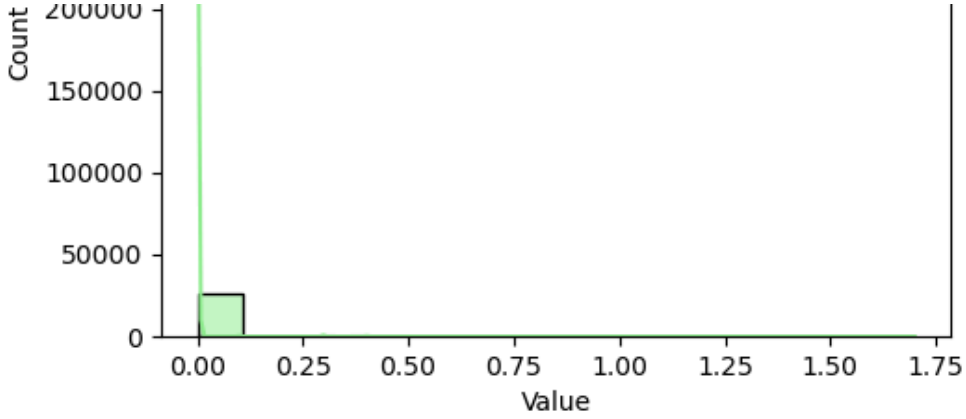
MaximumWindDirection — Distribution



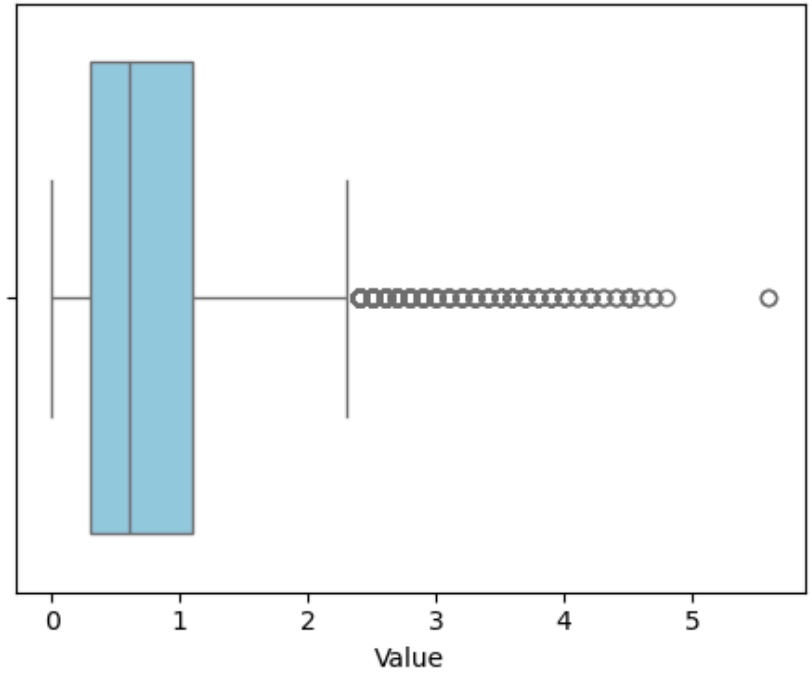
MinimumWindSpeed — Boxplot

MinimumWindSpeed — Distribution

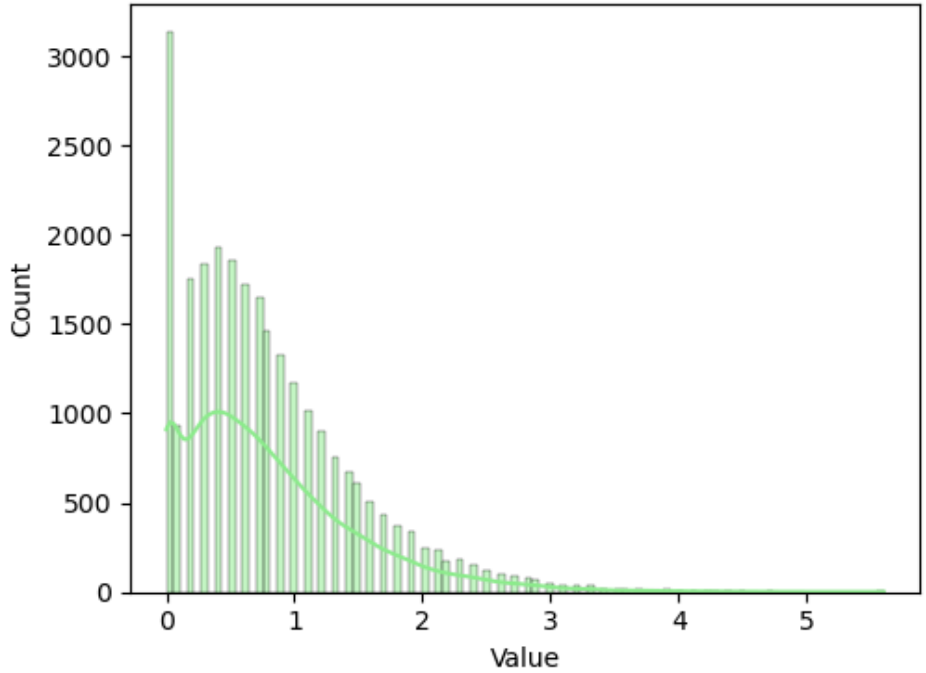




AverageWindSpeed — Boxplot

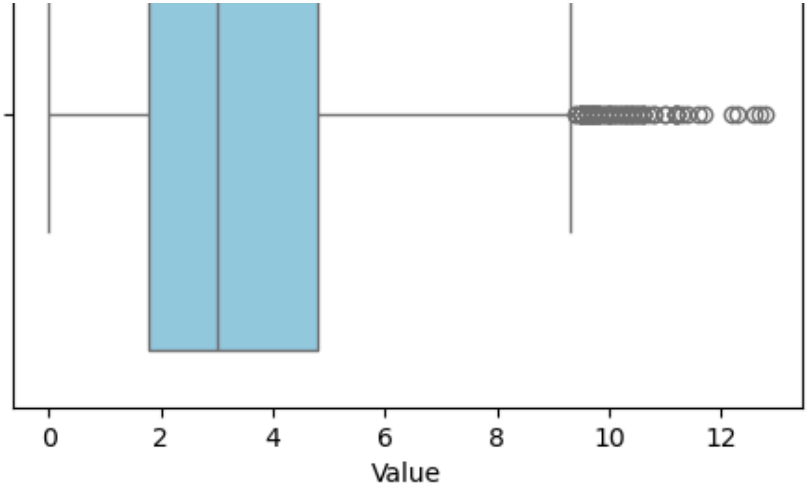


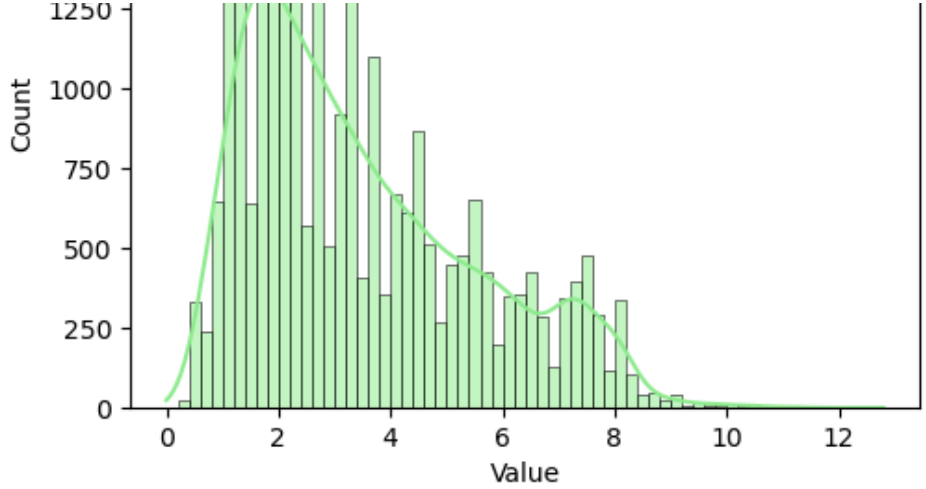
AverageWindSpeed — Distribution



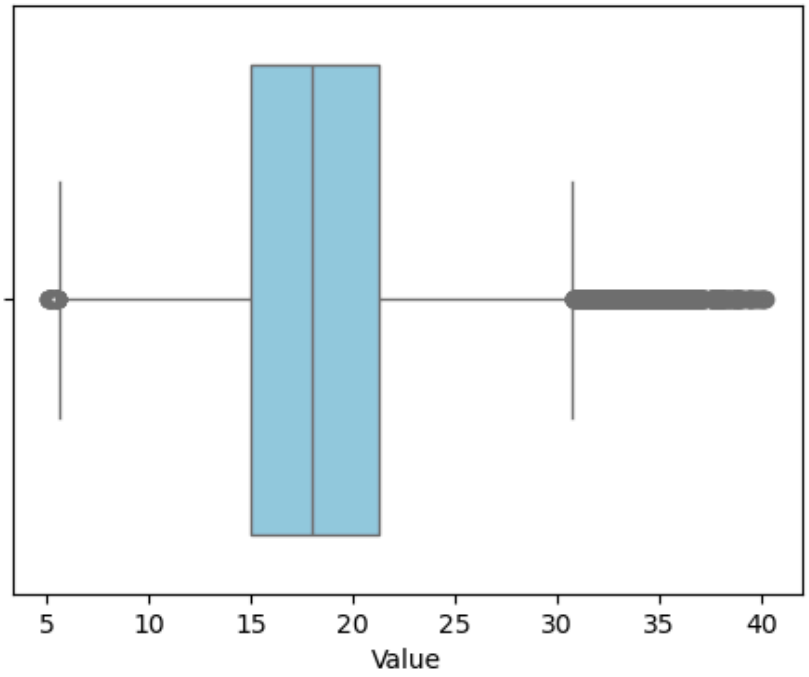
GustWindSpeed — Boxplot

GustWindSpeed — Distribution

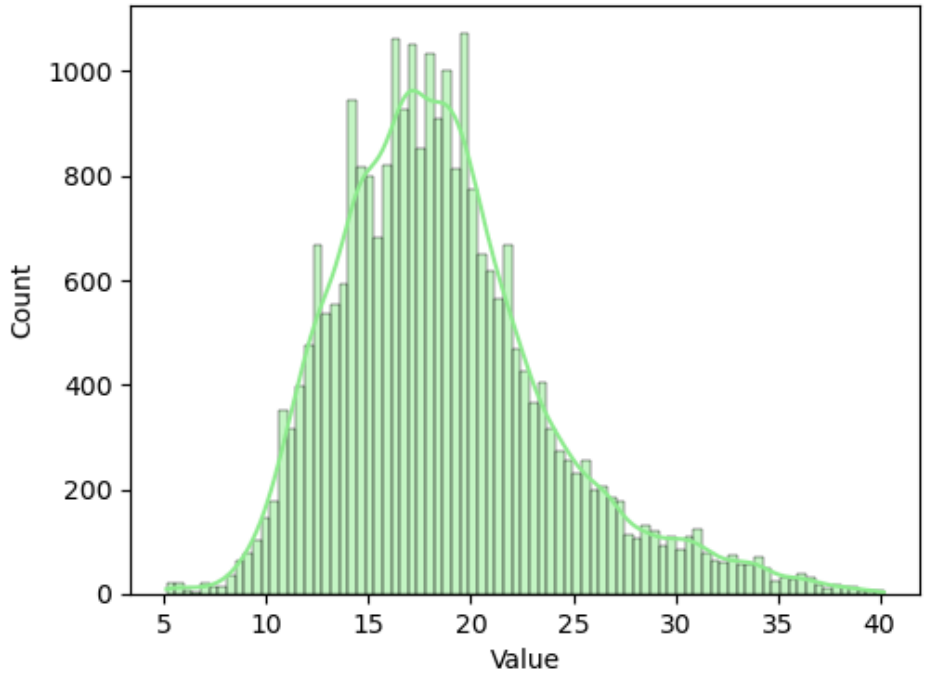




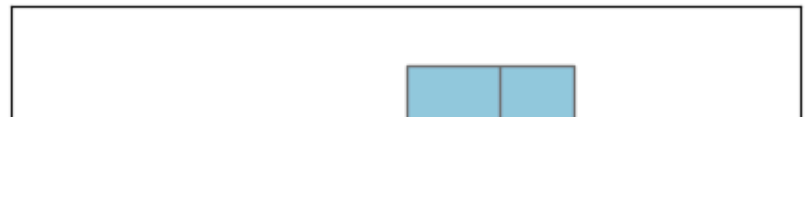
AirTemperature — Boxplot



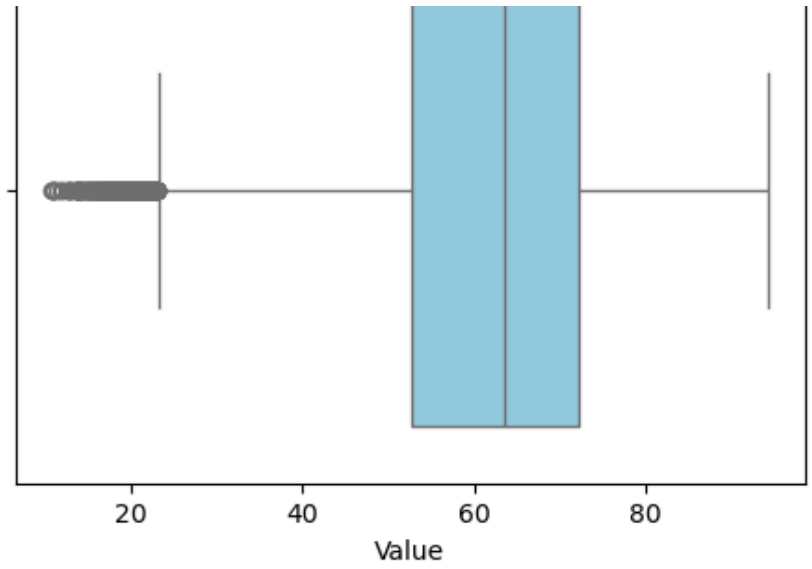
AirTemperature — Distribution

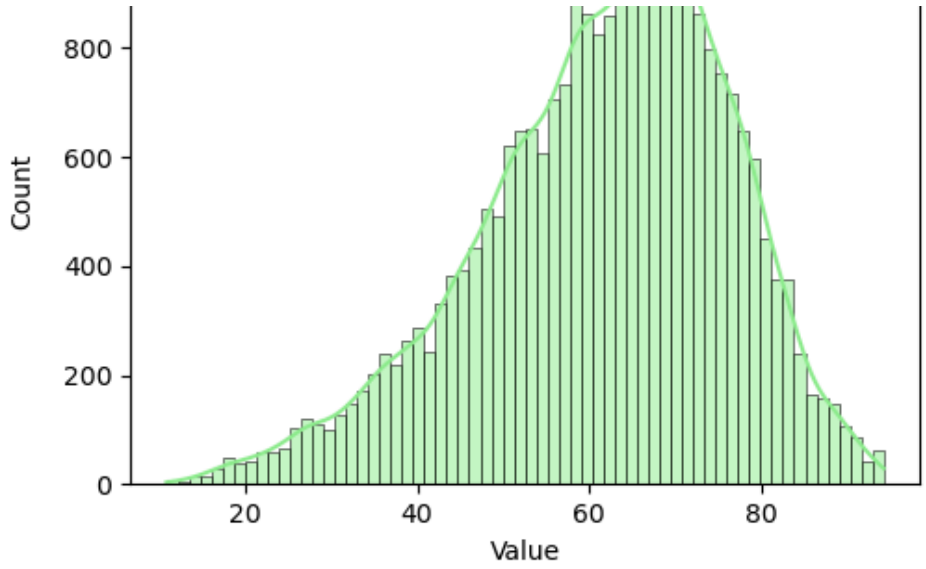


RelativeHumidity — Boxplot

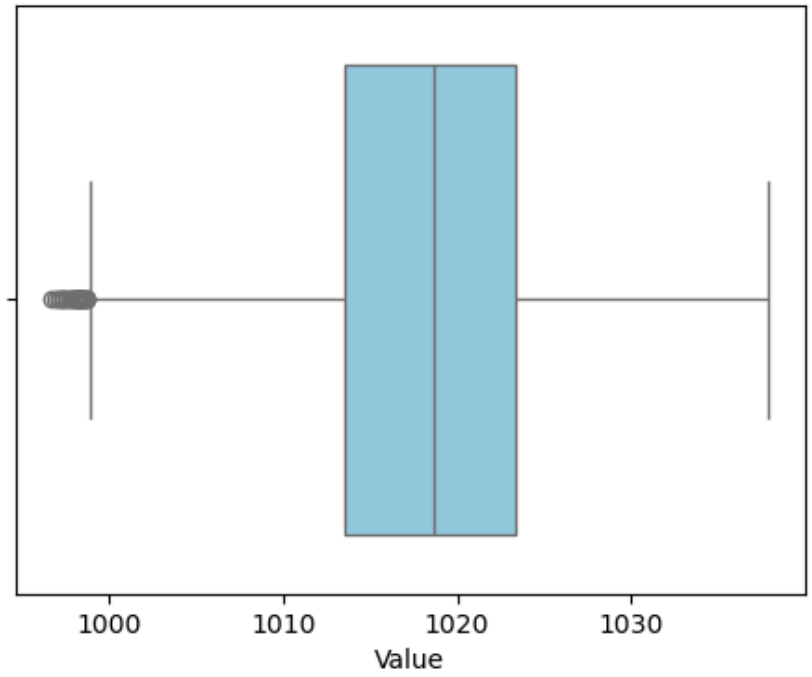


RelativeHumidity — Distribution

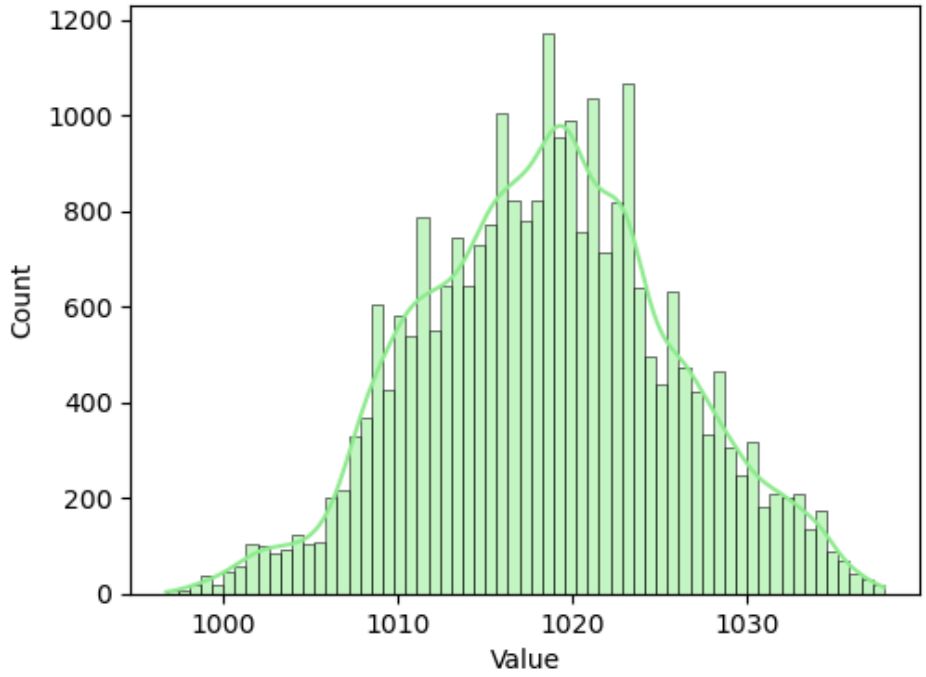




AtmosphericPressure — Boxplot

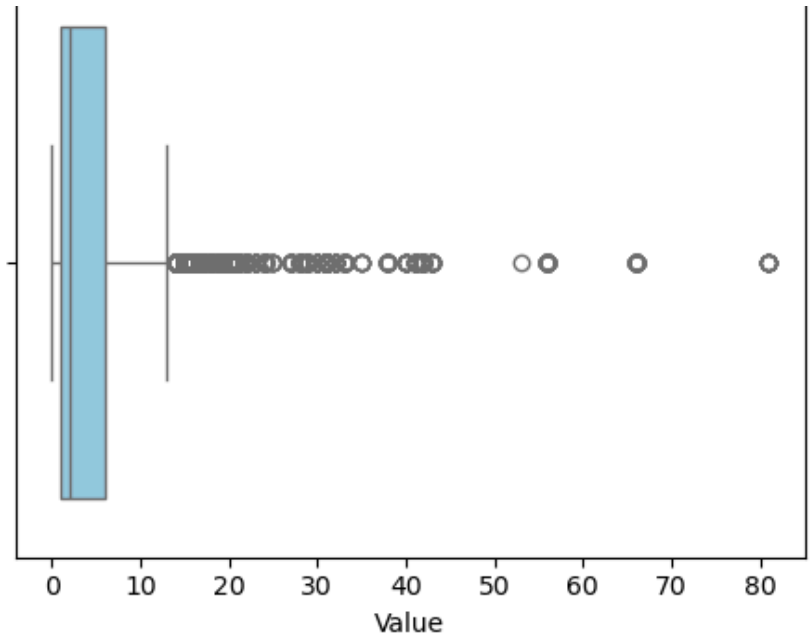


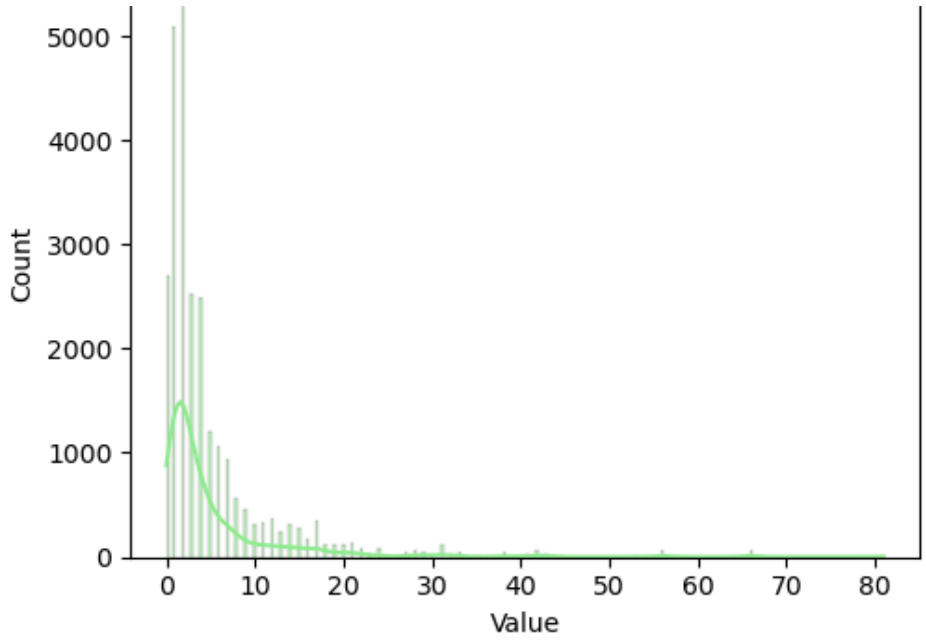
AtmosphericPressure — Distribution



PM25 — Boxplot

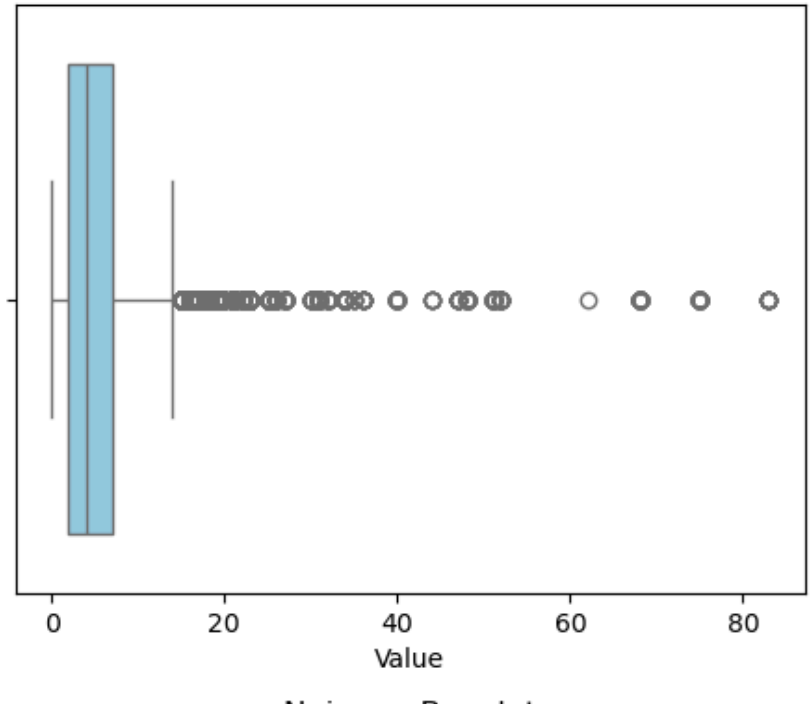
PM25 — Distribution

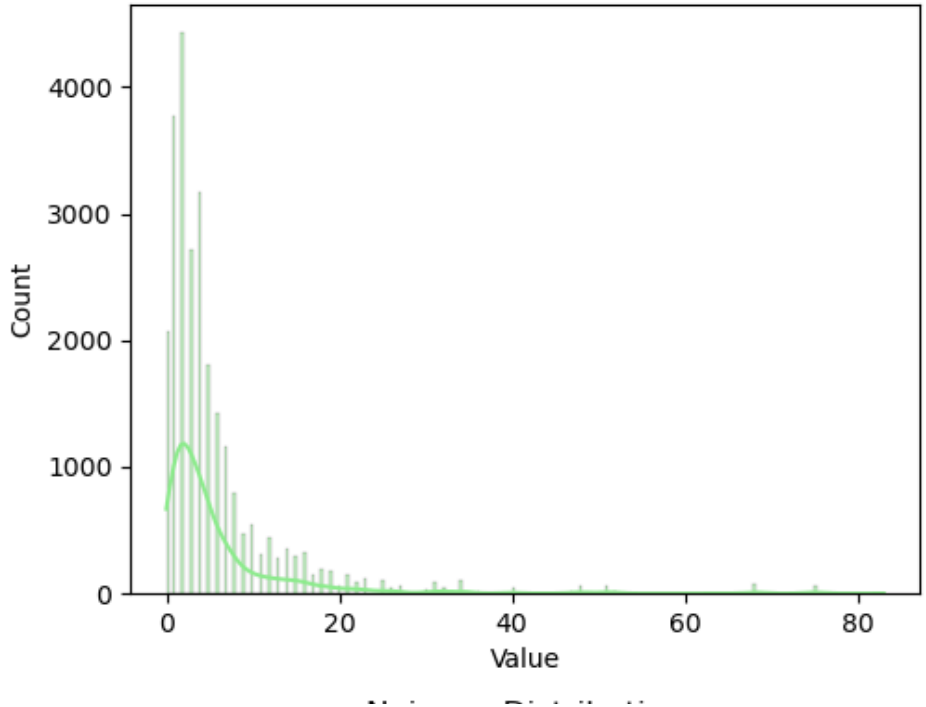


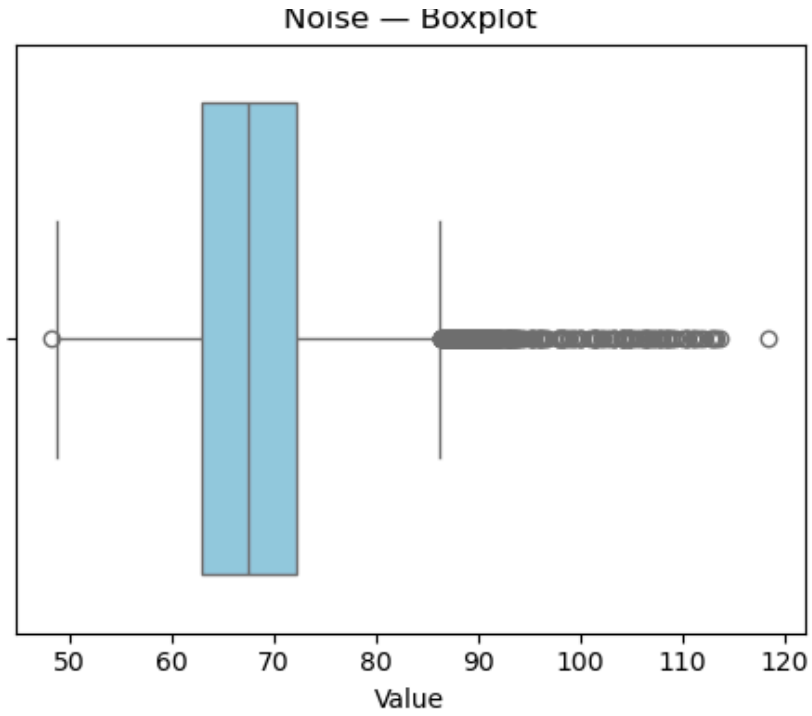


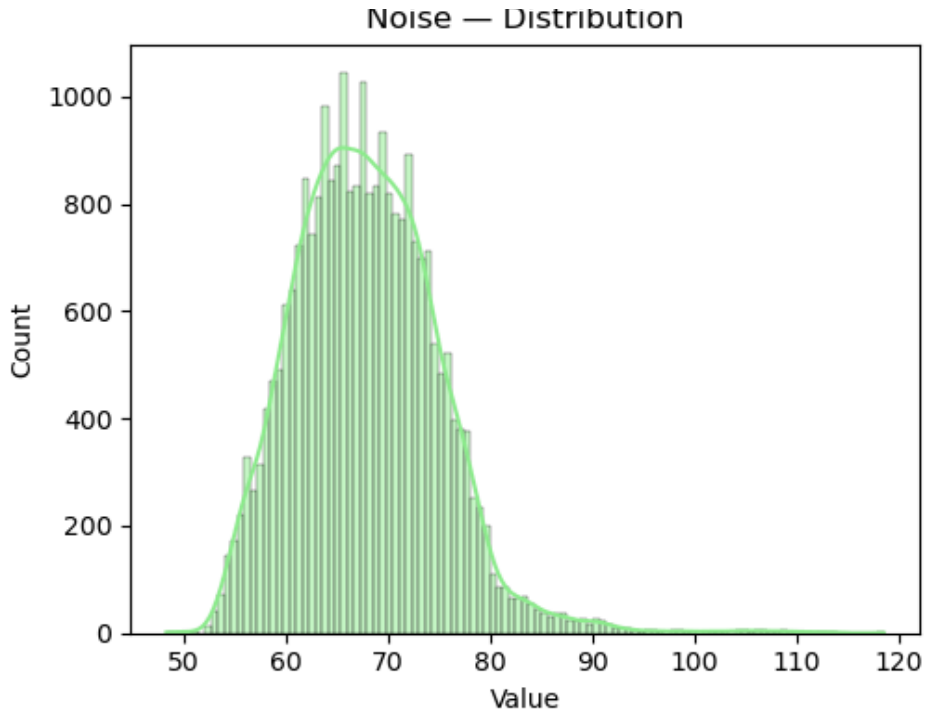
PM10 — Boxplot

PM10 — Distribution

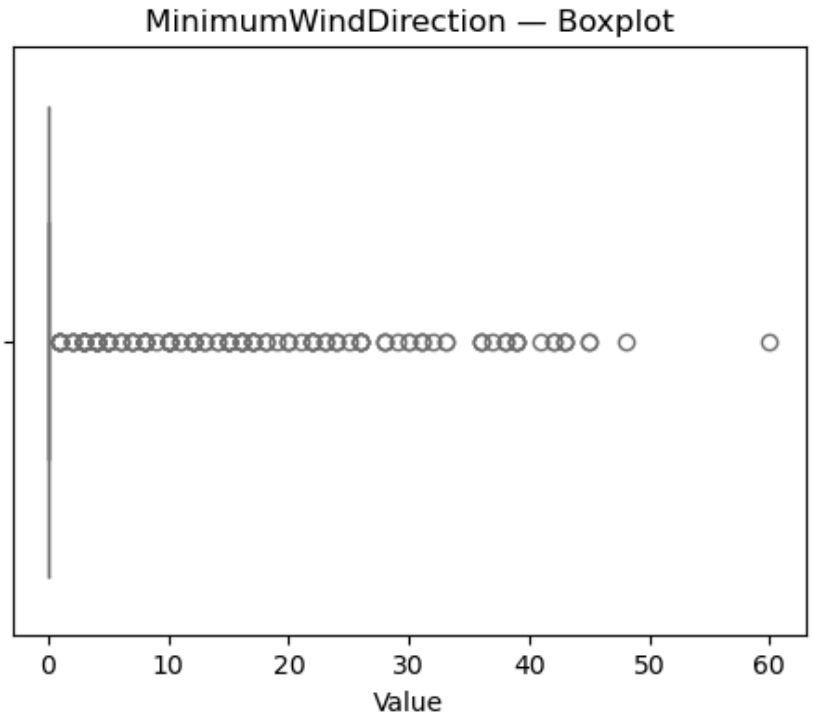


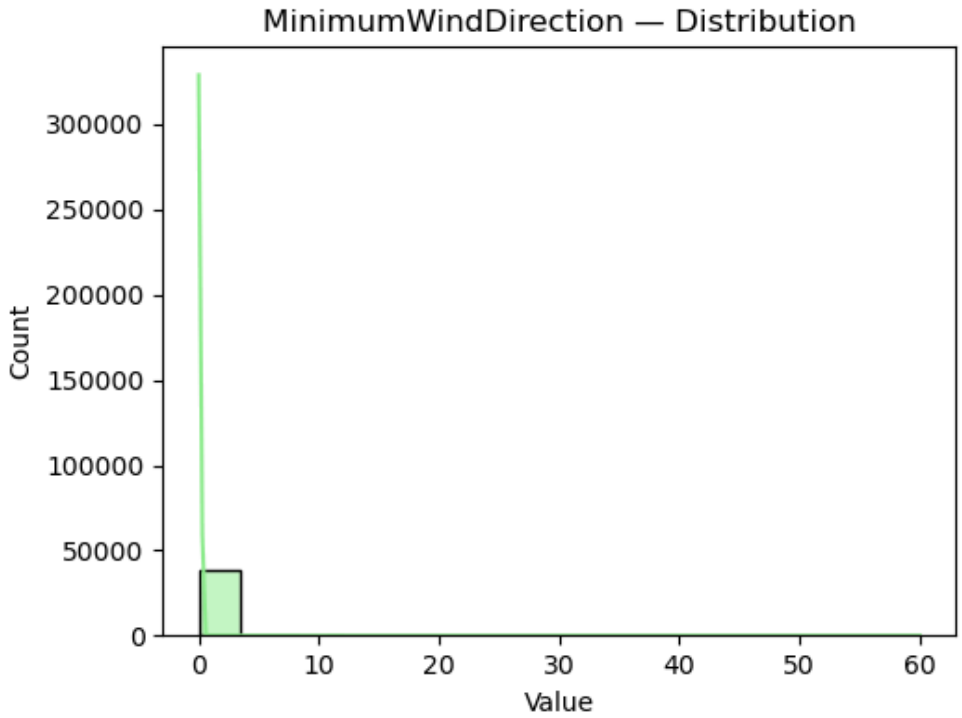


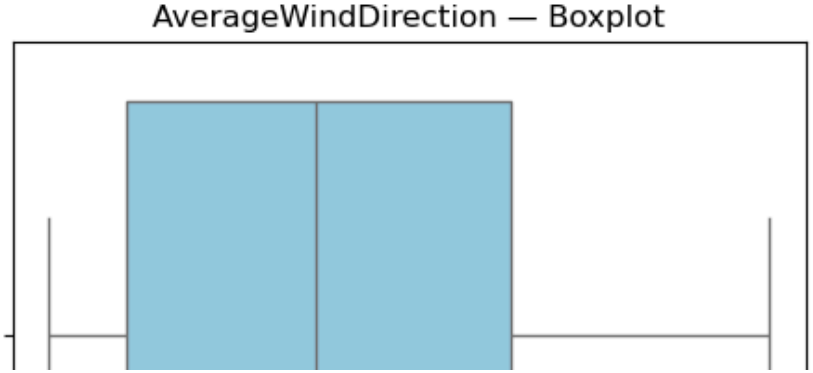

Noise — Boxplot
50
60
70
80
90
100
110
120
Value

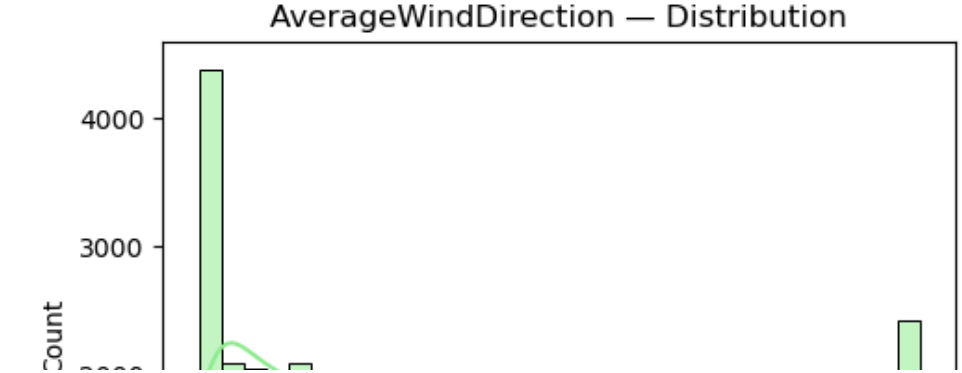

Noise — Distribution
1000
800
600
400
200
0
Count
50
60
70
80
90
100
110
120
Value

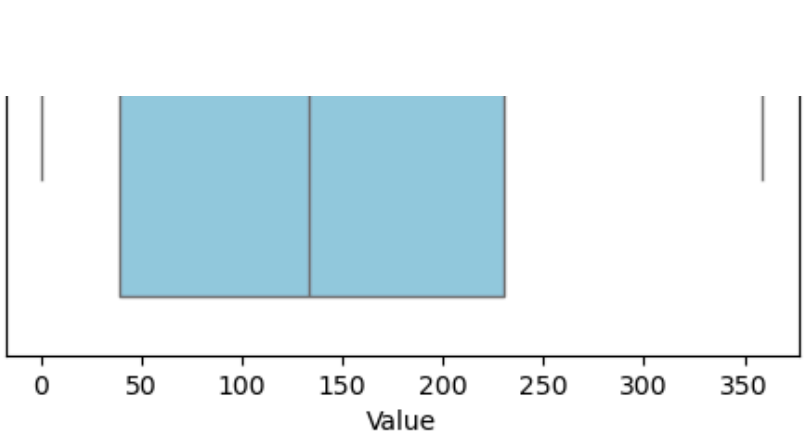
Device: ICTMicroclimate-06

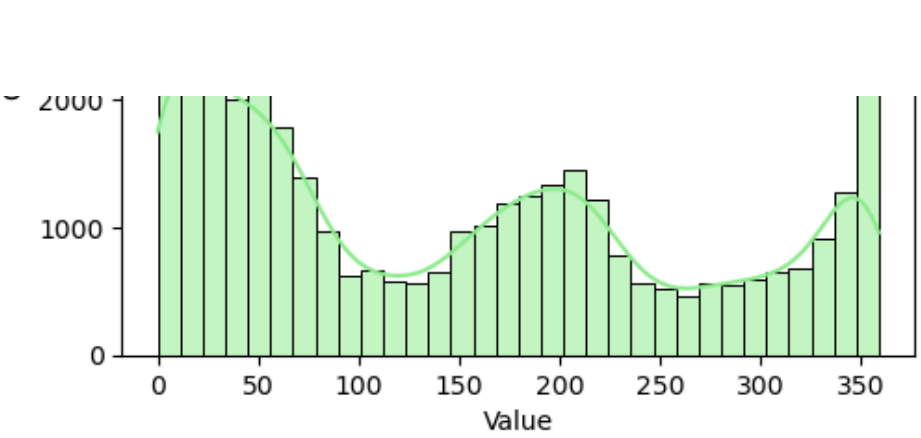




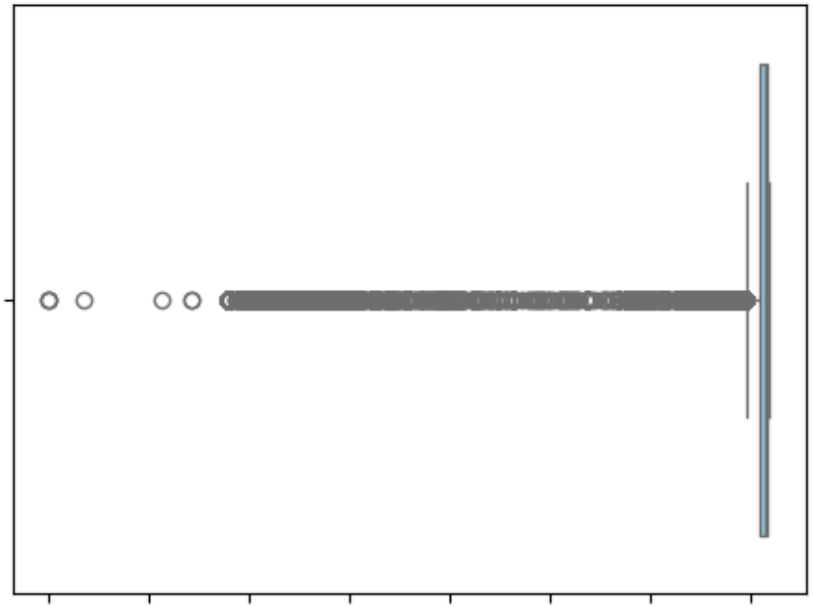




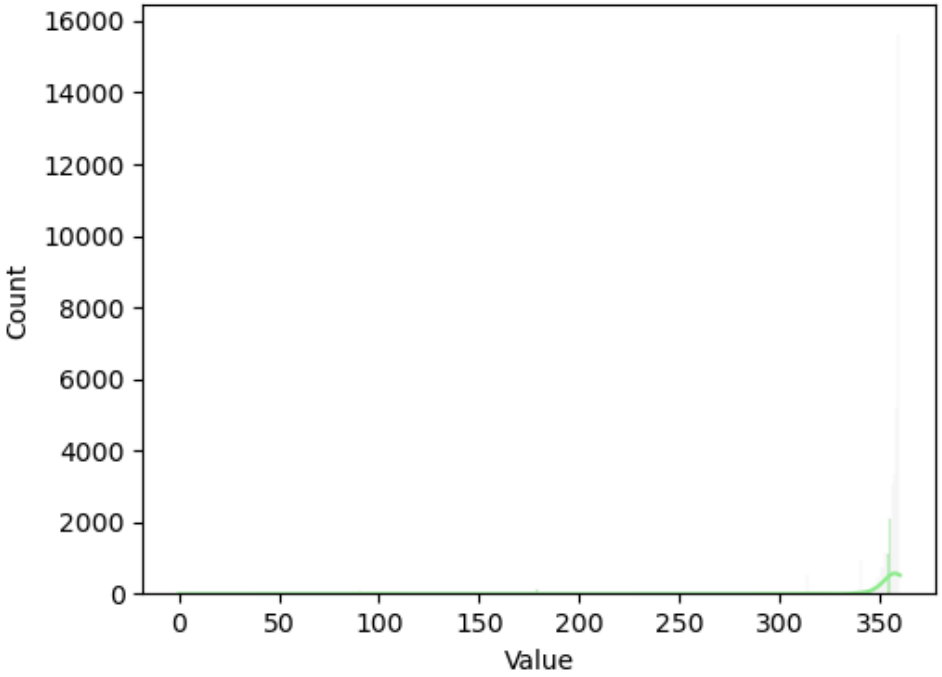




MaximumWindDirection — Boxplot

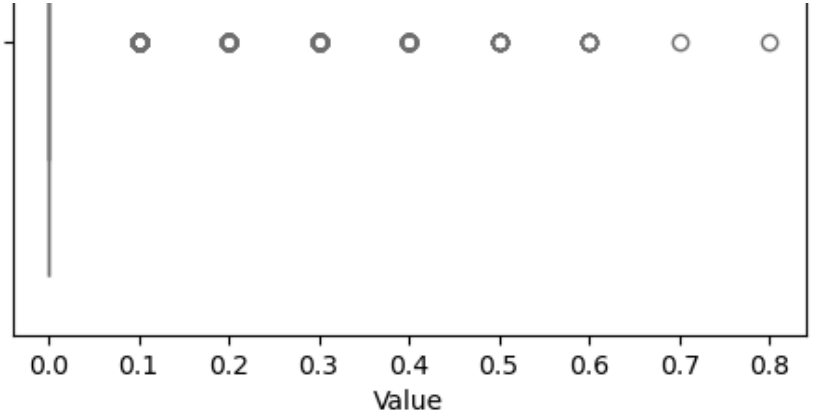


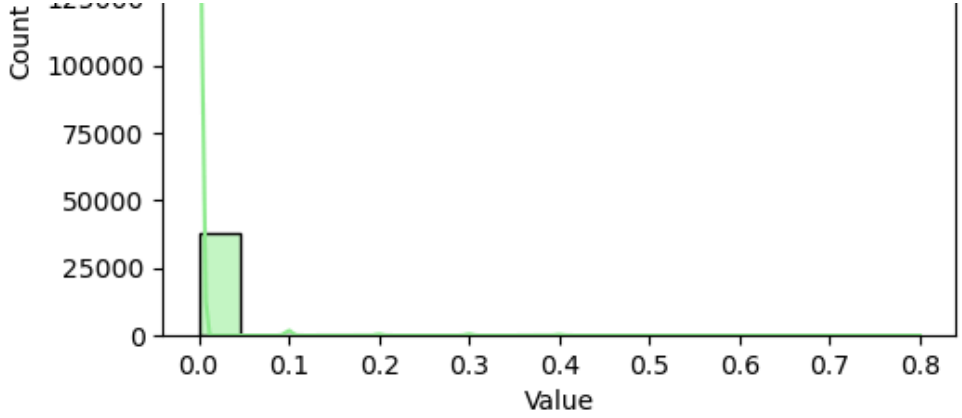
MaximumWindDirection — Distribution



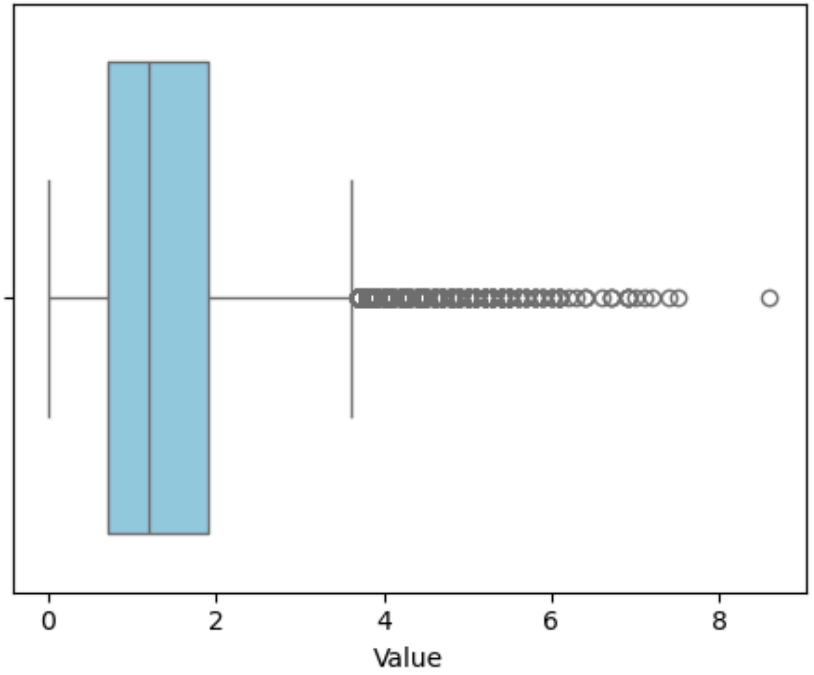
MinimumWindSpeed — Boxplot

MinimumWindSpeed — Distribution

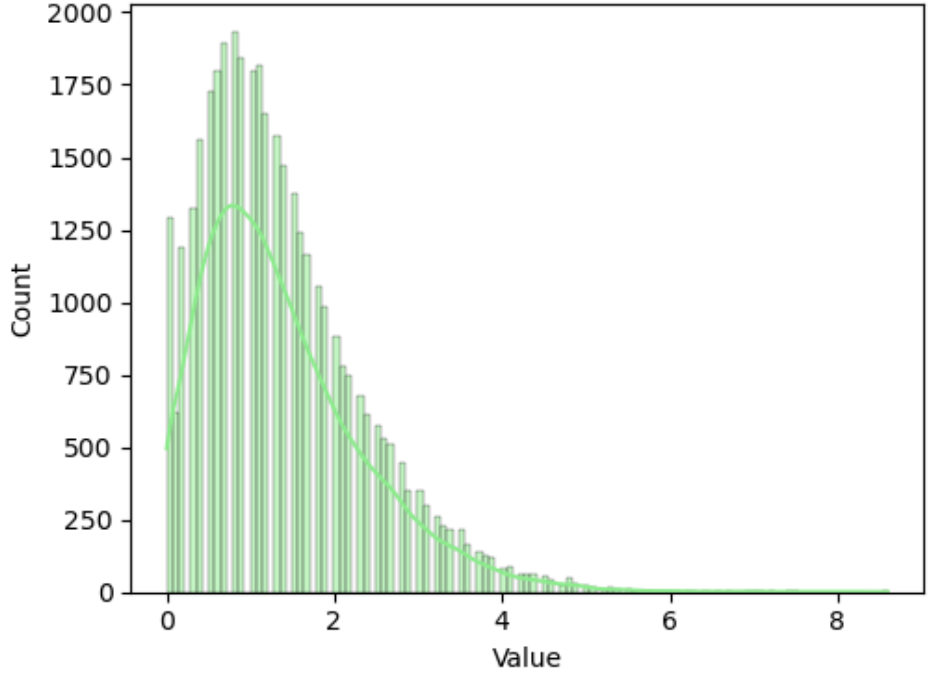




AverageWindSpeed — Boxplot

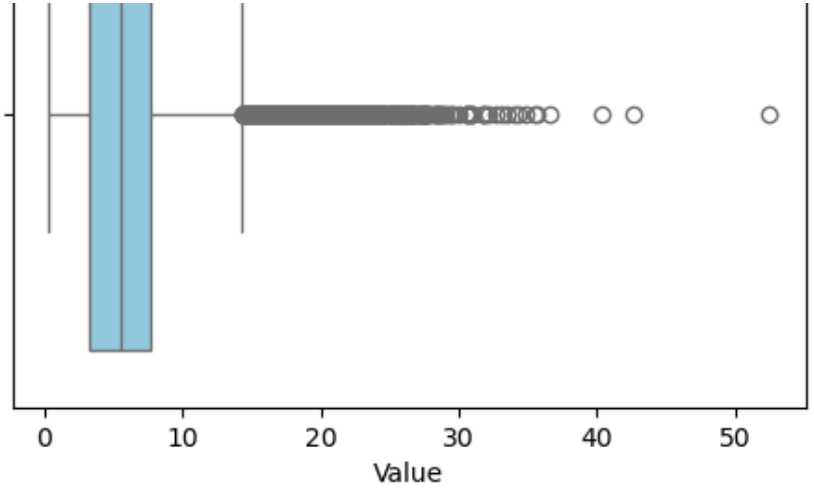


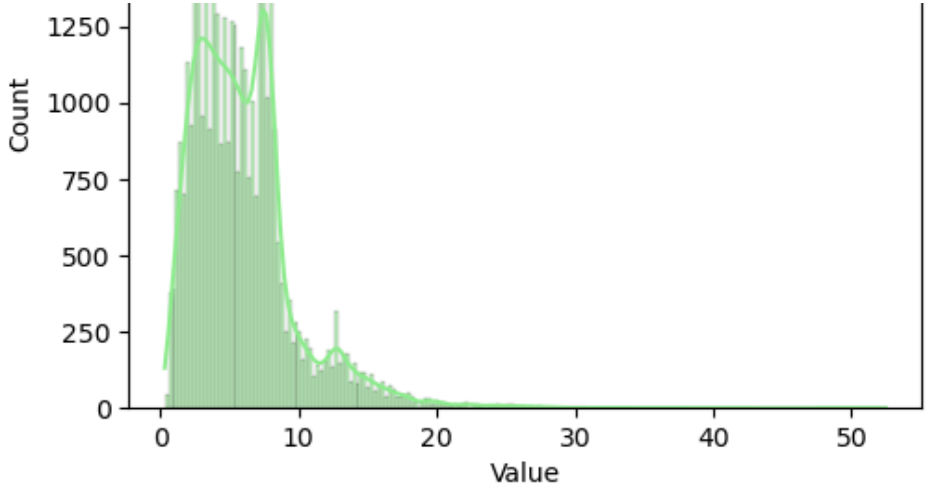
AverageWindSpeed — Distribution



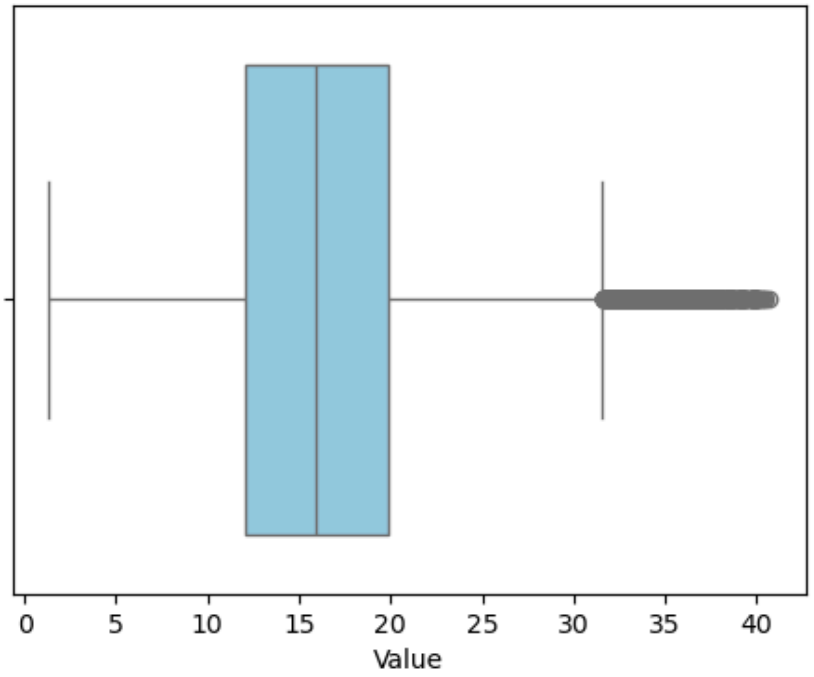
GustWindSpeed — Boxplot

GustWindSpeed — Distribution

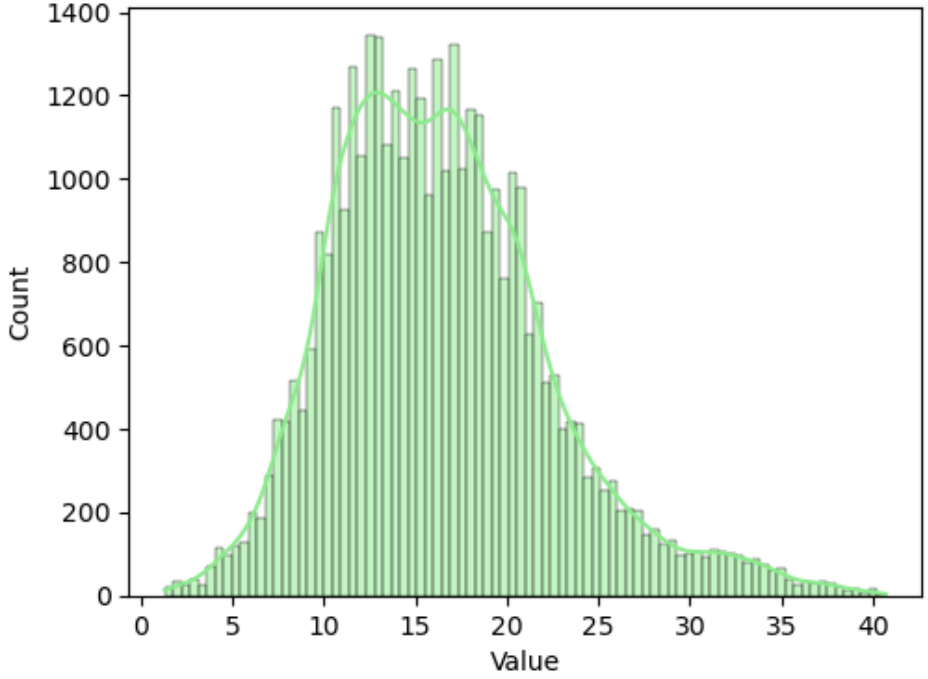




AirTemperature — Boxplot

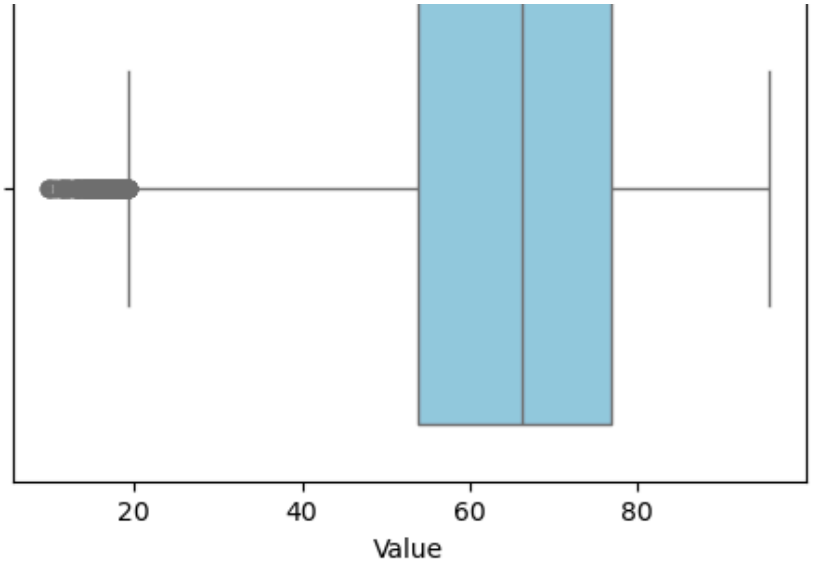


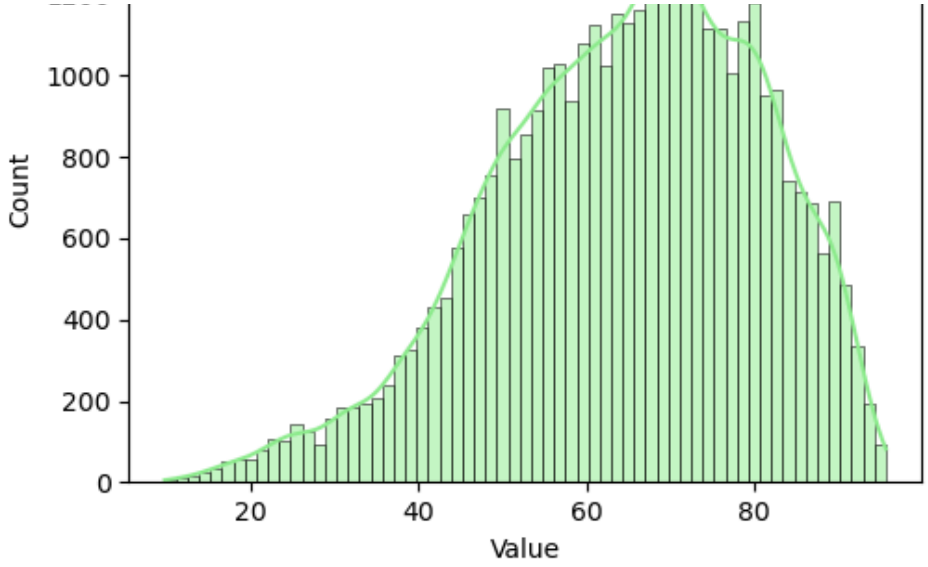
AirTemperature — Distribution



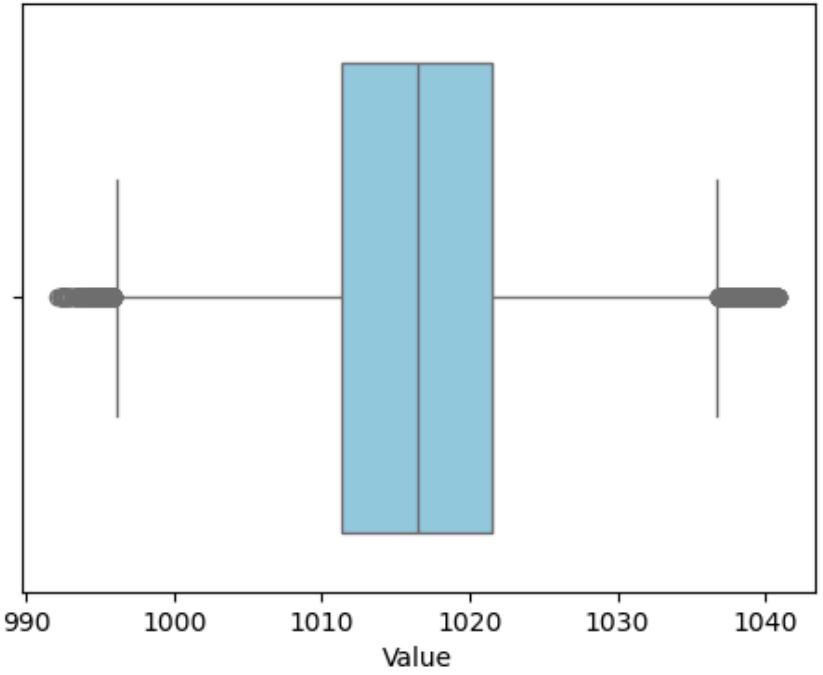
RelativeHumidity — Boxplot

RelativeHumidity — Distribution

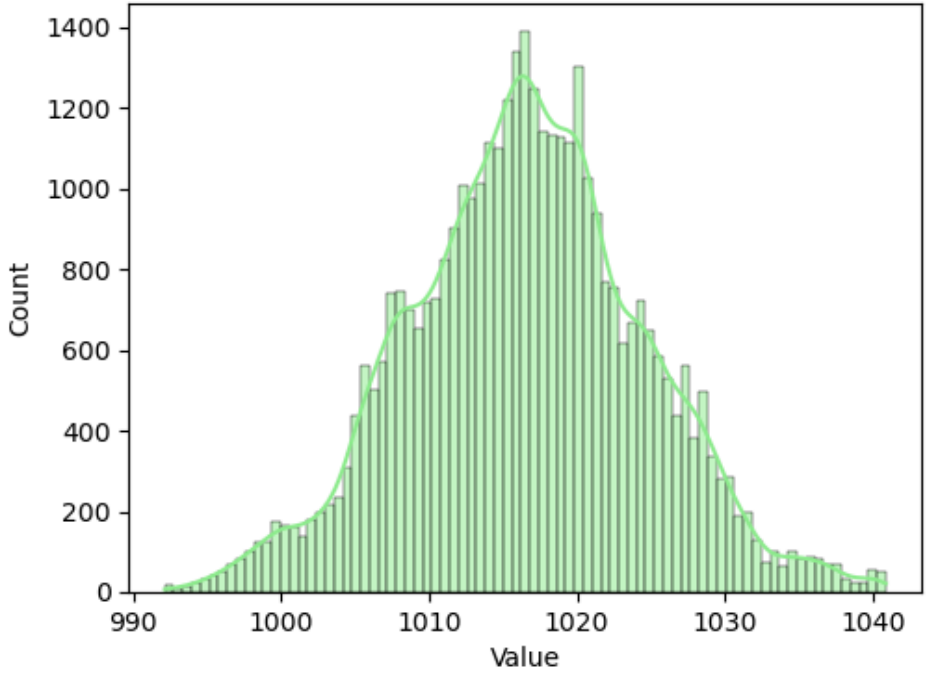




AtmosphericPressure — Boxplot

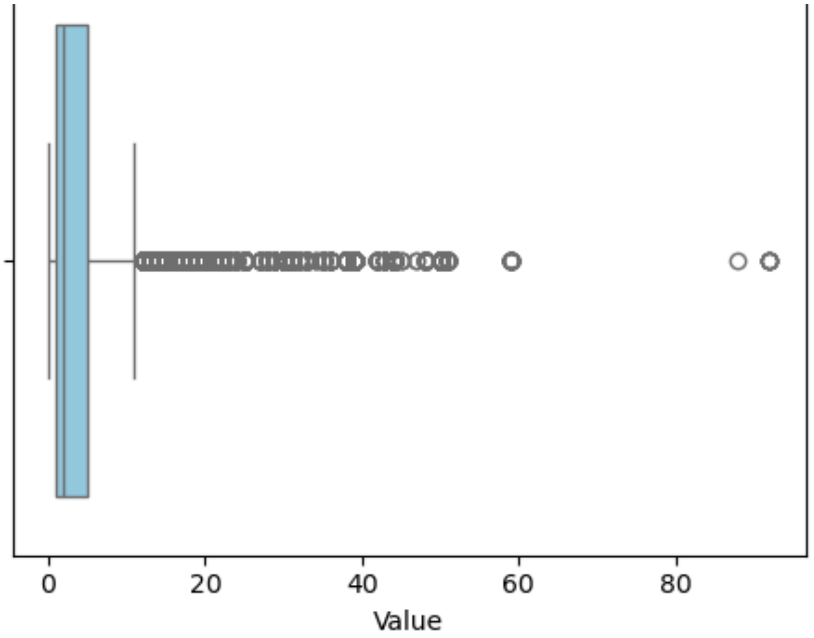


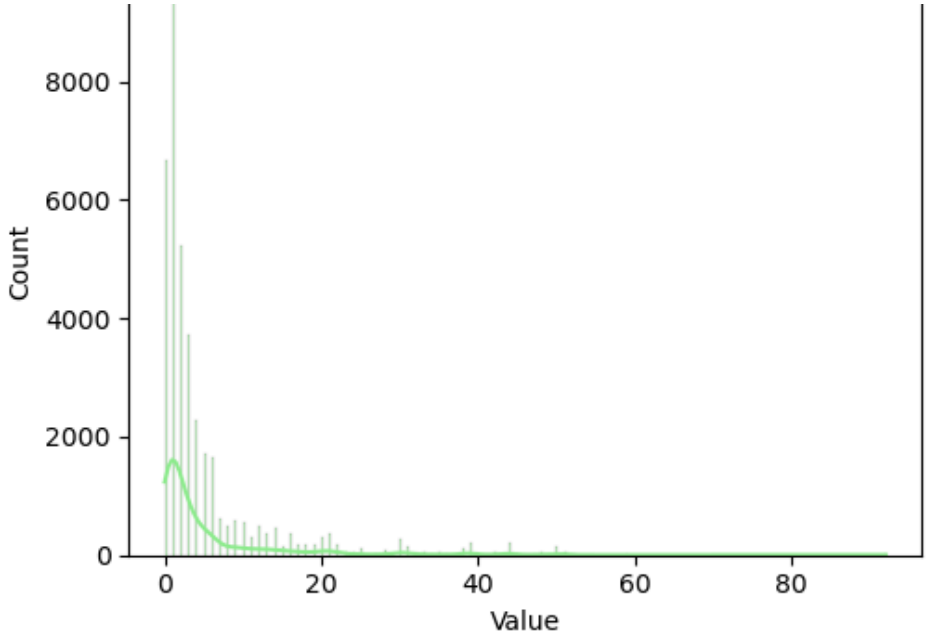
AtmosphericPressure — Distribution



PM25 — Boxplot

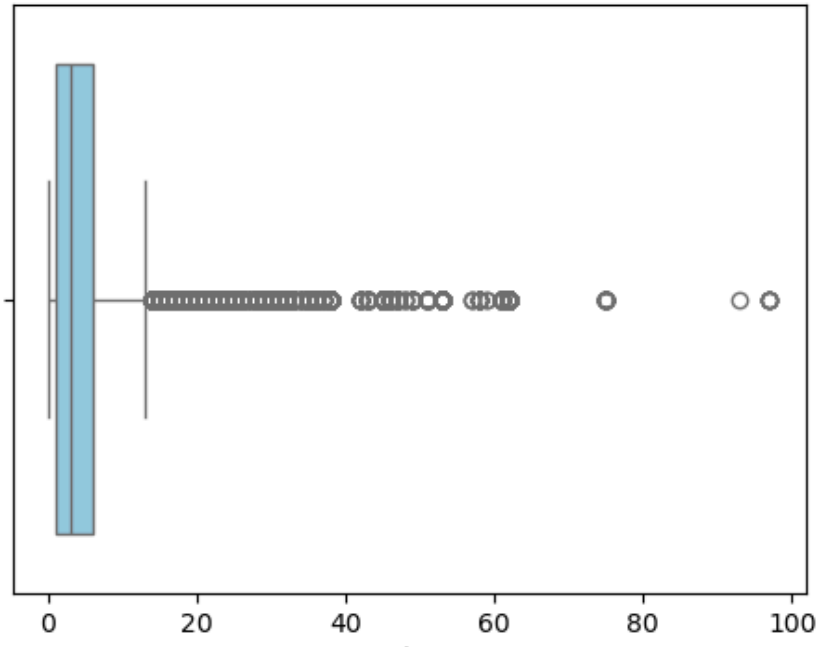
PM25 — Distribution

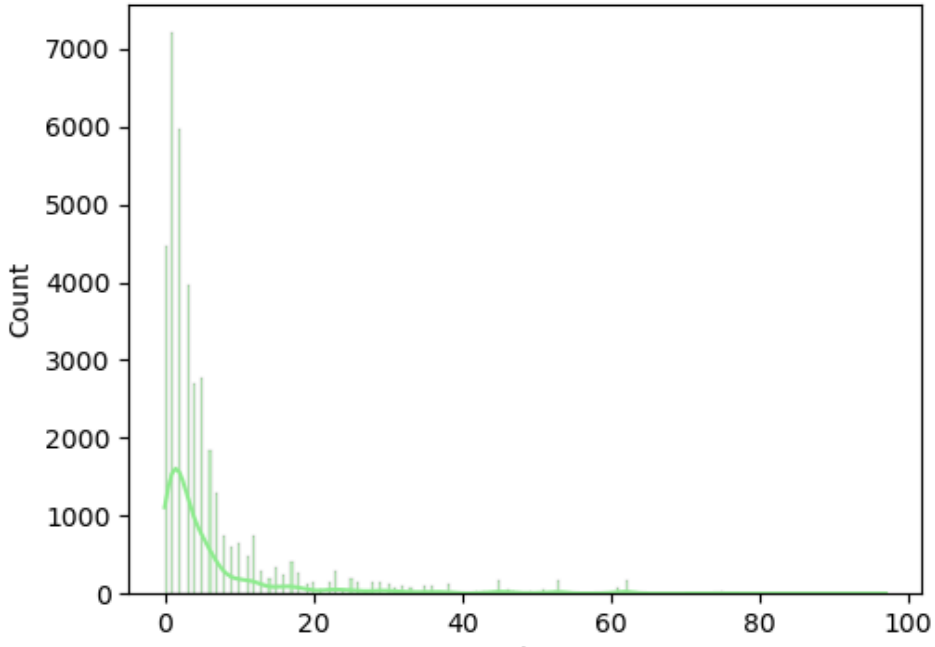


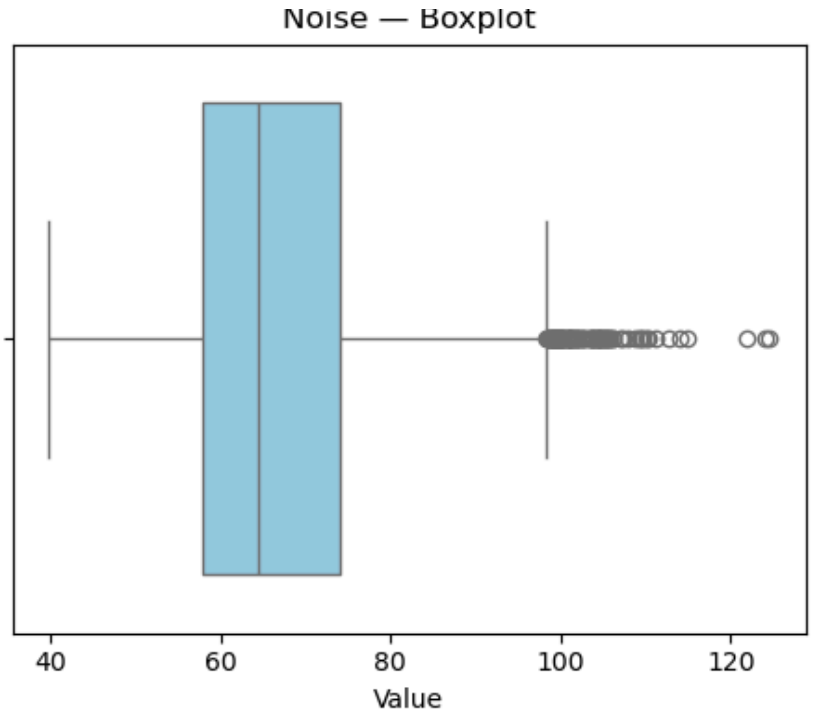


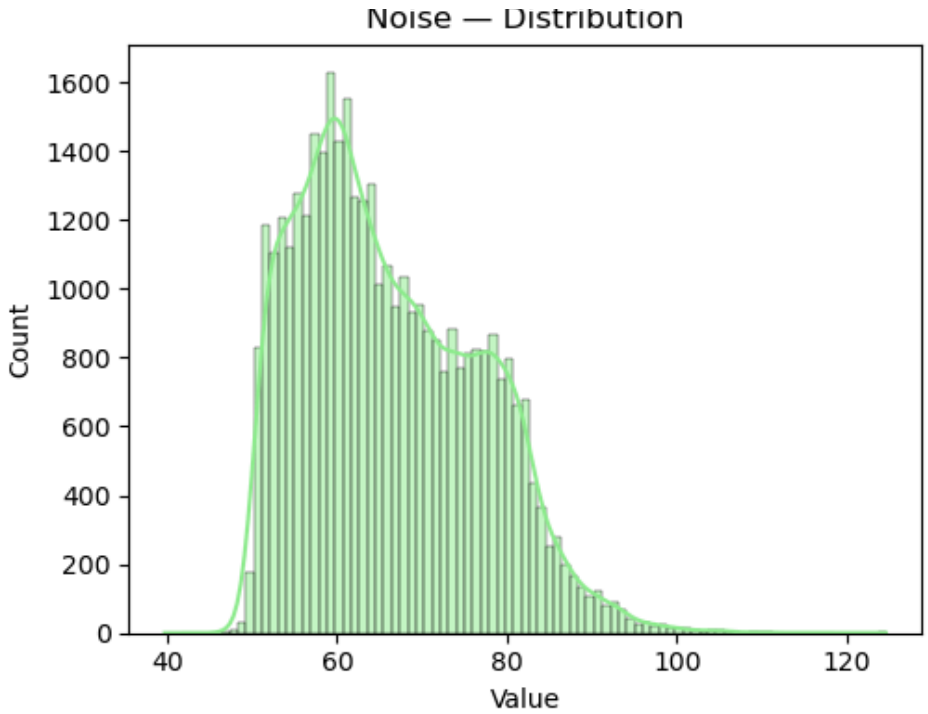
PM10 — Boxplot

PM10 — Distribution

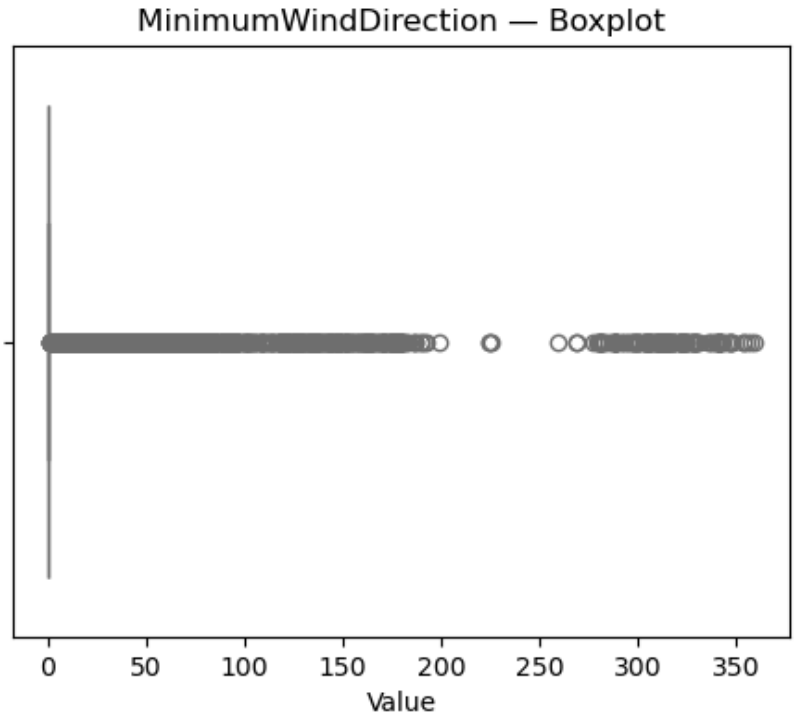


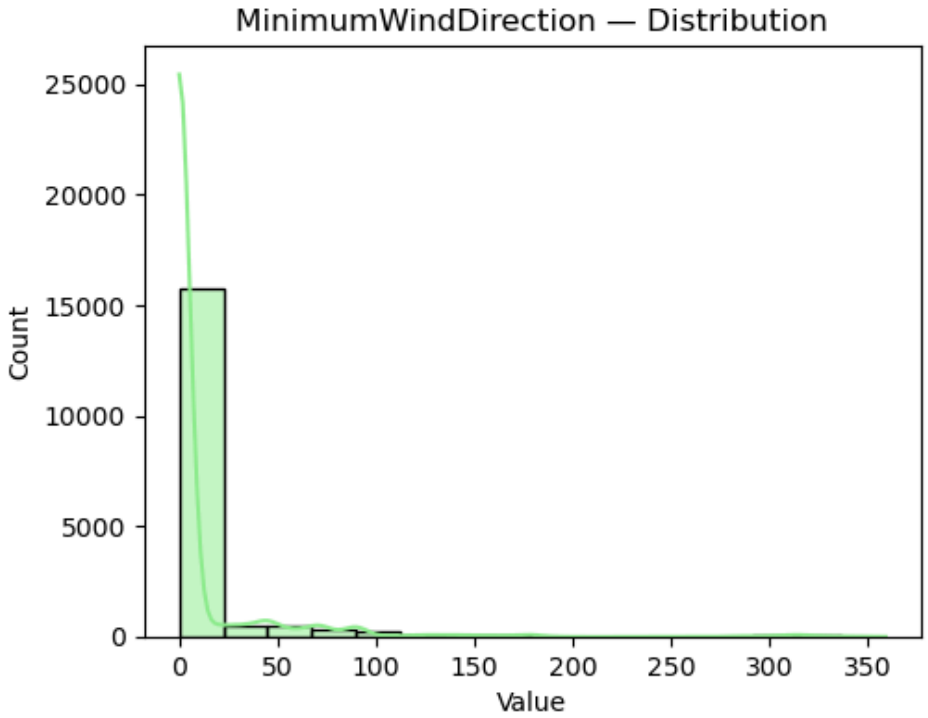


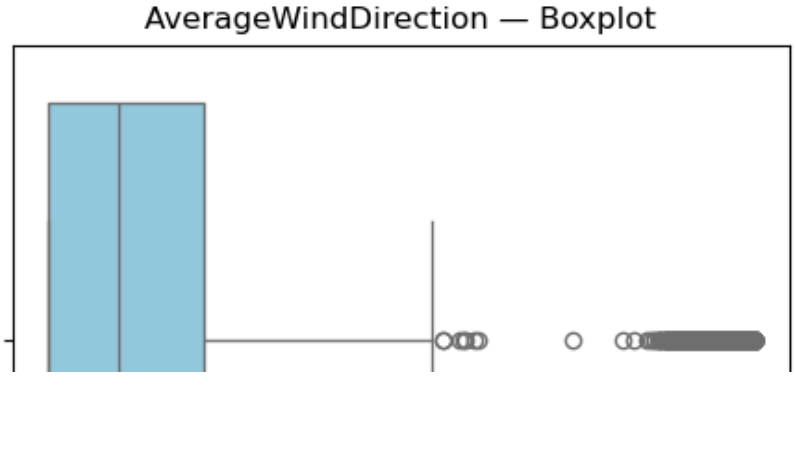

Noise — Boxplot
40
60
80
100
120
Value

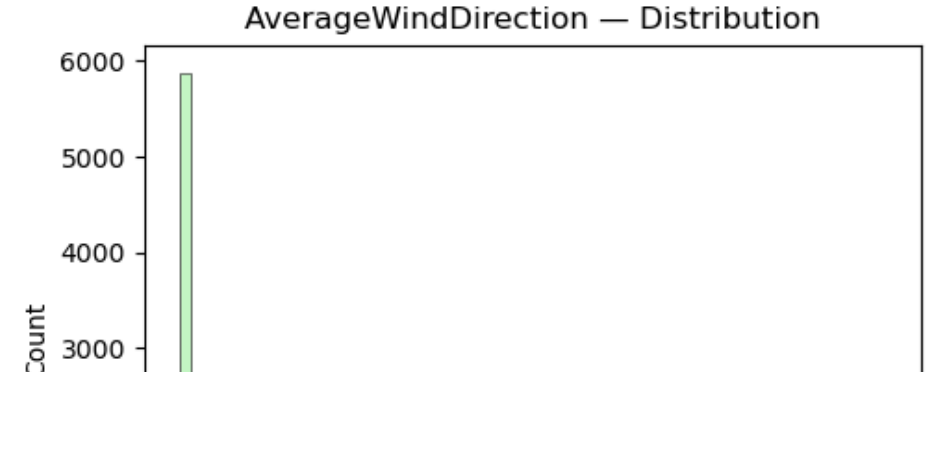

Noise — Distribution
1600
1400
1200
1000
800
600
400
200
0
Count
40
60
80
100
120
Value

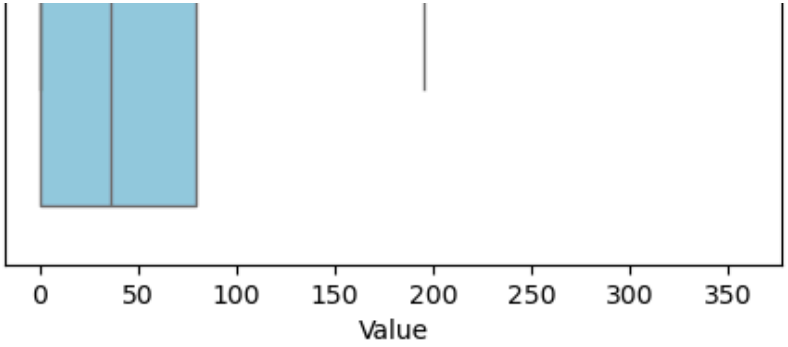
Device: aws5-0999

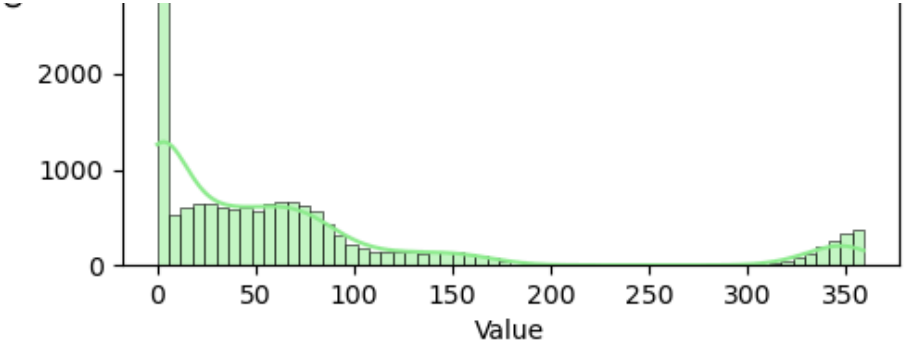




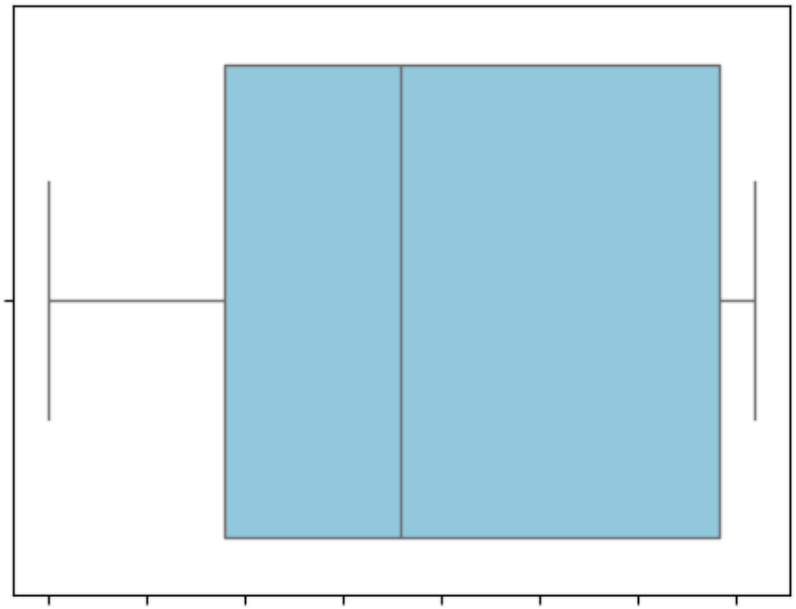




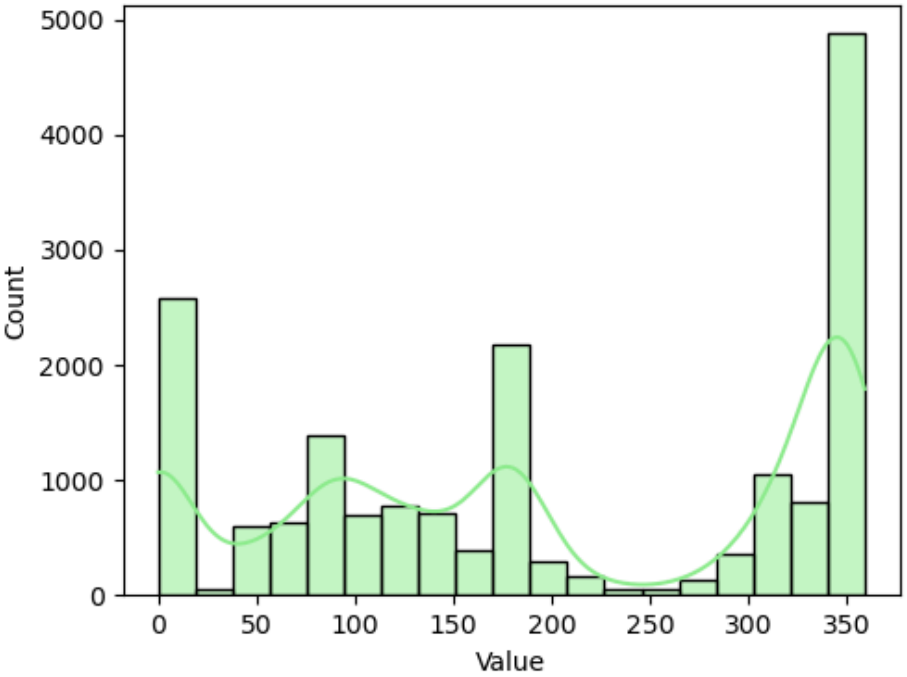




MaximumWindDirection — Boxplot

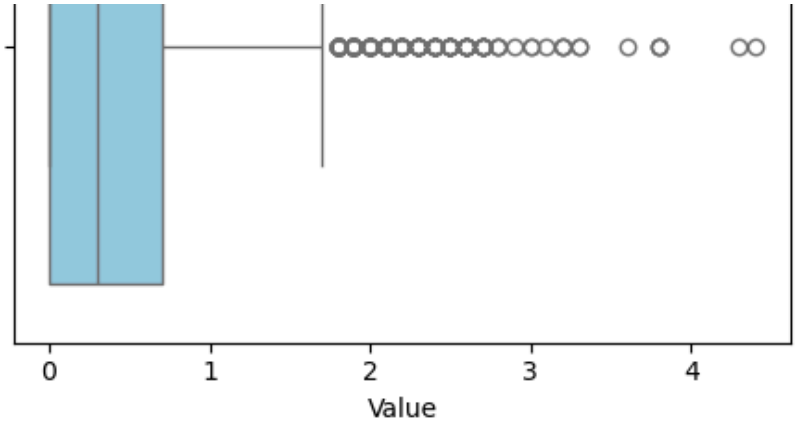


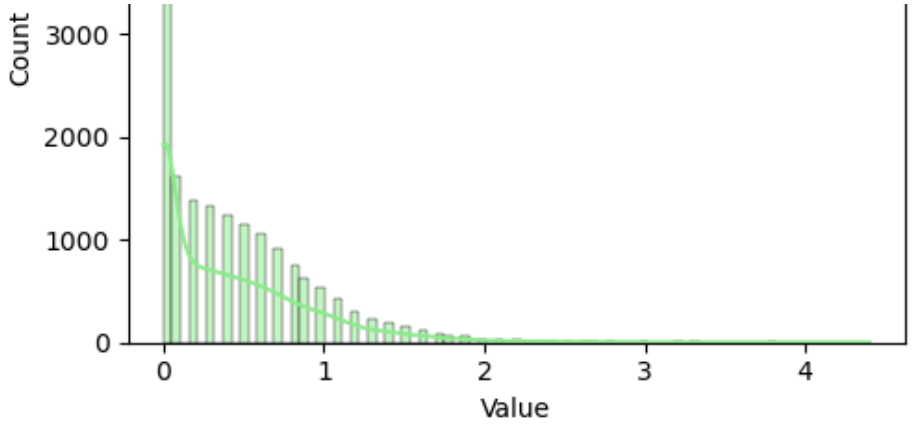
MaximumWindDirection — Distribution



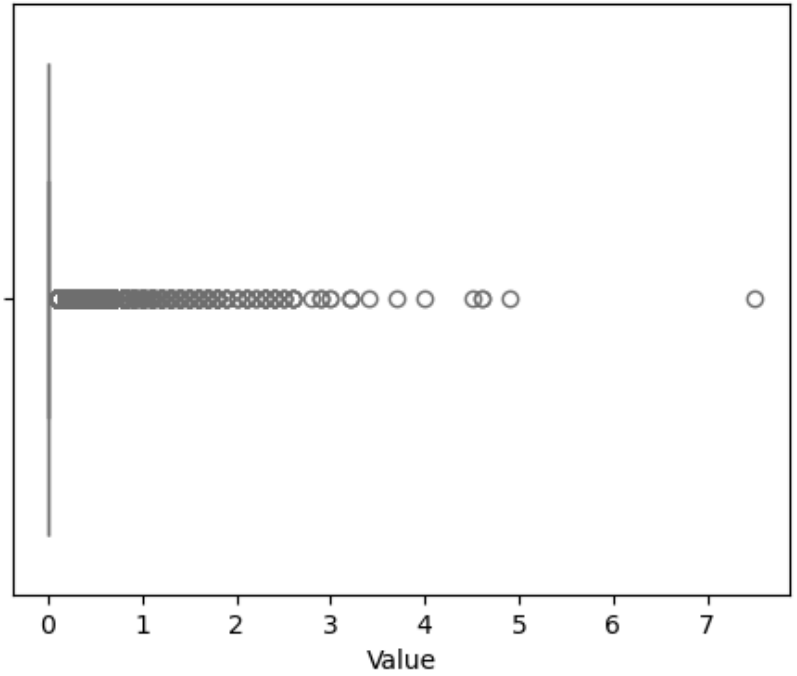
MinimumWindSpeed — Boxplot

MinimumWindSpeed — Distribution

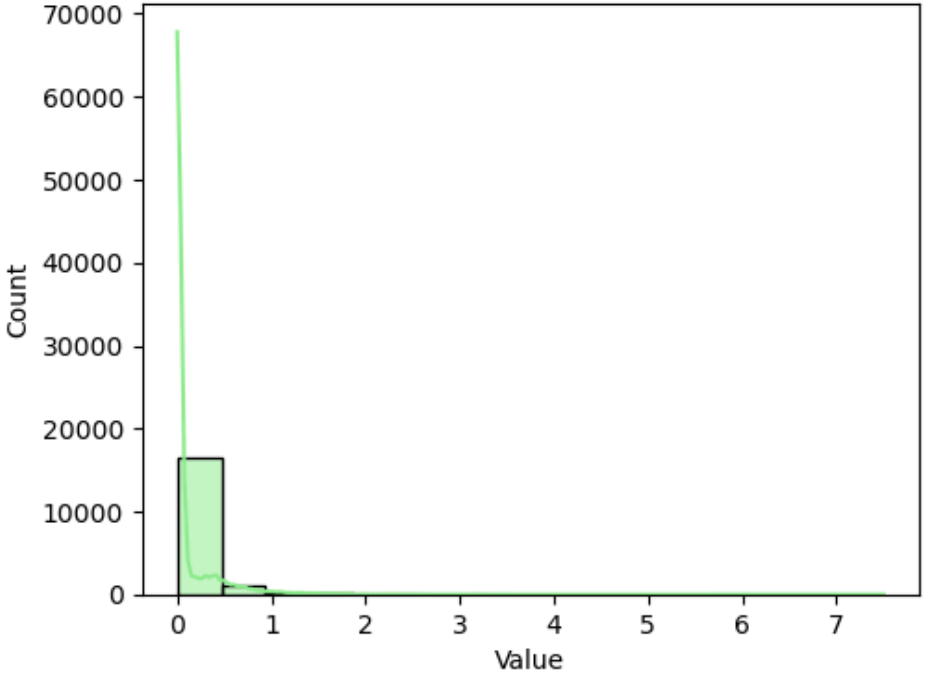




AverageWindSpeed — Boxplot

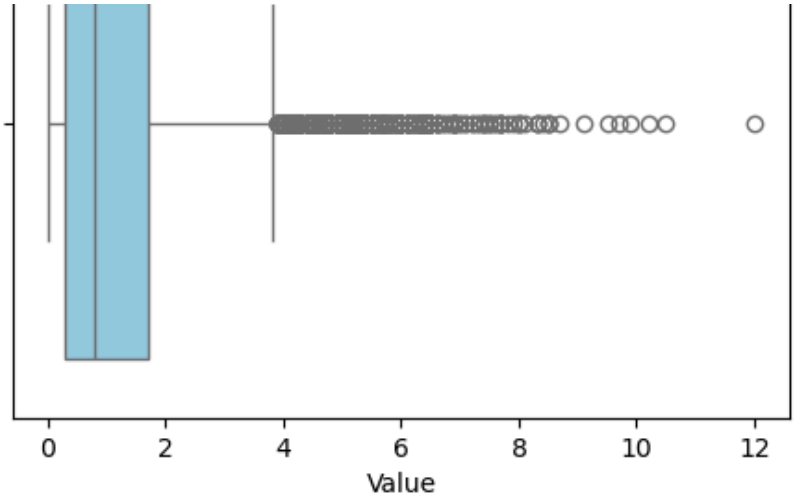


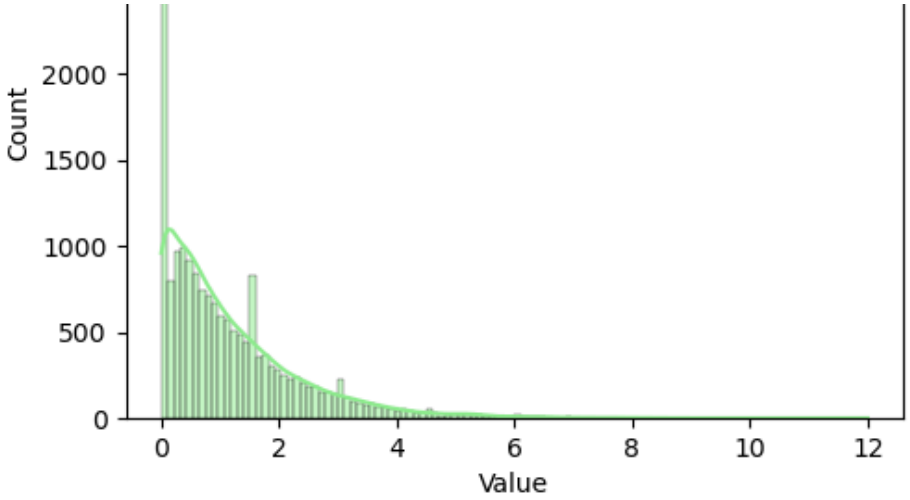
AverageWindSpeed — Distribution



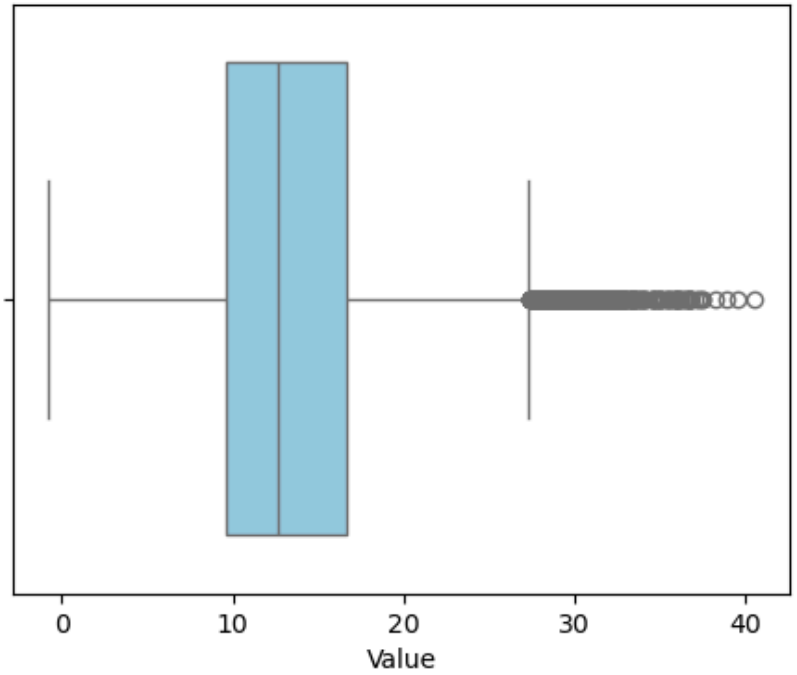
GustWindSpeed — Boxplot

GustWindSpeed — Distribution

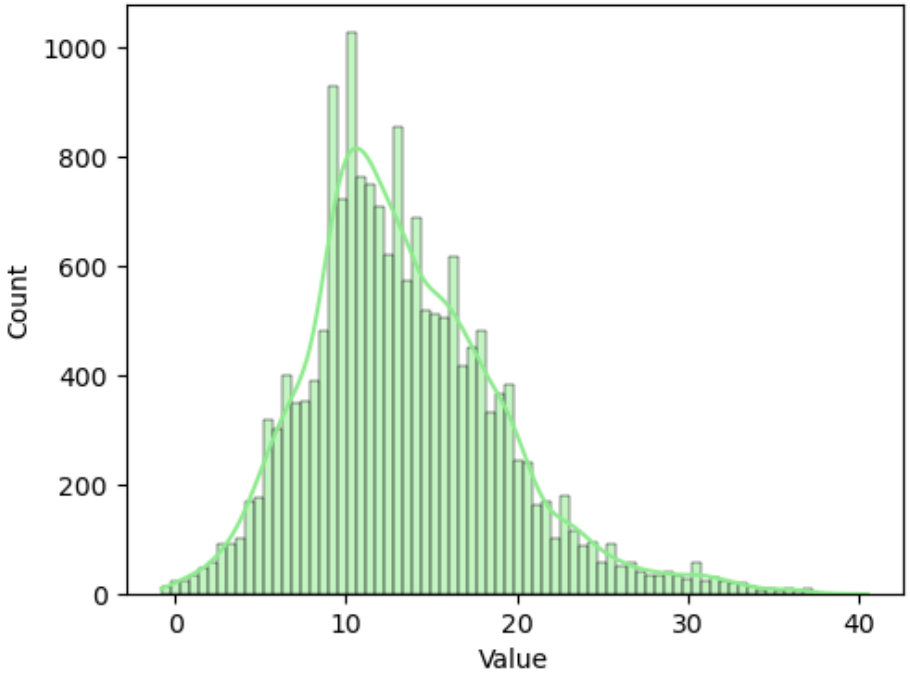




AirTemperature — Boxplot

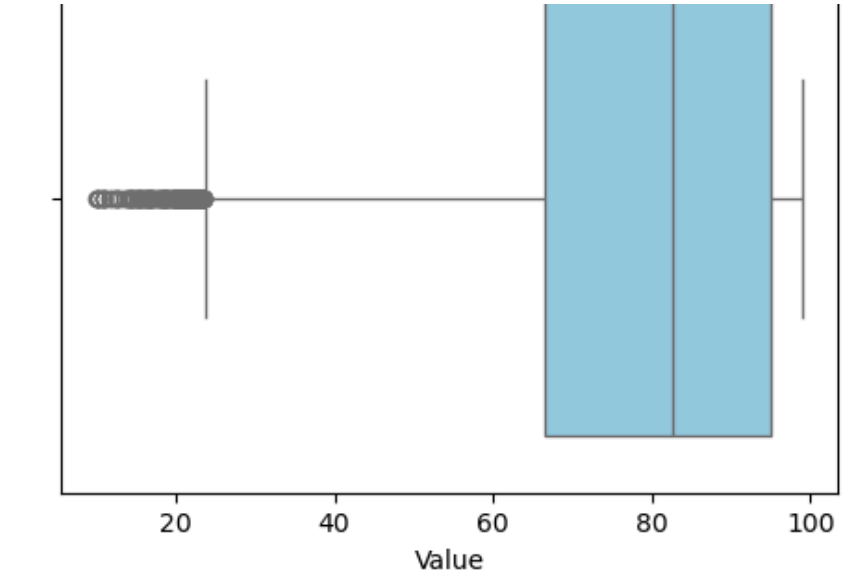


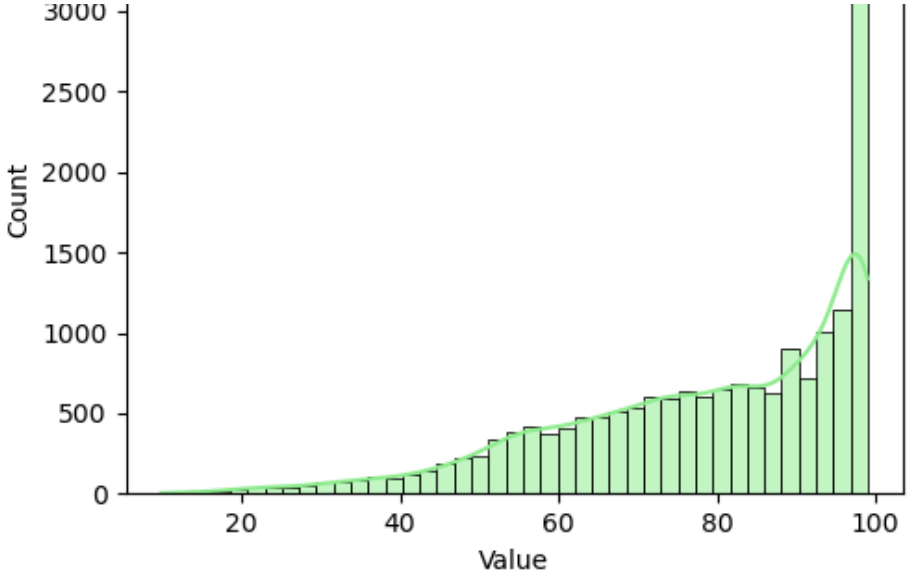
AirTemperature — Distribution



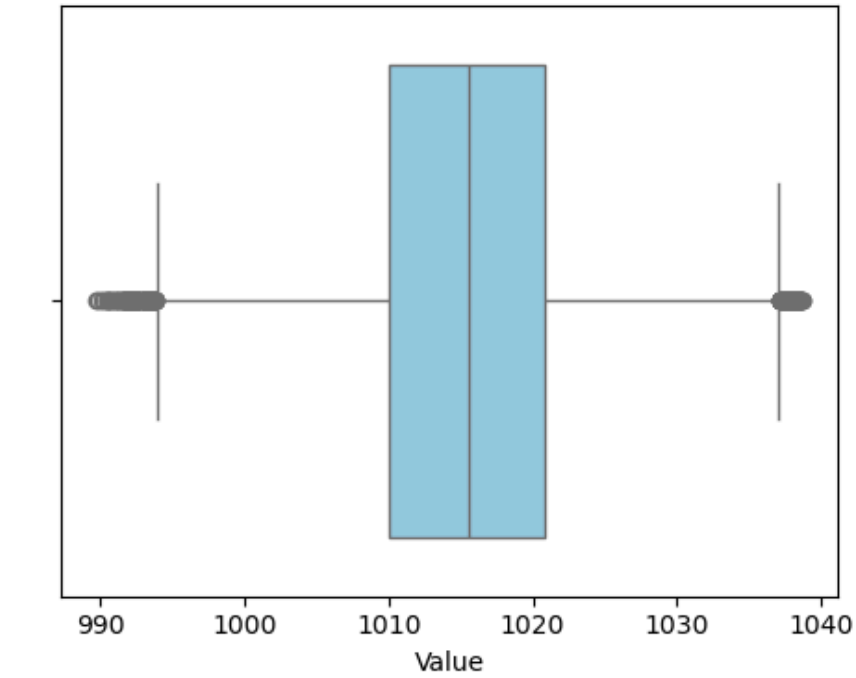
RelativeHumidity — Boxplot

RelativeHumidity — Distribution

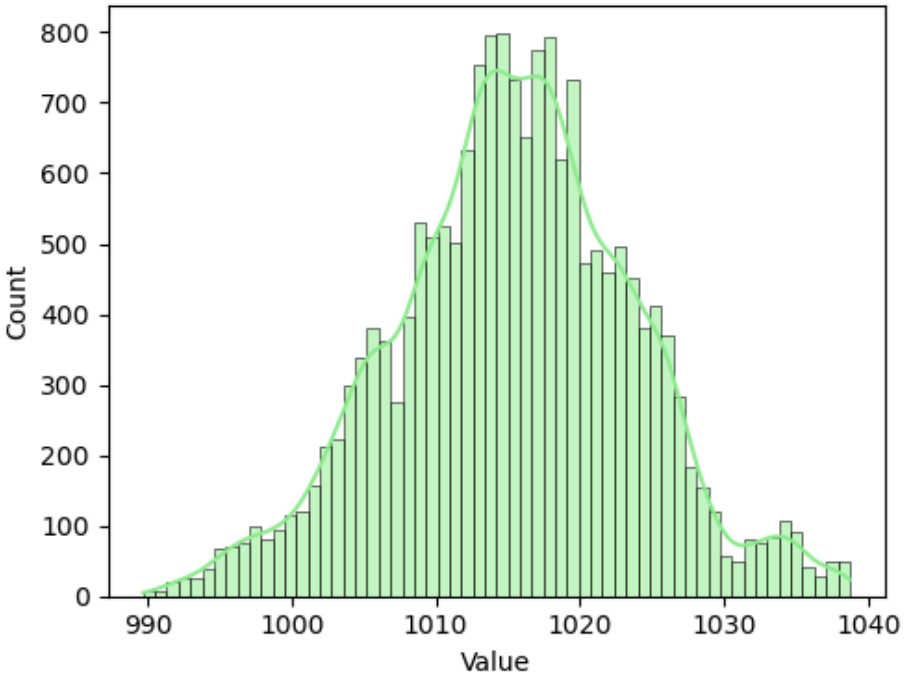




AtmosphericPressure — Boxplot

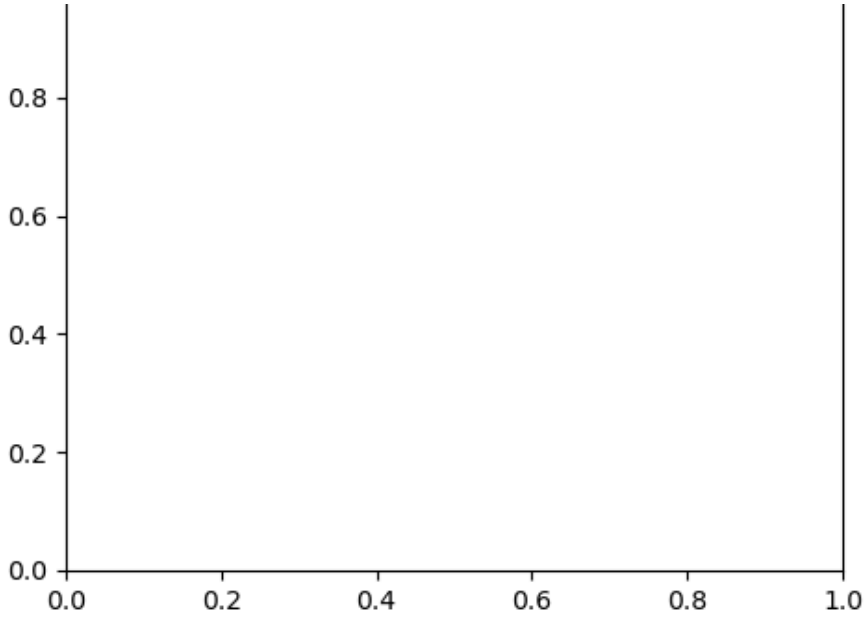


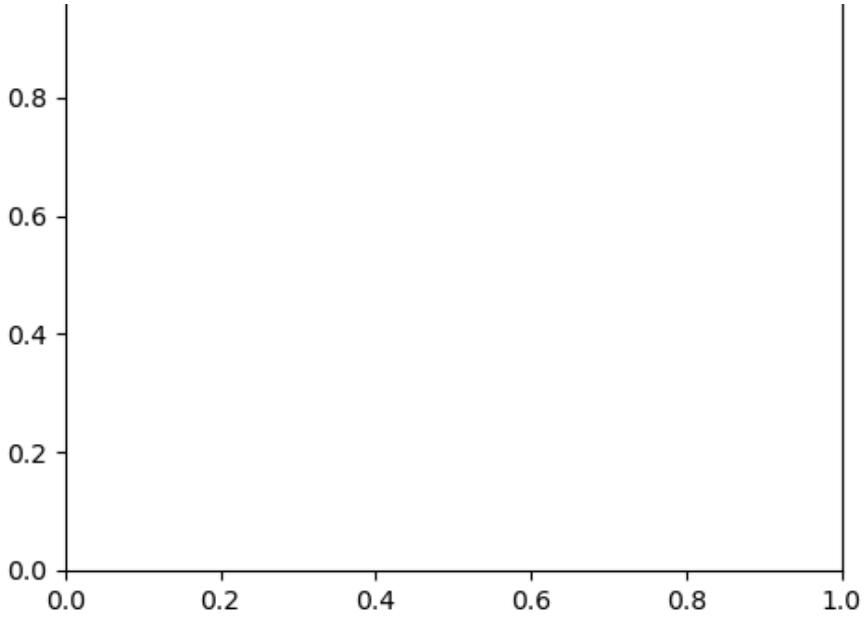
AtmosphericPressure — Distribution

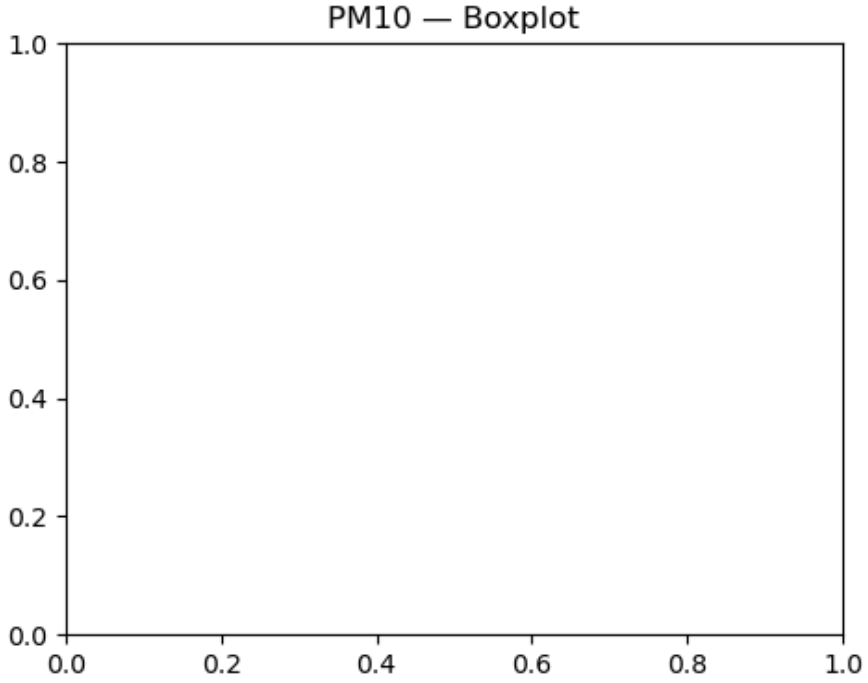


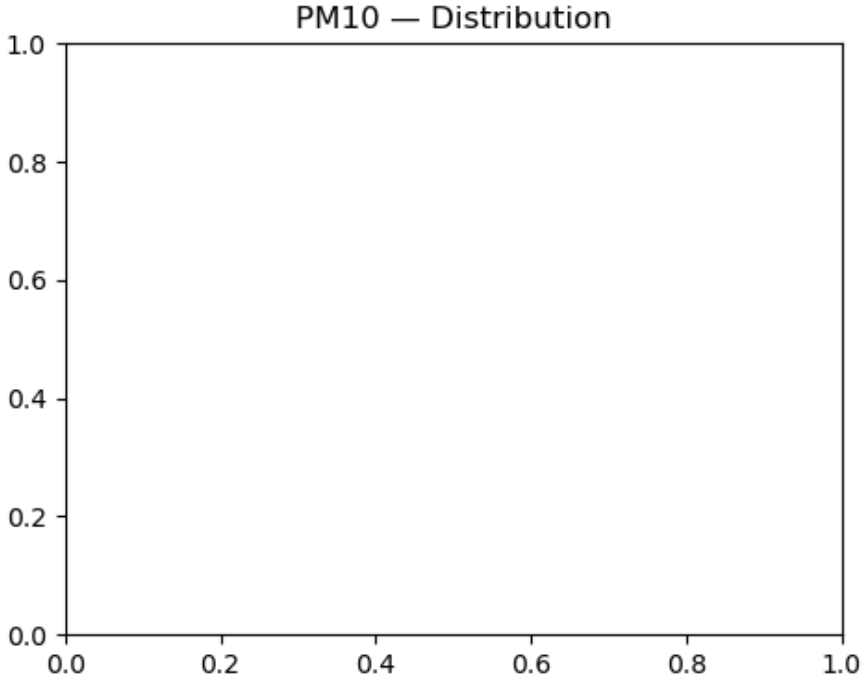
PM25 — Boxplot

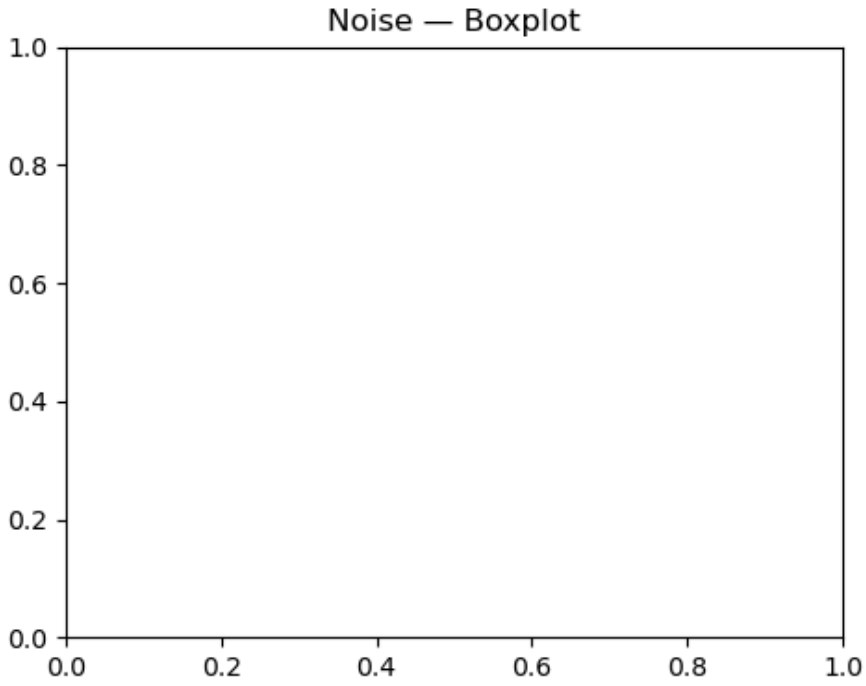
PM25 — Distribution

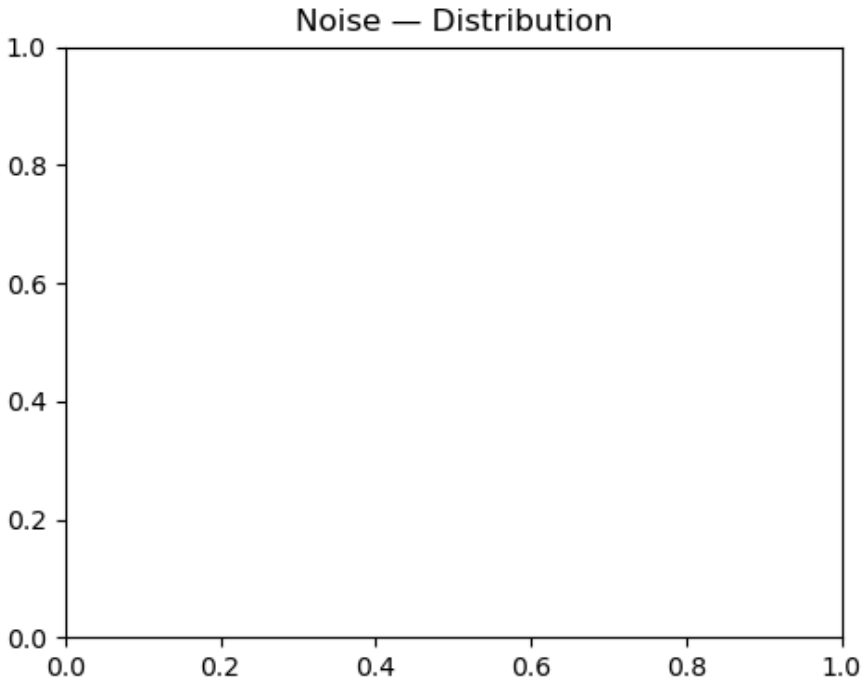

0.0
0.2
0.4
0.6
0.8
1.0


0.0
0.2
0.4
0.6
0.8
1.0

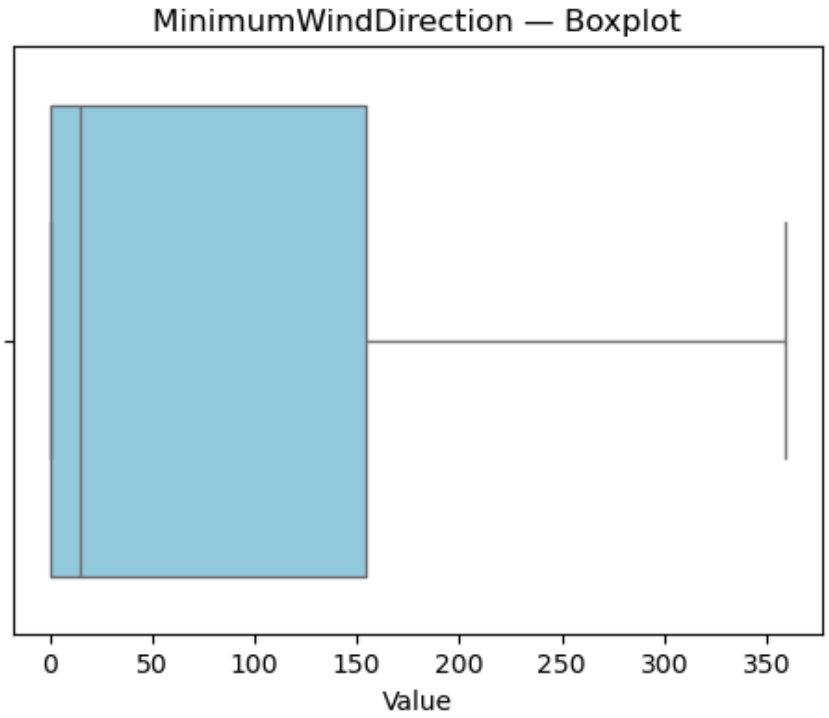

PM10 — Boxplot
0.0
0.2
0.4
0.6
0.8
1.0

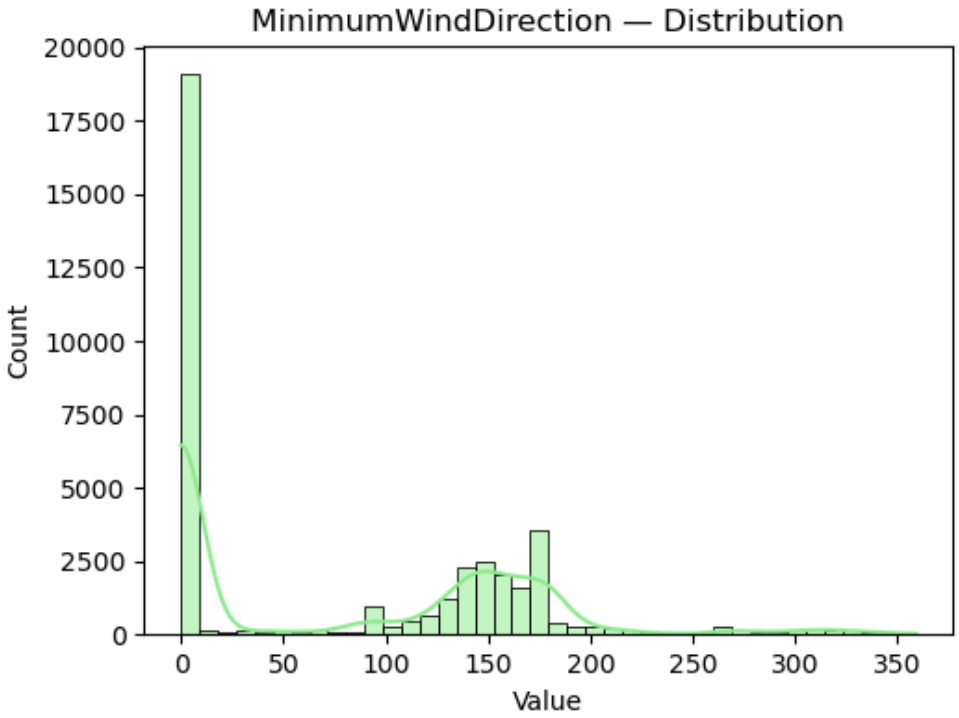

PM10 — Distribution
0.0
0.2
0.4
0.6
0.8
1.0

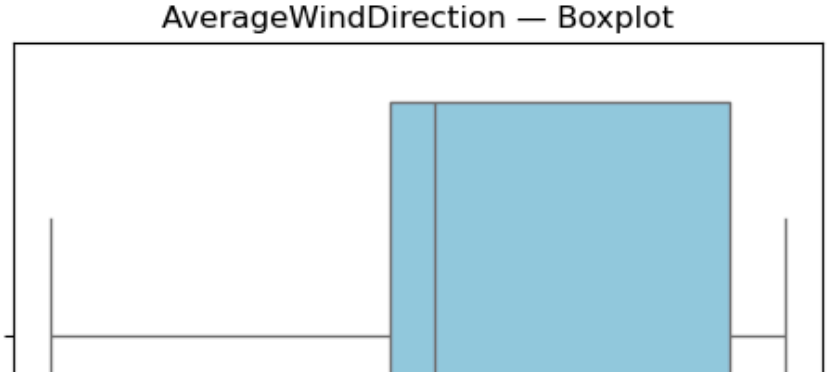

Noise — Boxplot
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

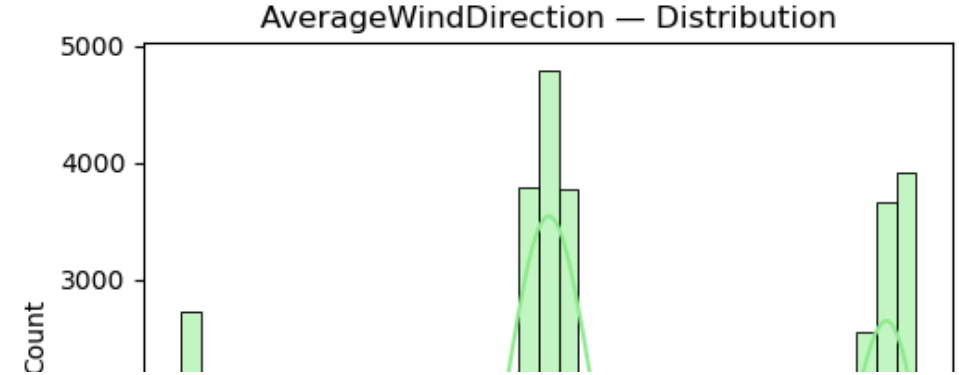

Noise — Distribution
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

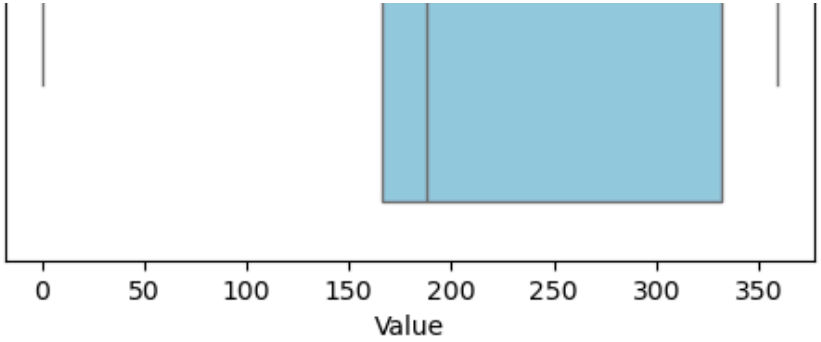
Device: ICTMicroclimate-03

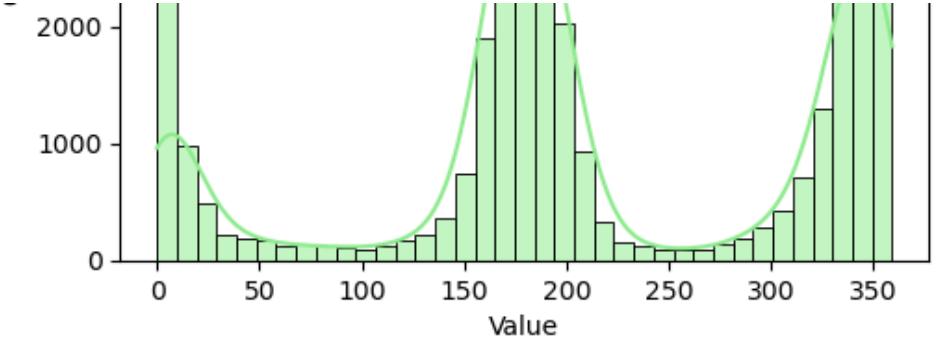




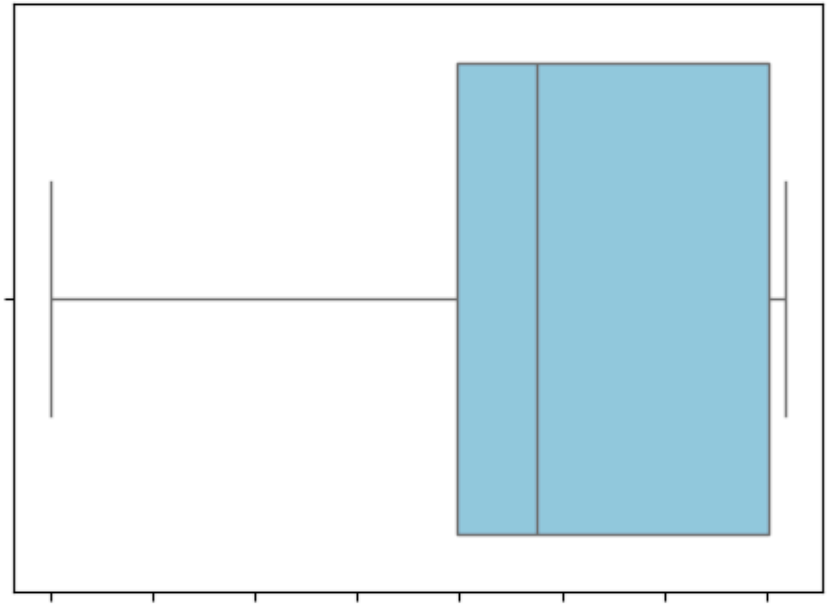




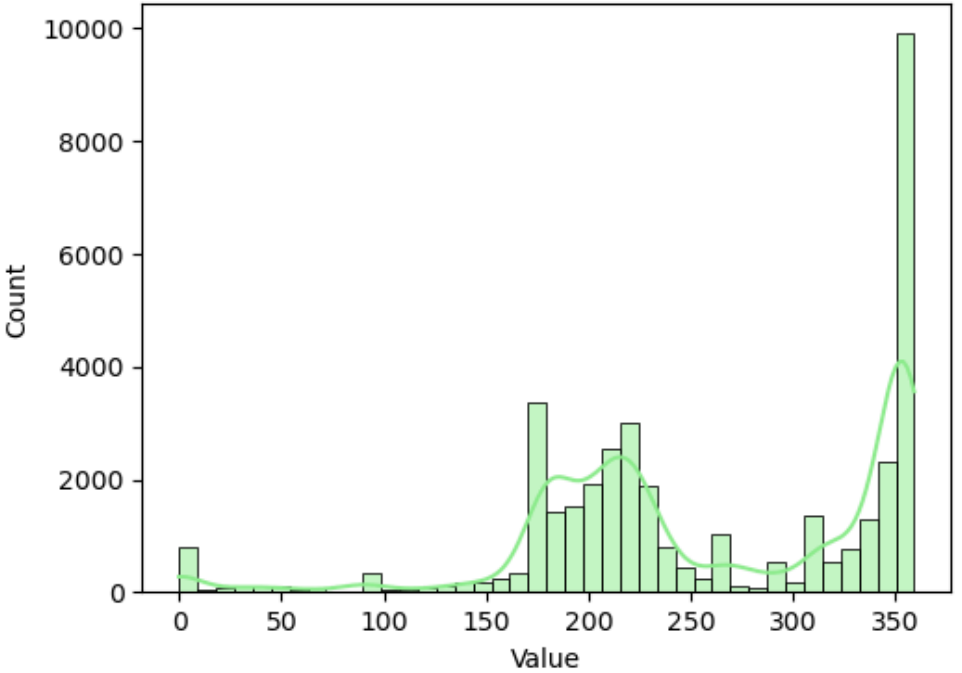




MaximumWindDirection — Boxplot

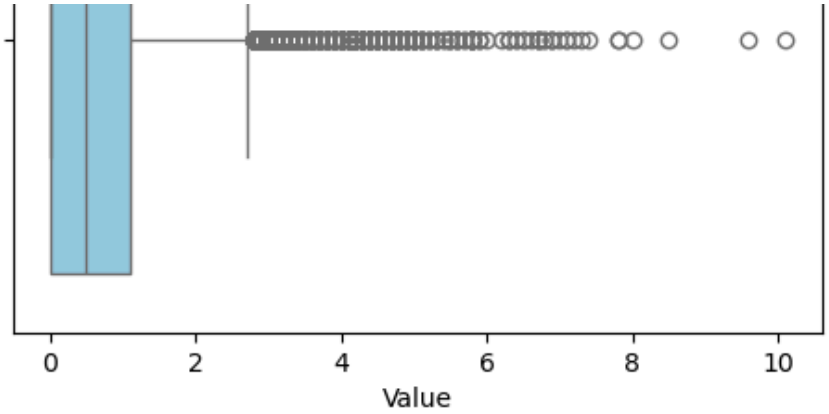


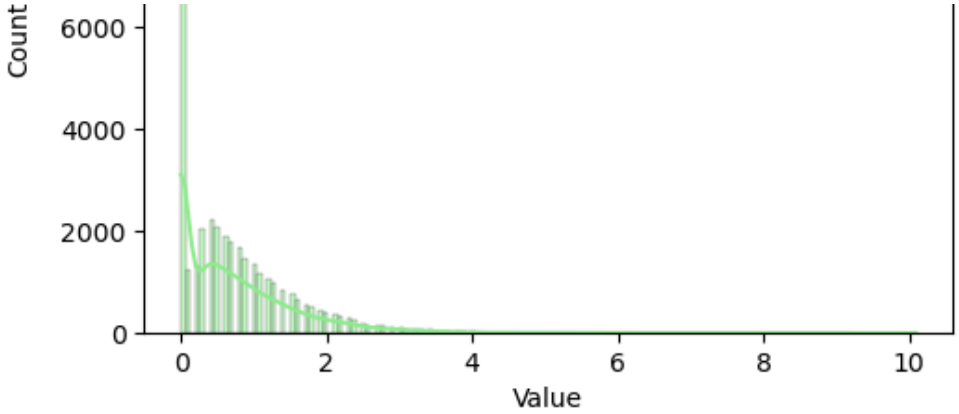
MaximumWindDirection — Distribution



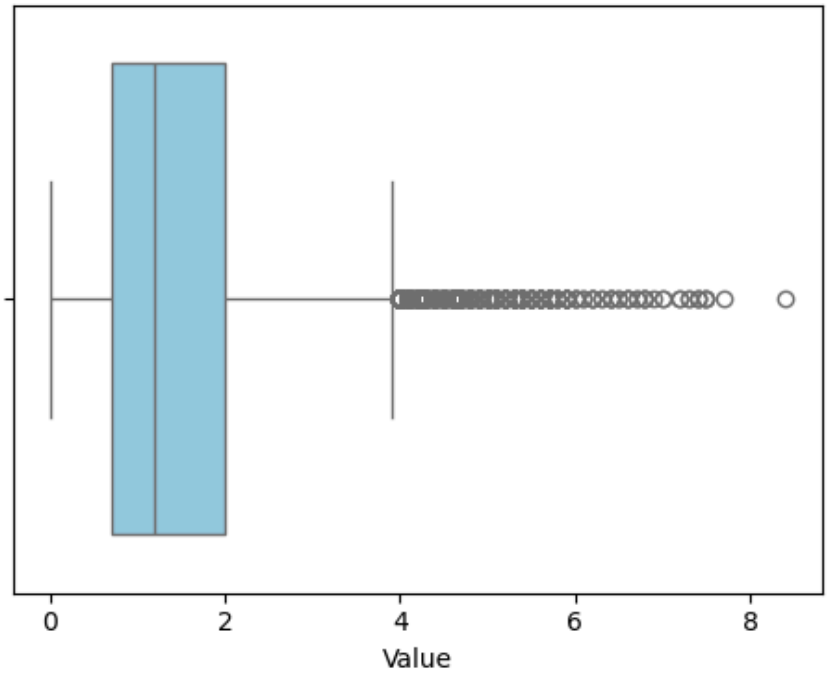
MinimumWindSpeed — Boxplot

MinimumWindSpeed — Distribution

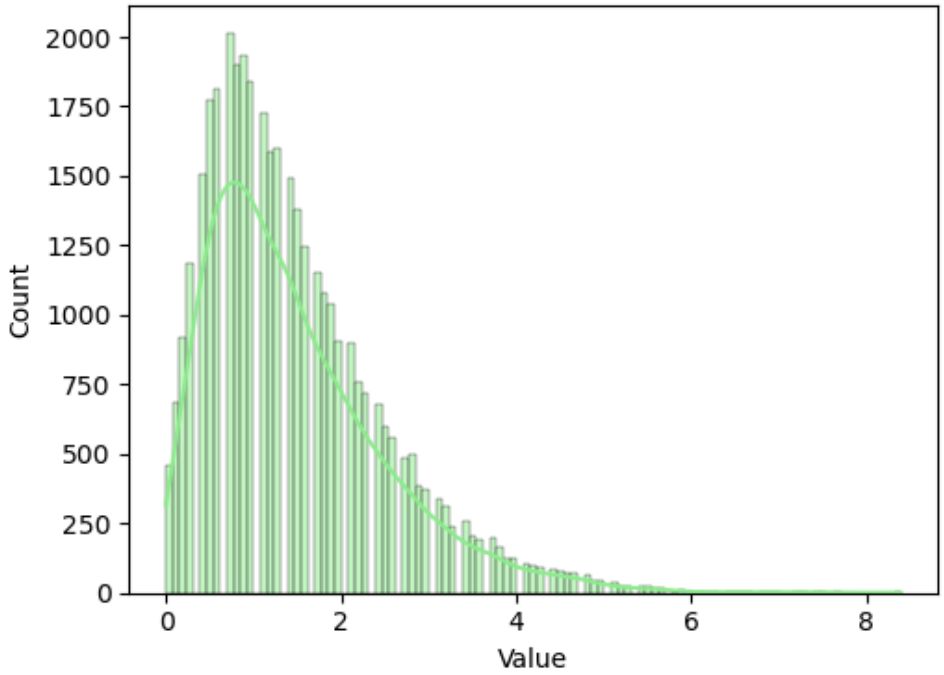




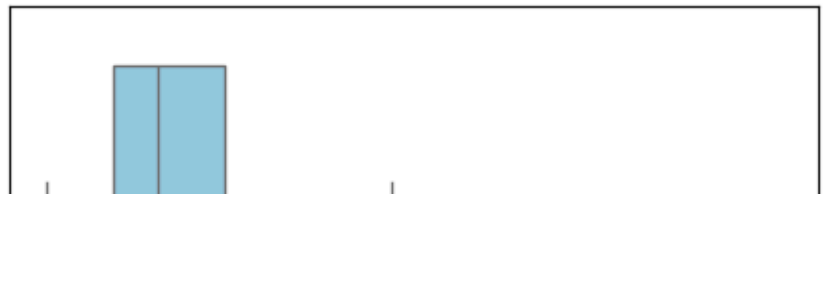
AverageWindSpeed — Boxplot



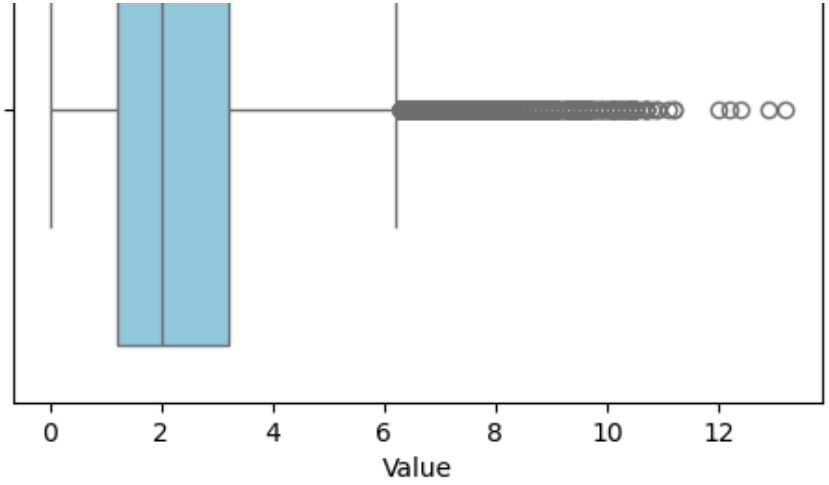
AverageWindSpeed — Distribution

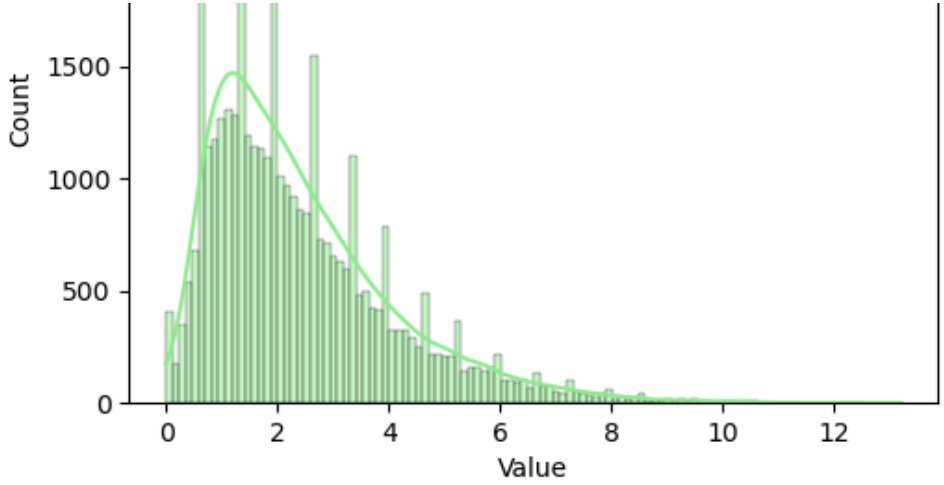


GustWindSpeed — Boxplot

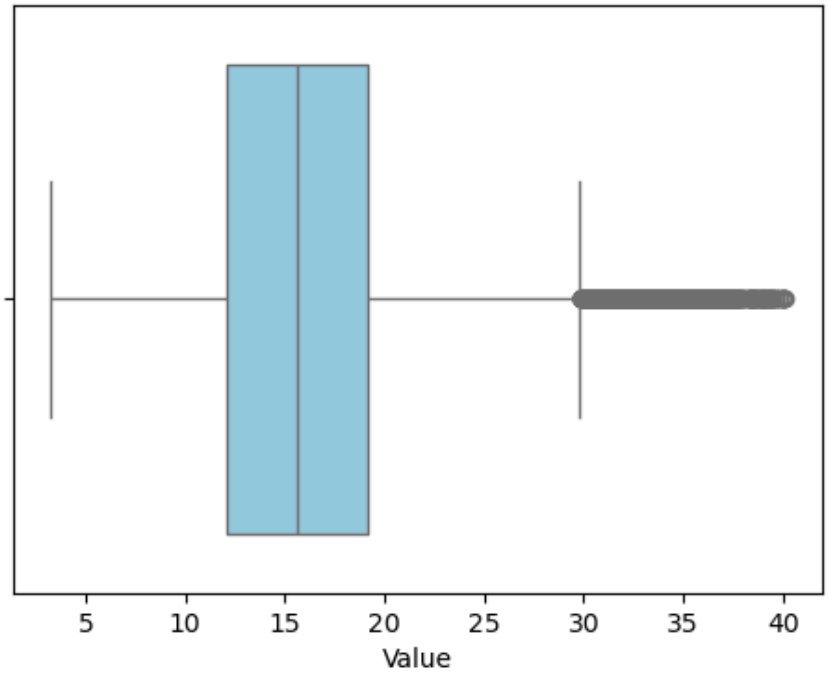


GustWindSpeed — Distribution

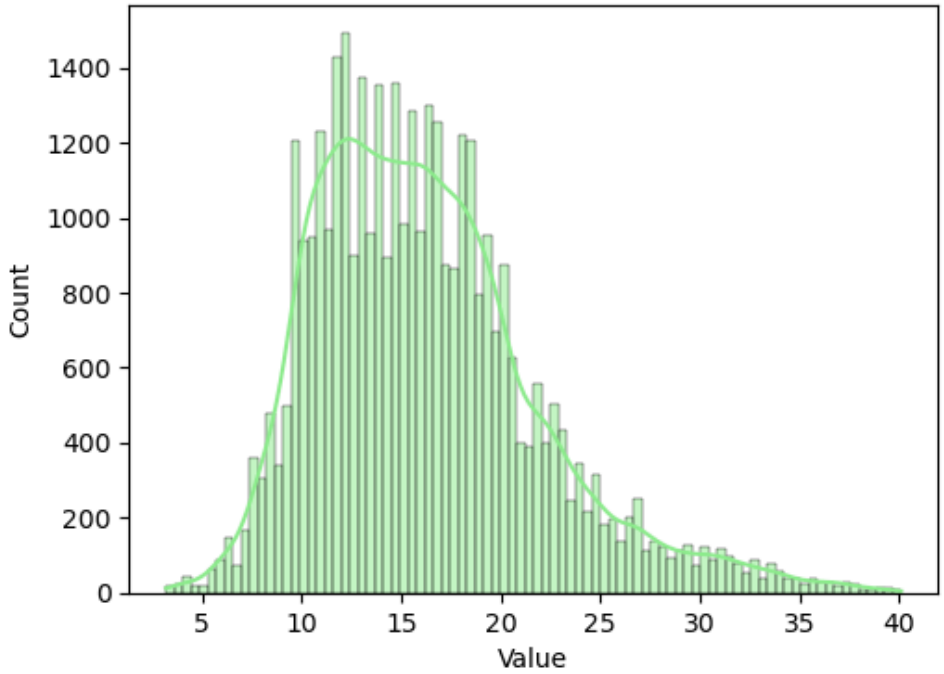




AirTemperature — Boxplot

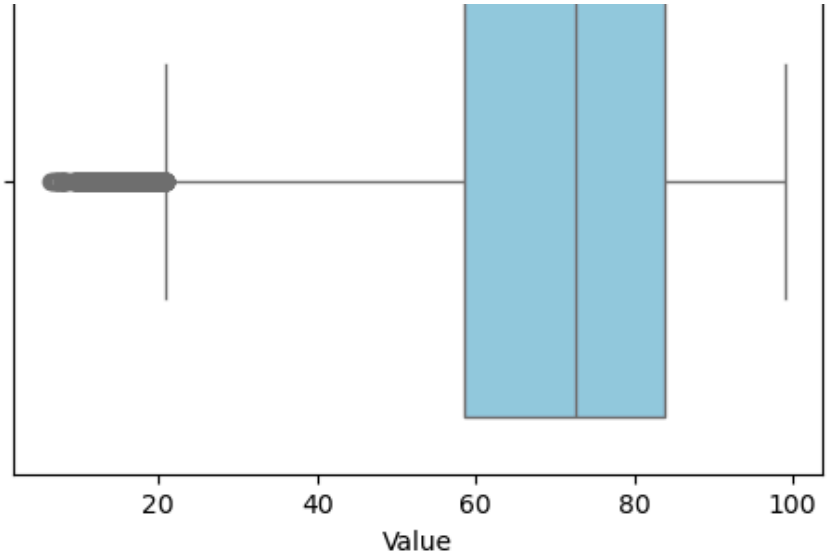


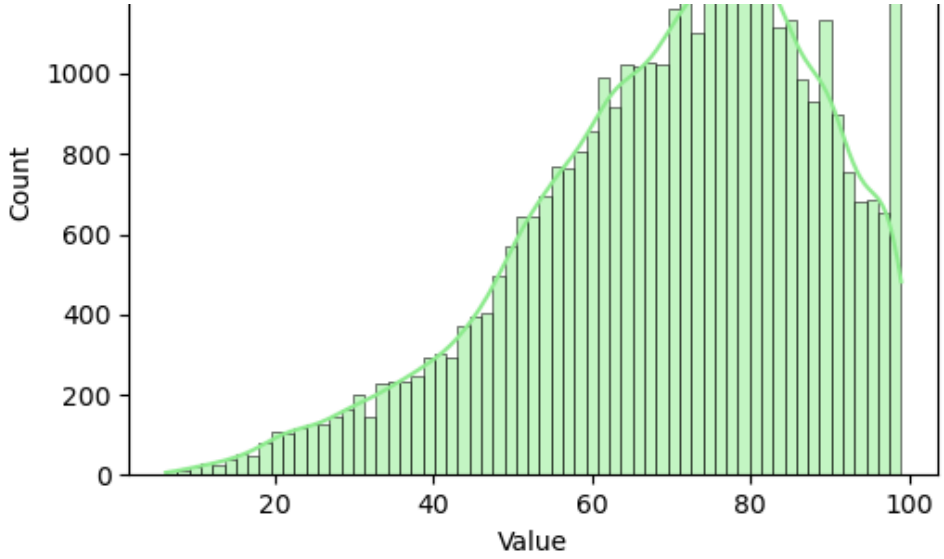
AirTemperature — Distribution



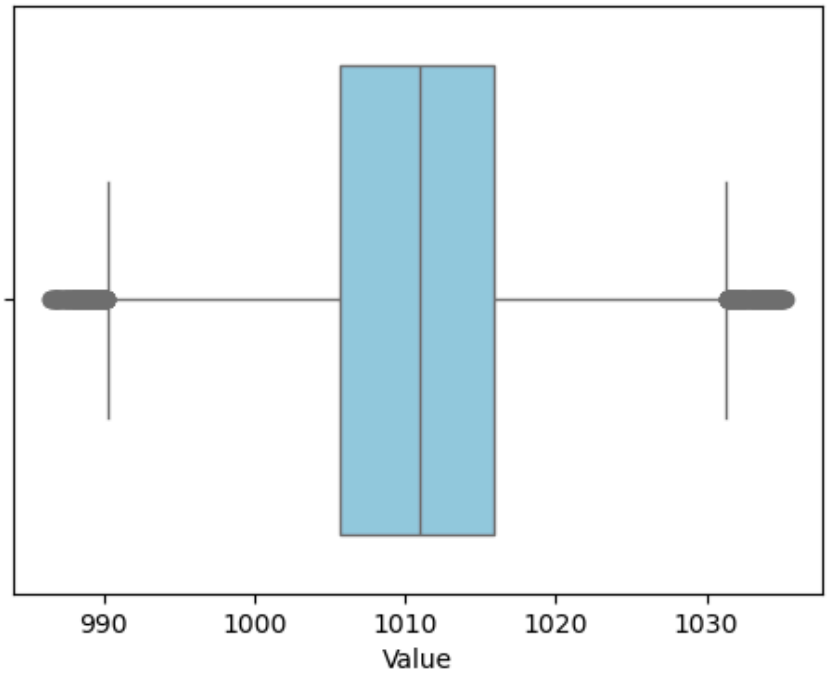
RelativeHumidity — Boxplot

RelativeHumidity — Distribution

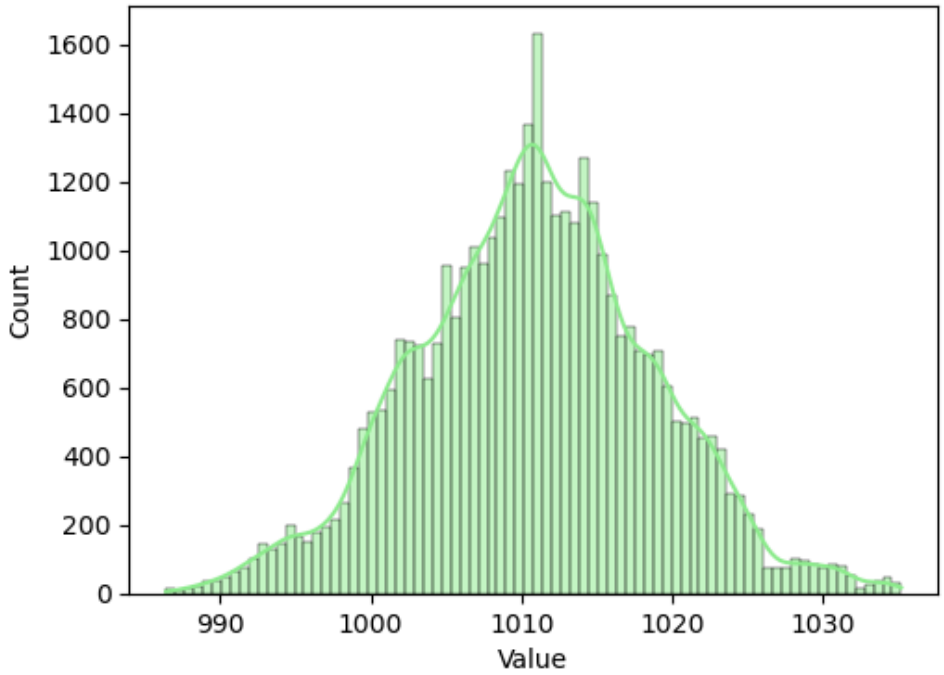




AtmosphericPressure — Boxplot

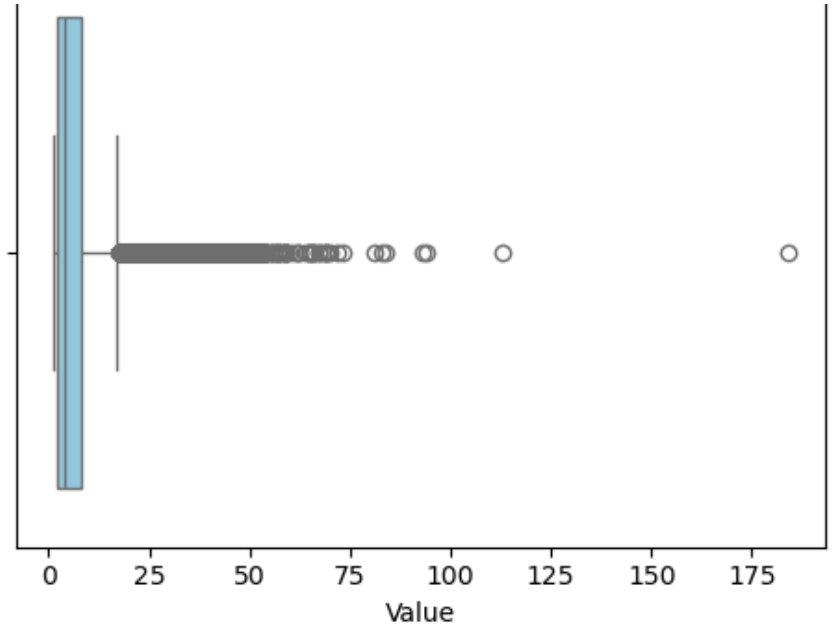


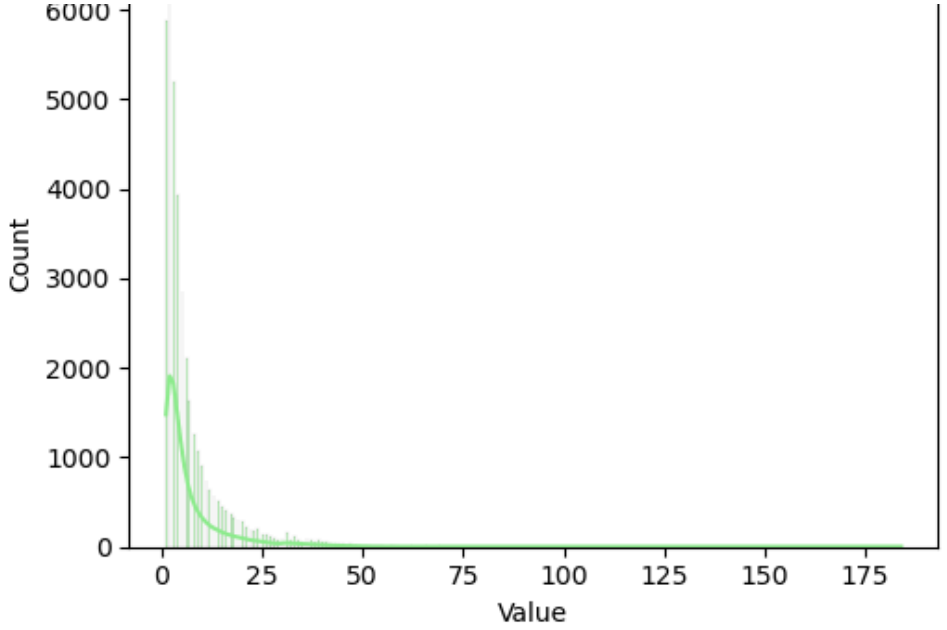
AtmosphericPressure — Distribution



PM25 — Boxplot

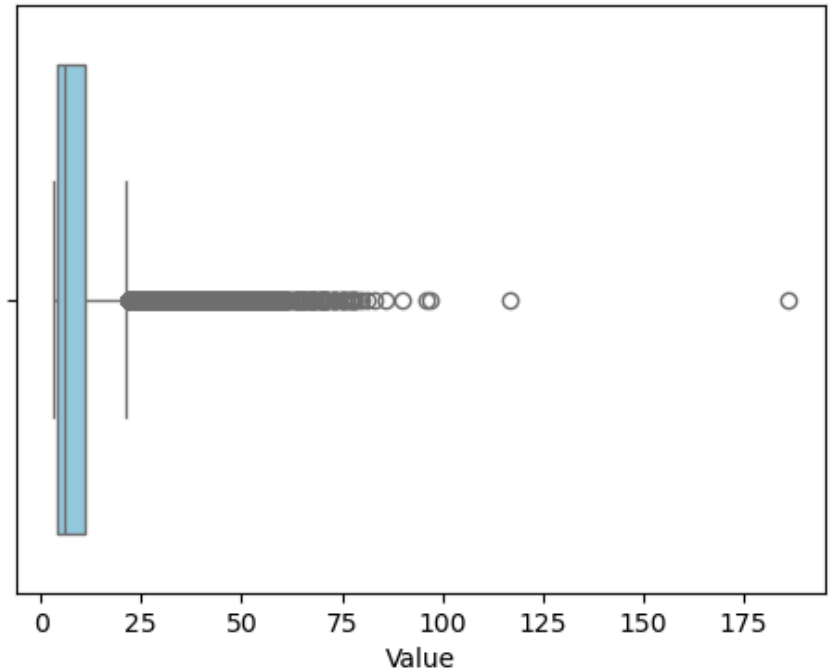
PM25 — Distribution

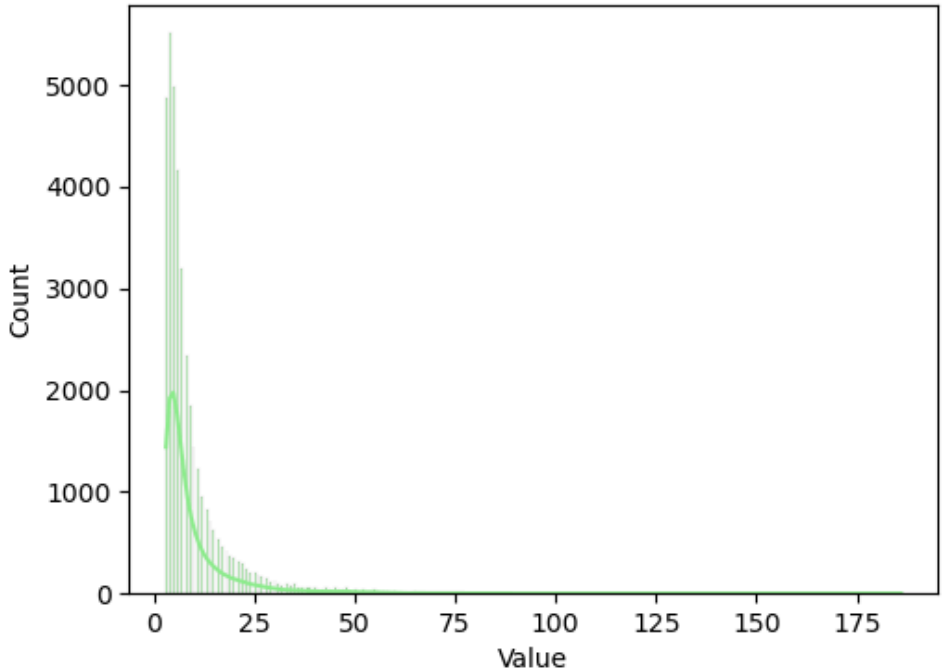




PM10 — Boxplot

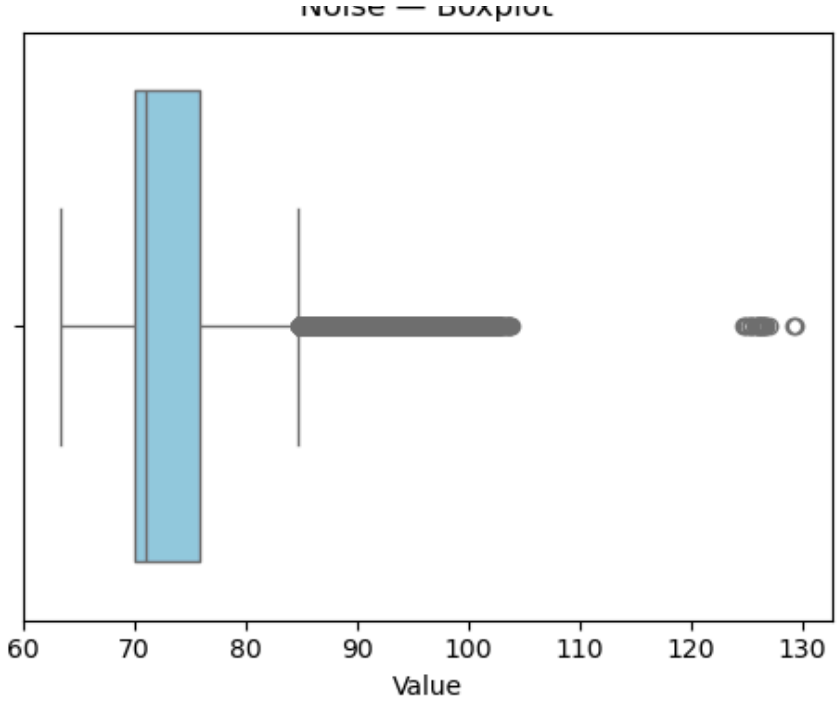
PM10 — Distribution

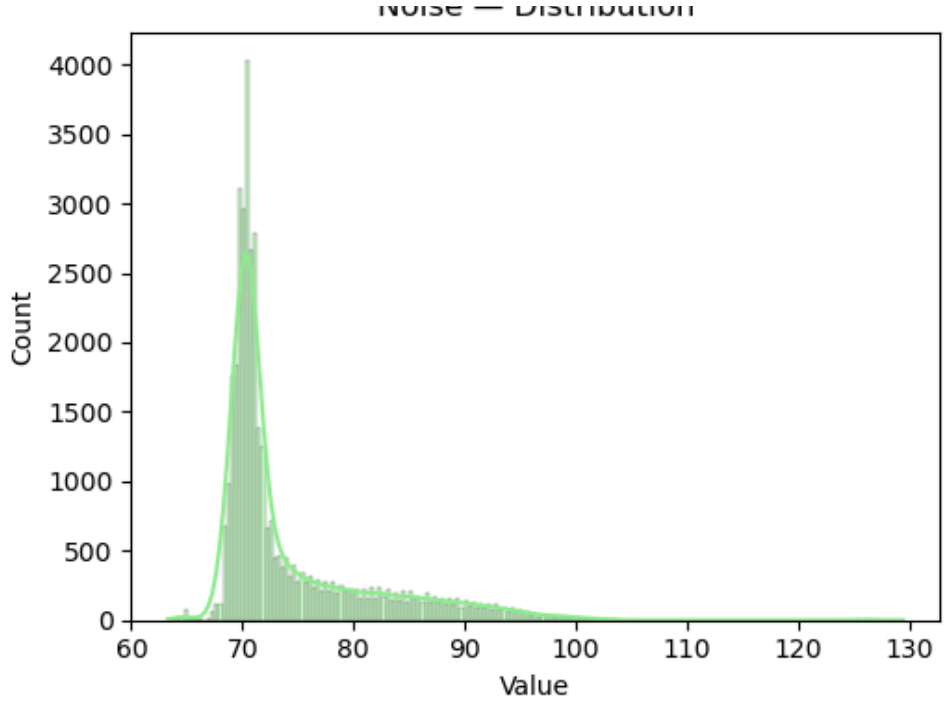




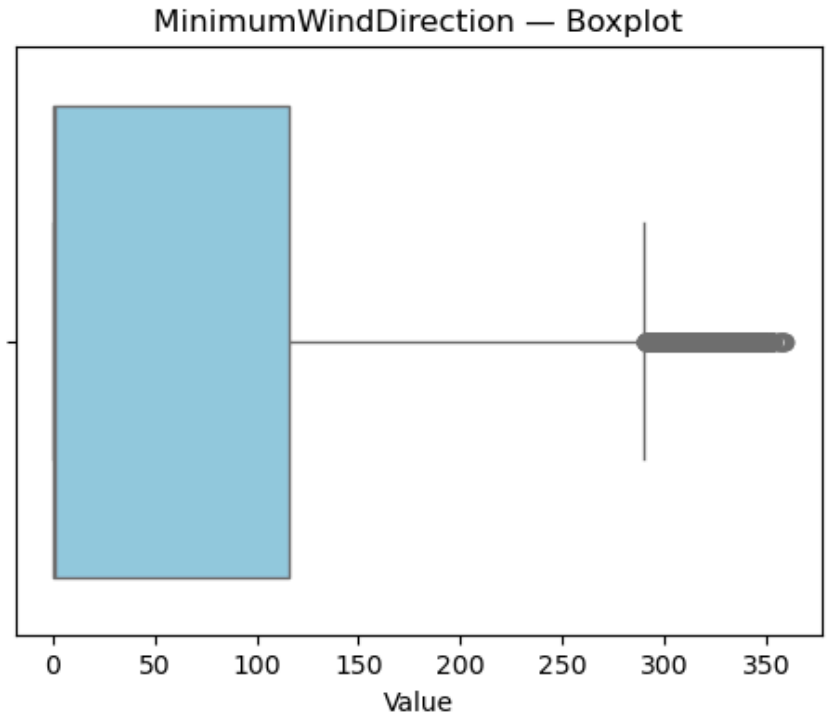
Noise — Boxplot

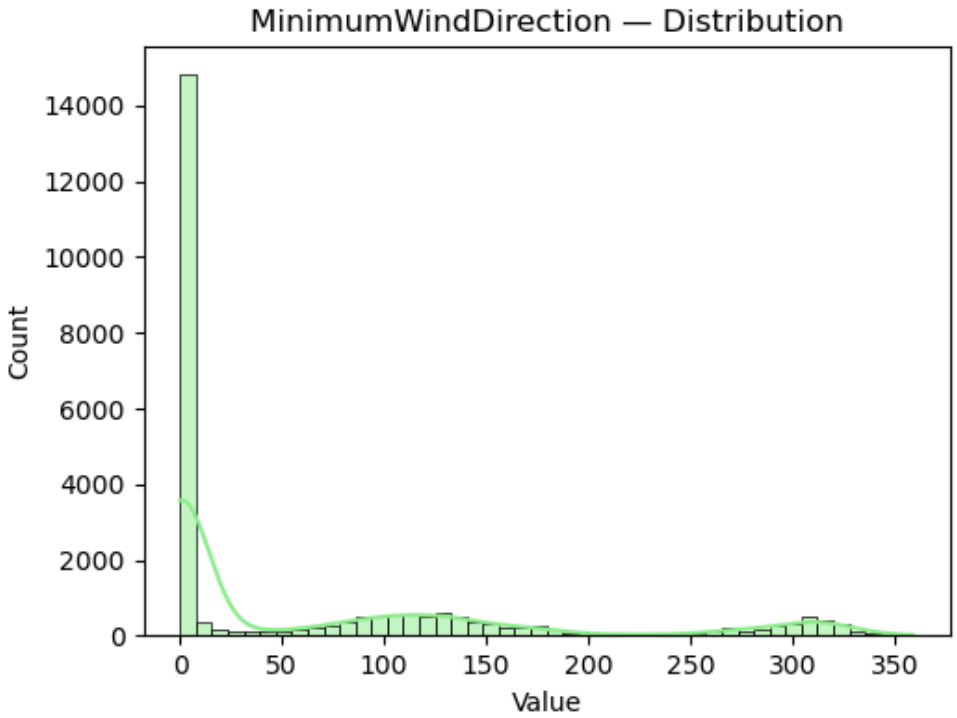
Noise — Distribution

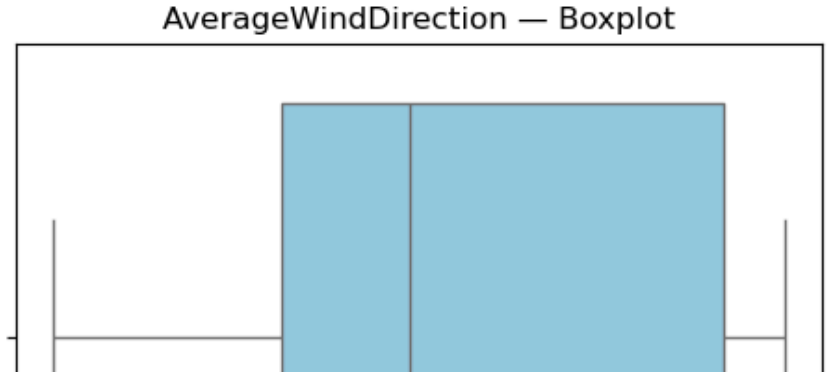

Noise — Boxplot
60
70
80
90
100
110
120
130
Value

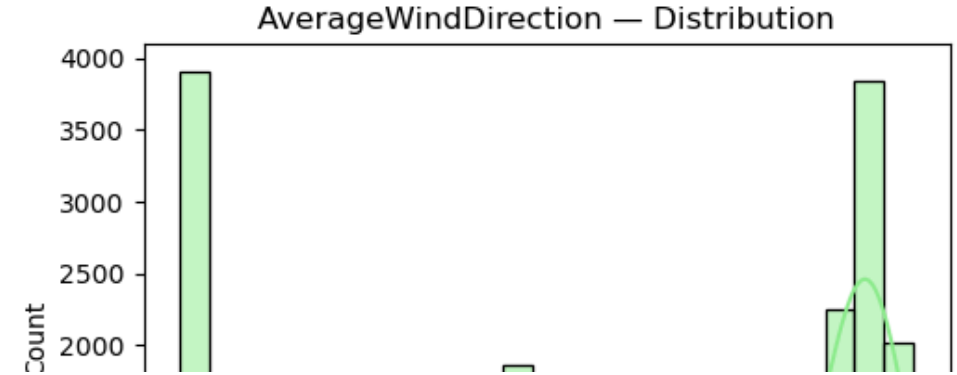

Noise — Distribution
4000
3500
3000
2500
2000
1500
1000
500
0
Count
60
70
80
90
100
110
120
130
Value

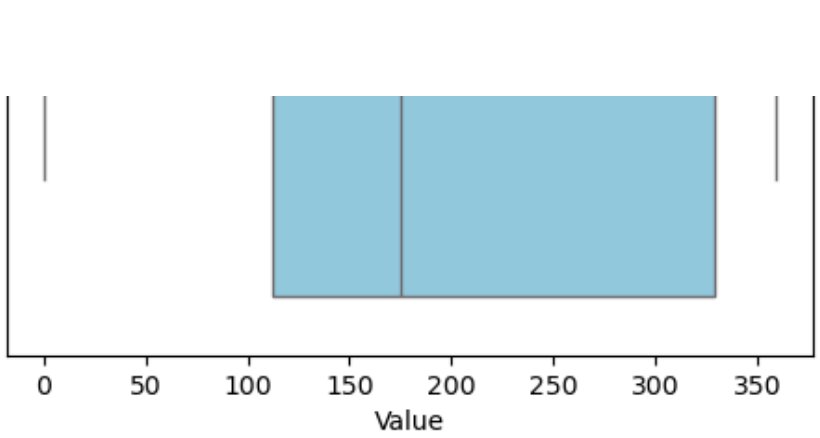
Device: ICTMicroclimate-10

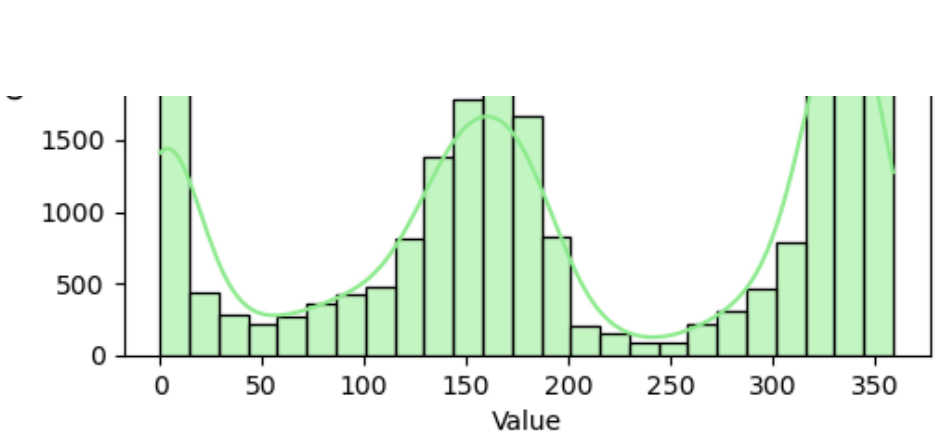




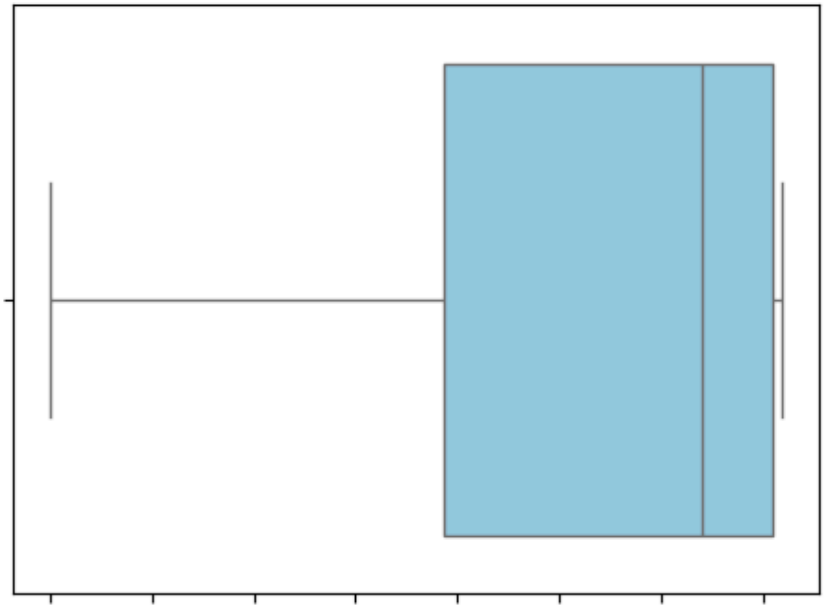




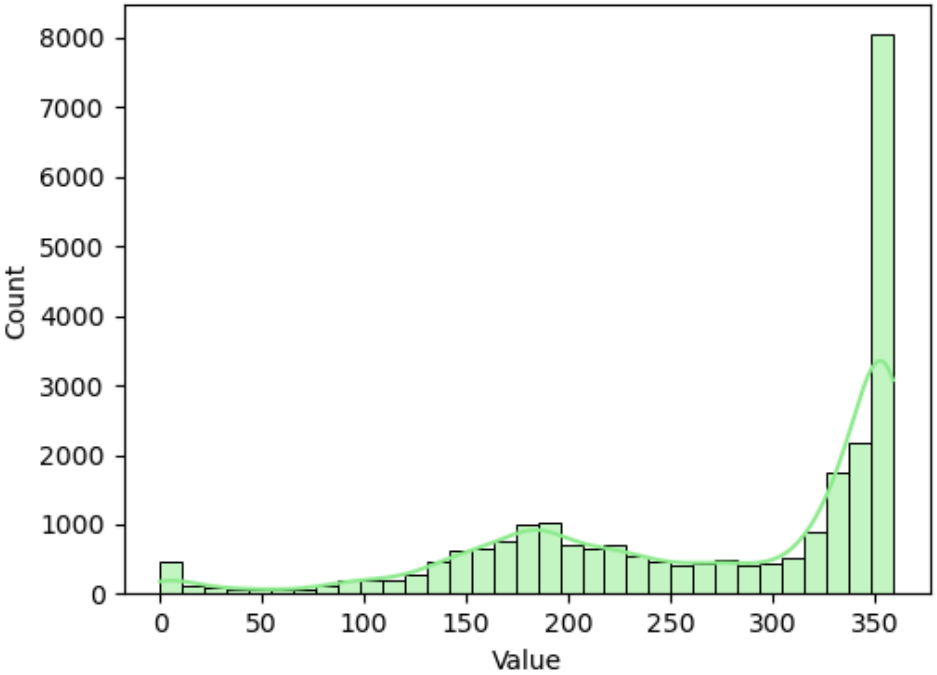




MaximumWindDirection — Boxplot

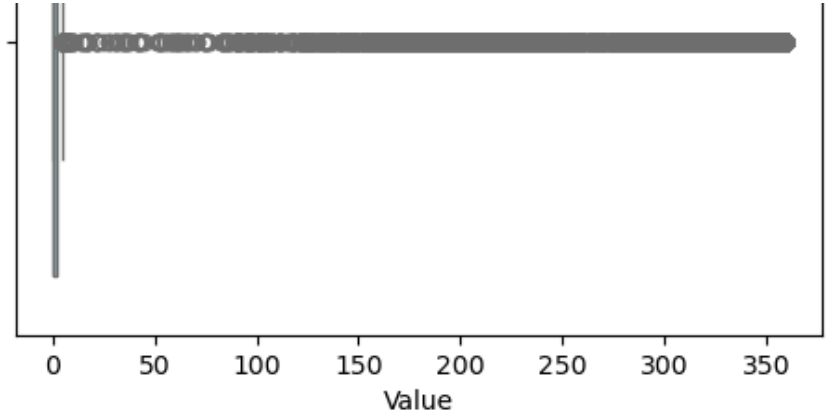


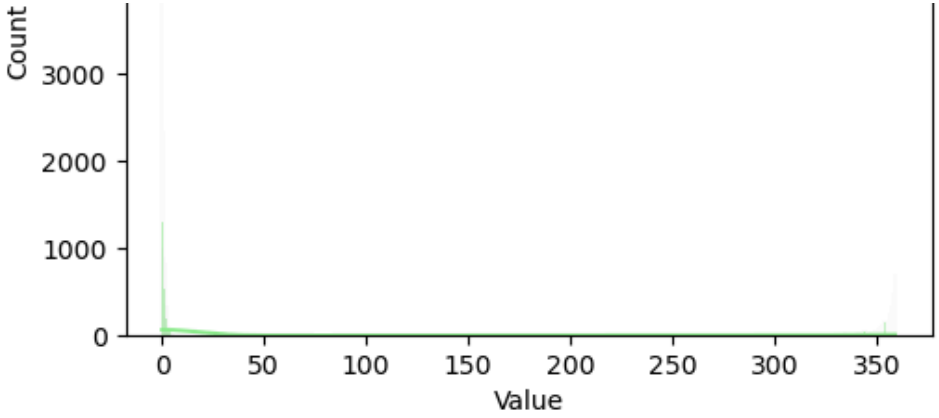
MaximumWindDirection — Distribution



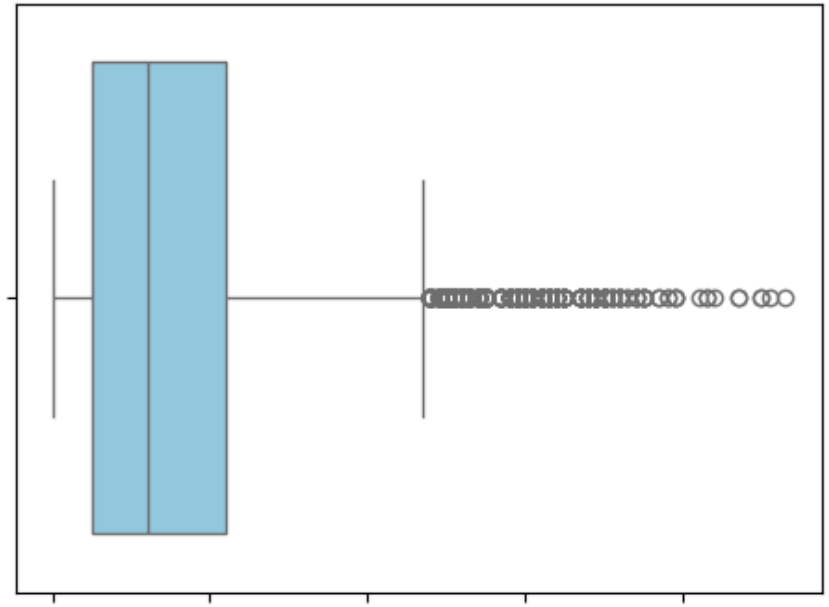
MinimumWindSpeed — Boxplot

MinimumWindSpeed — Distribution

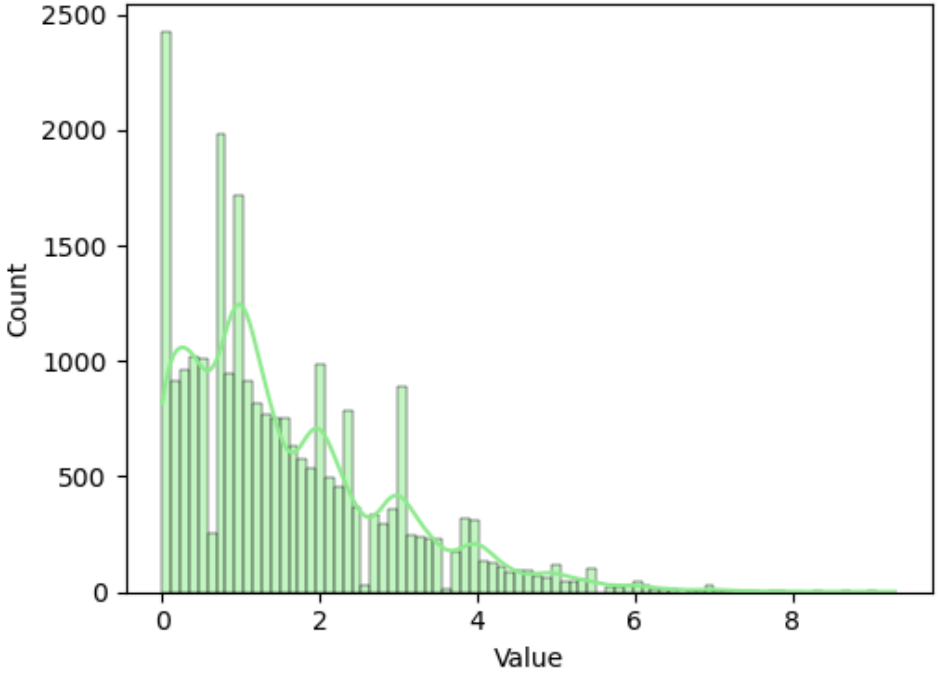




AverageWindSpeed — Boxplot

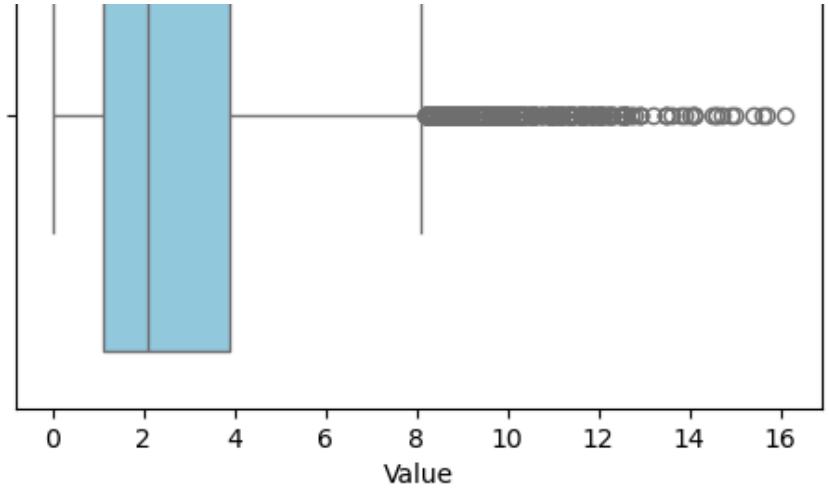


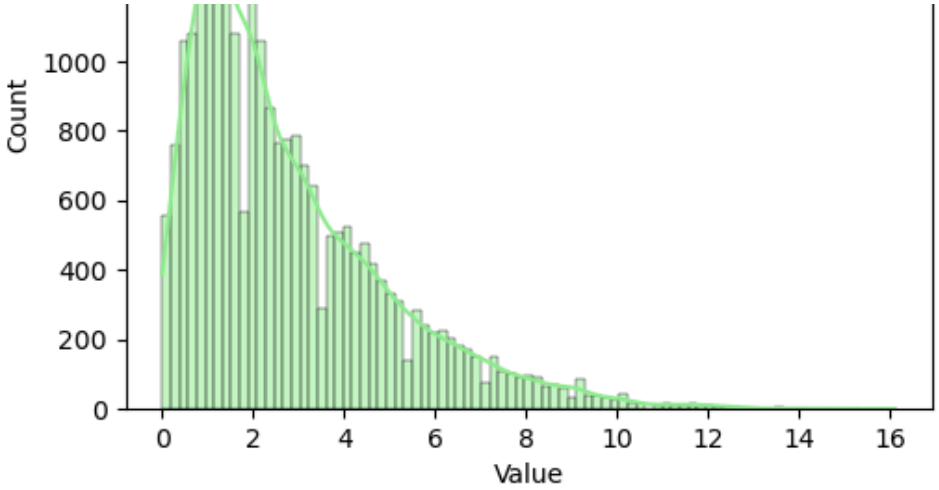
AverageWindSpeed — Distribution



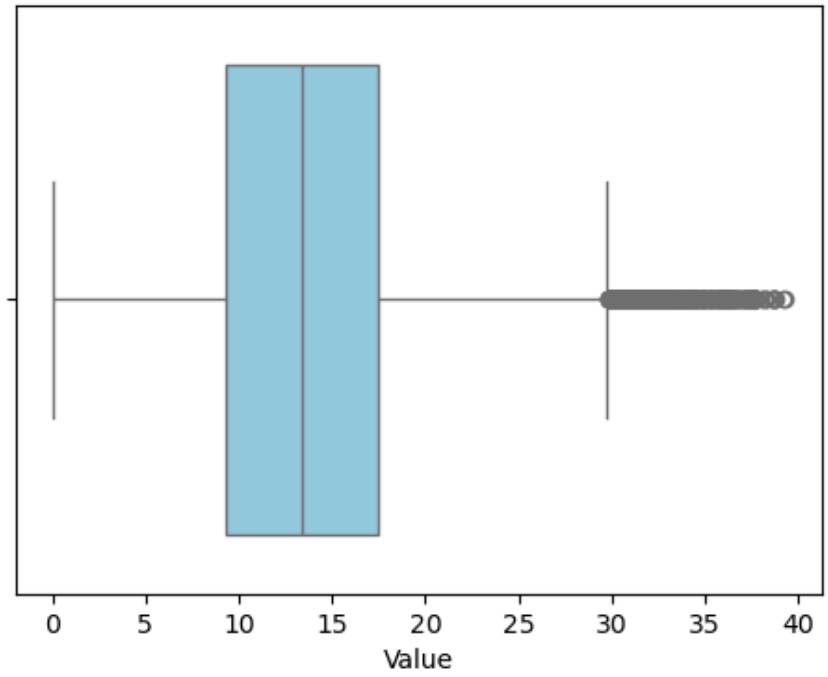
GustWindSpeed — Boxplot

GustWindSpeed — Distribution

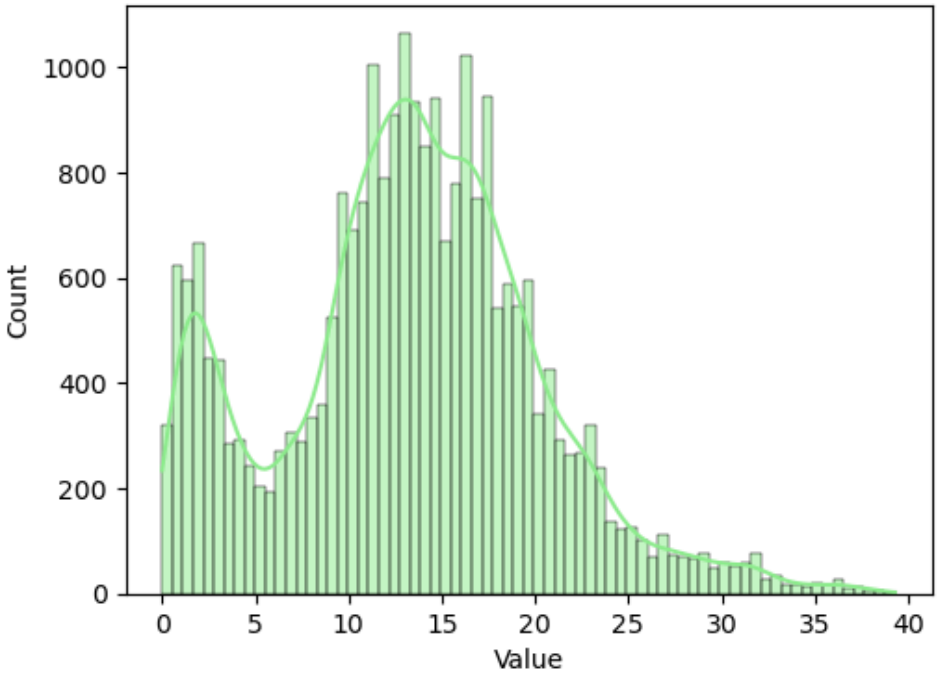




AirTemperature — Boxplot

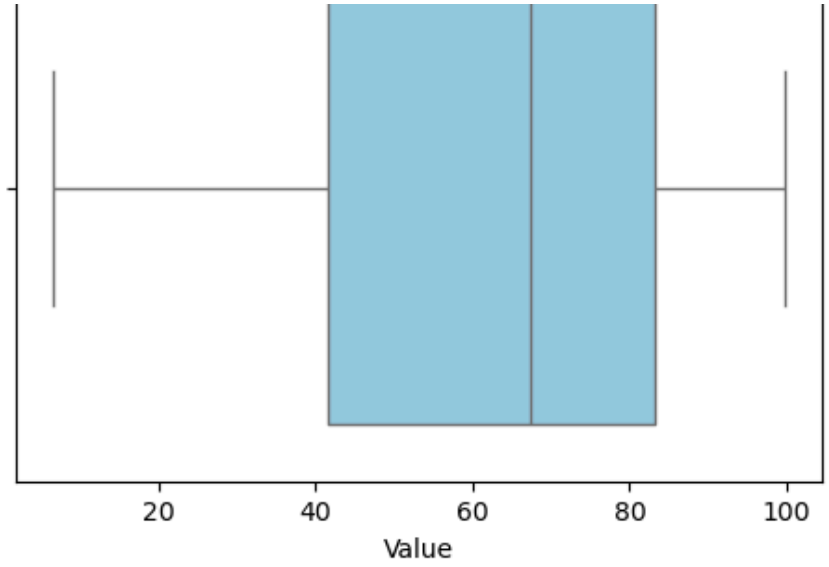


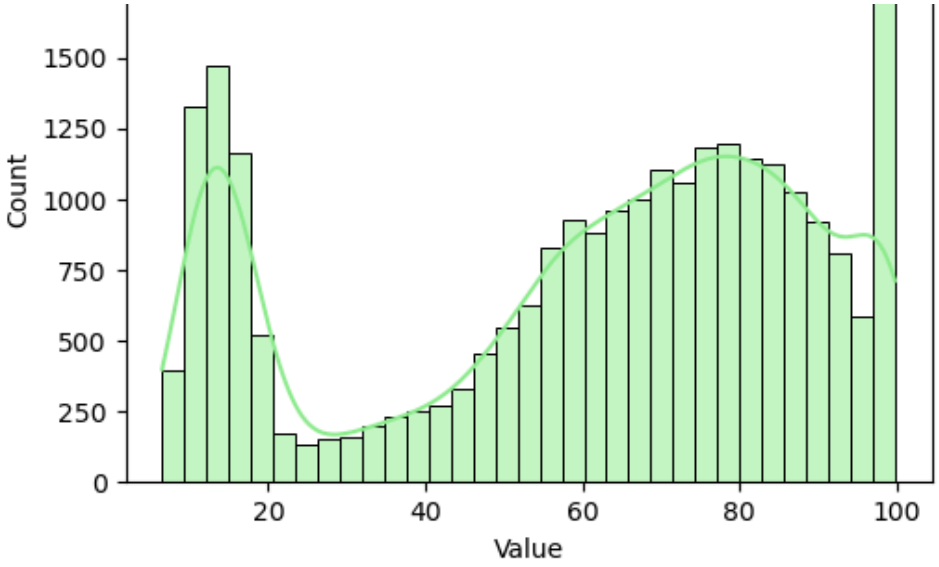
AirTemperature — Distribution



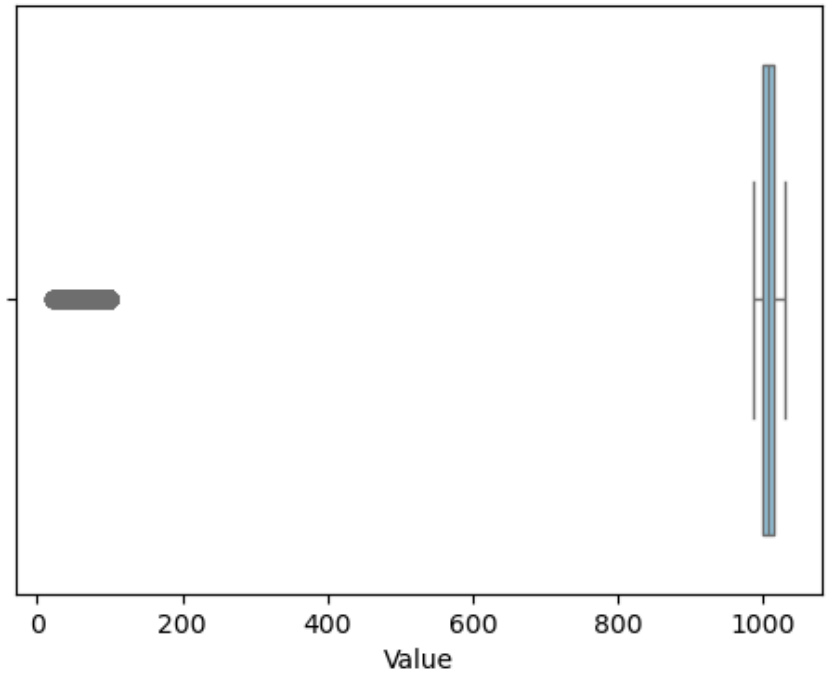
RelativeHumidity — Boxplot

RelativeHumidity — Distribution

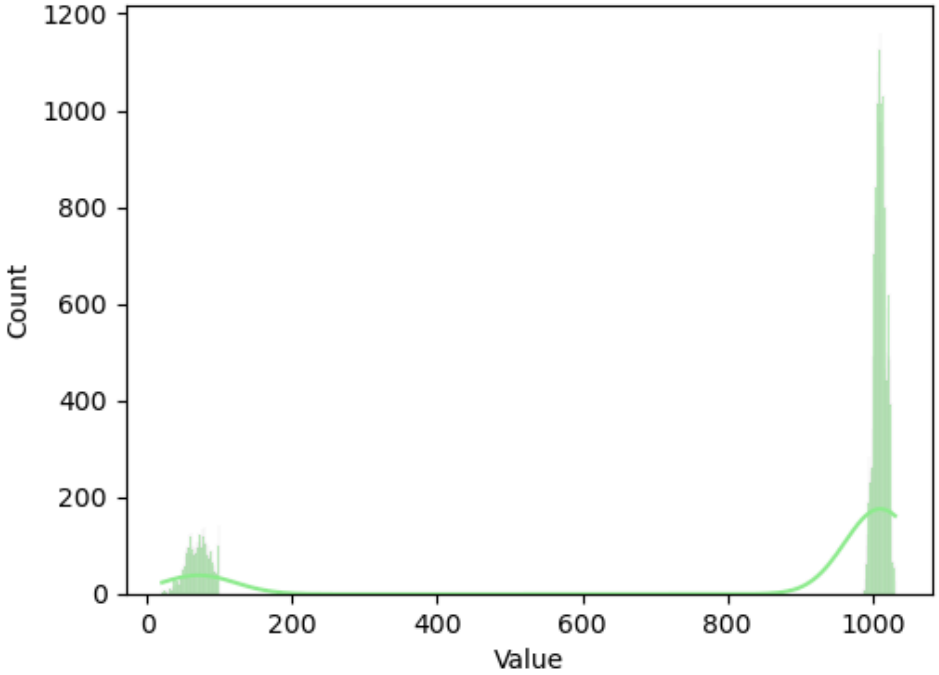




AtmosphericPressure — Boxplot

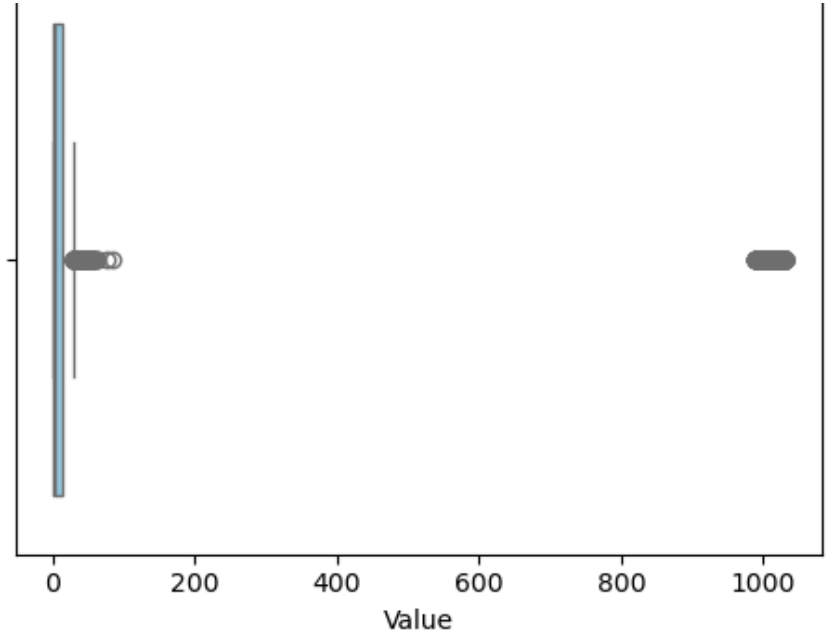


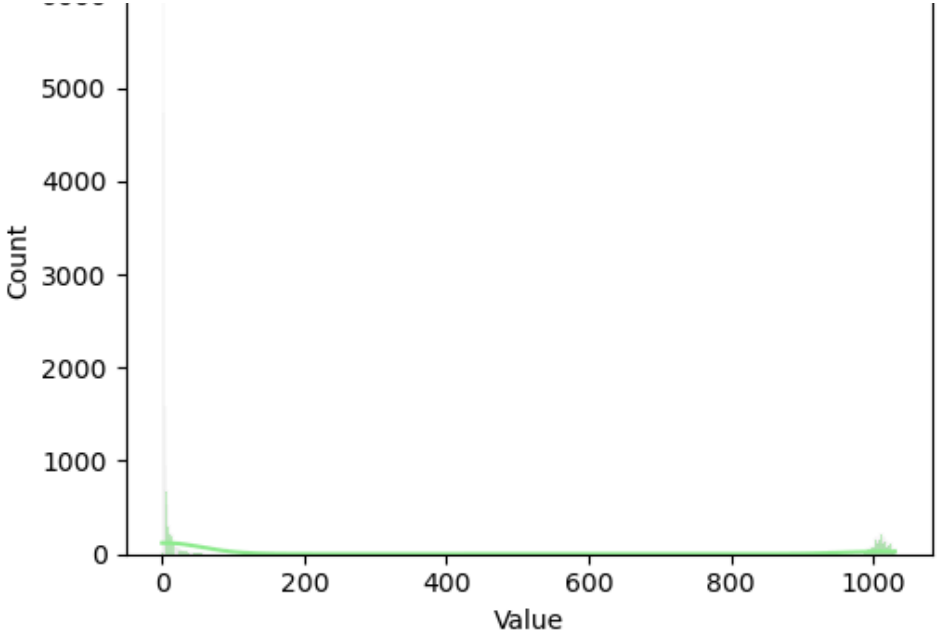
AtmosphericPressure — Distribution



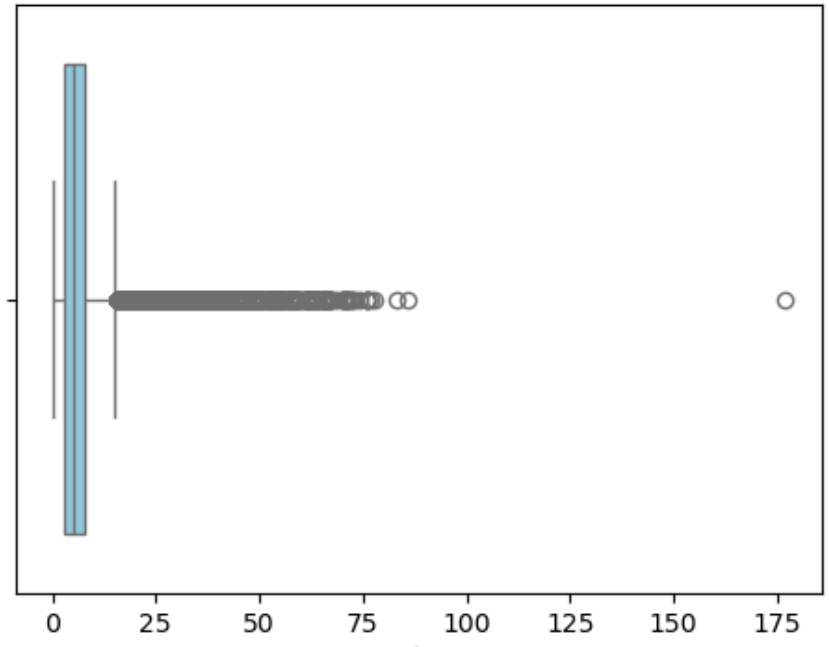
PM25 — Boxplot

PM25 — Distribution

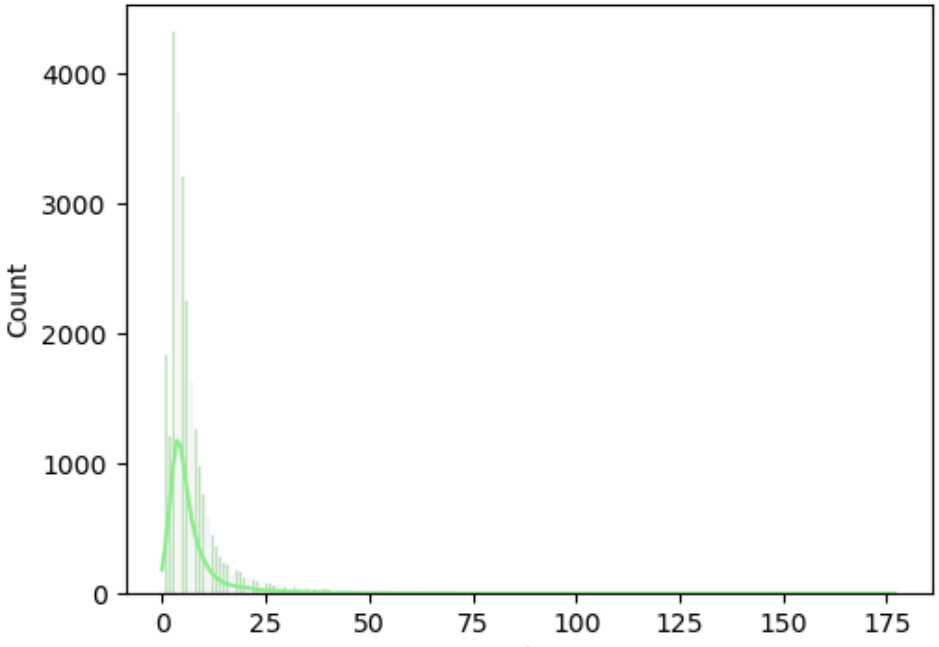


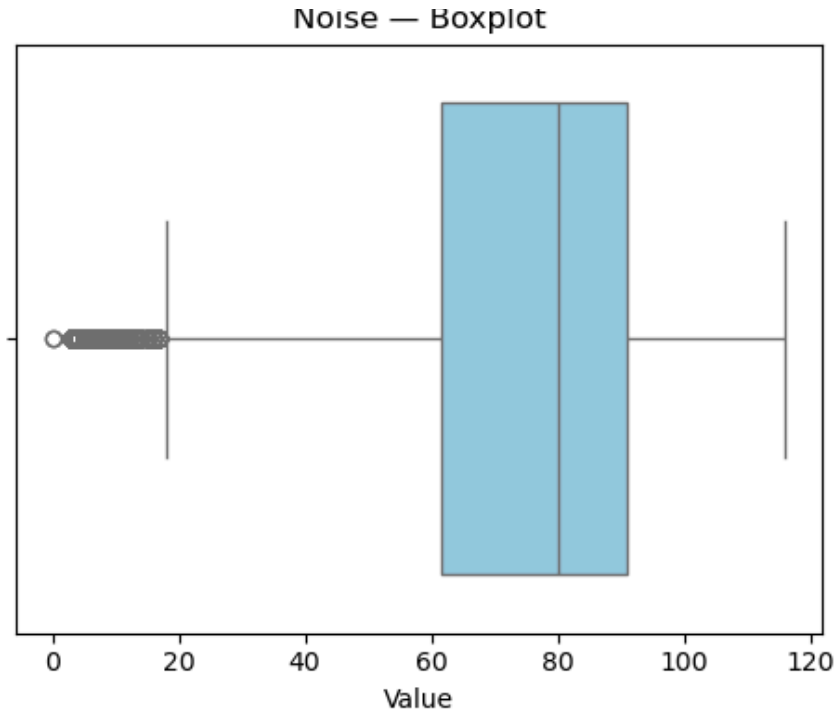


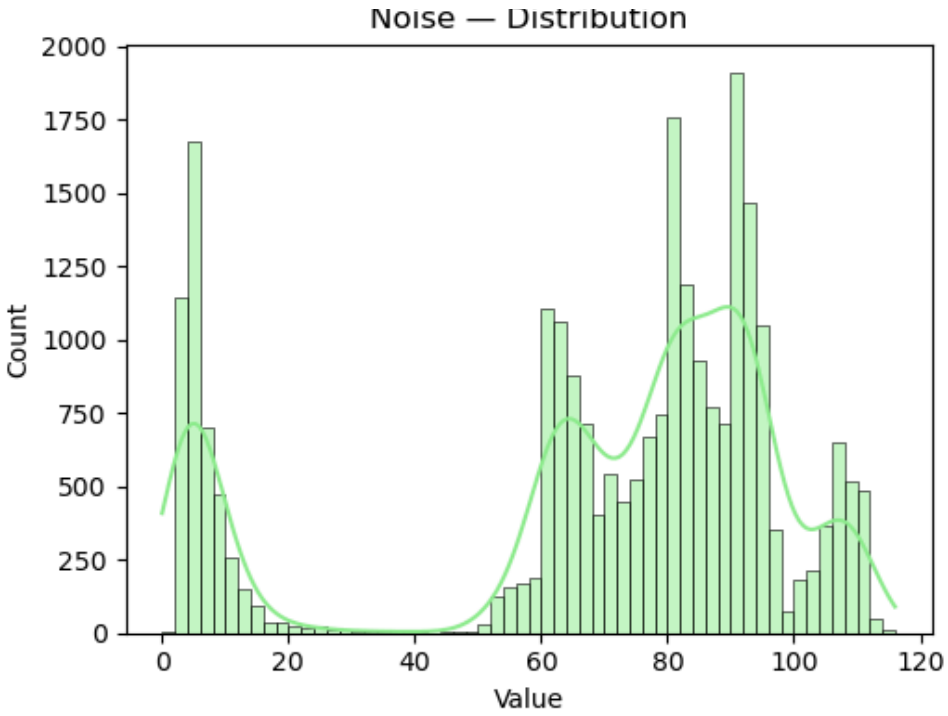
PM10 — Boxplot



PM10 — Distribution







In [45]:
```
data2.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 386468 entries, 0 to 386467
Data columns (total 16 columns):
 #   Column                Non-Null Count   Dtype  
---  ------                --------------   -----  
 0   Device_id             386468 non-null  object 
 1   Time                  386468 non-null  object 
 2   SensorLocation        386468 non-null  object 
 3   LatLong               386468 non-null  object 
 4   MinimumWindDirection  347670 non-null  float64
 5   AverageWindDirection  385967 non-null  float64
 6   MaximumWindDirection  347512 non-null  float64
 7   MinimumWindSpeed      347512 non-null  float64
 8   AverageWindSpeed      385967 non-null  float64
 9   GustWindSpeed         347512 non-null  float64
 10  AirTemperature        385967 non-null  float64
 11  RelativeHumidity      385967 non-null  float64
 12  AtmosphericPressure   385967 non-null  float64
 13  PM25                  368138 non-null  float64
 14  PM10                  368138 non-null  float64
 15  Noise                 368138 non-null  float64
dtypes: float64(12), object(4)
memory usage: 47.2+ MB
```

Step 1: Device-wise median imputation

In [46]:

```python
#  Make a copy of the original DataFrame to keep the raw data unchanged
data_imputed = data2.copy()

#  Loop through each sensor column you want to impute
for col in sensor_cols:
    #  For each column:
    # - Group by 'Device_id'
    # - For each group, compute the median for that device
    # - Fill NaN values in the group with that median
    # - .transform() keeps the original DataFrame shape and aligns results properly
    data_imputed[col] = (
        data_imputed
        .groupby('Device_id')[col]
```

```
        .transform(lambda grp: grp.fillna(grp.median()))
    )
```

In [47]:
```
# Ensure no NaNs remain in sensor columns
data_imputed.groupby('Device_id')[sensor_cols].apply(lambda g: g.isna().sum())
```

Out[47]:

Device_id	MinimumWindDirection	AverageWindDirection	MaximumWindDirection	MinimumWindSpeed	AverageWindSpeed	Gust
ICTMicroclimate-01	38483	0	38483	38483	0	
ICTMicroclimate-02	0	0	0	0	0	
ICTMicroclimate-03	0	0	0	0	0	
ICTMicroclimate-04	0	0	0	0	0	
ICTMicroclimate-05	0	0	0	0	0	
ICTMicroclimate-06	0	0	0	0	0	
ICTMicroclimate-07	0	0	0	0	0	
ICTMicroclimate-08	0	0	0	0	0	
ICTMicroclimate-09	0	0	0	0	0	
ICTMicroclimate-10	0	0	0	0	0	
ICTMicroclimate-11	0	0	0	0	0	
aws5-0999	0	0	0	0	0	

In [48]:
```
data_imputed.isnull().sum()
```

Out[48]:
```
Device_id                 0
Time                      0
SensorLocation            0
LatLong                   0
MinimumWindDirection    38483
AverageWindDirection        0
MaximumWindDirection    38483
MinimumWindSpeed        38483
AverageWindSpeed            0
GustWindSpeed           38483
AirTemperature              0
RelativeHumidity            0
AtmosphericPressure         0
PM25                    17829
PM10                    17829
Noise                   17829
dtype: int64
```

In [49]:
```
data_imputed.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 386468 entries, 0 to 386467
Data columns (total 16 columns):
 #   Column                Non-Null Count   Dtype
---  ------                --------------   -----
 0   Device_id             386468 non-null  object
 1   Time                  386468 non-null  object
 2   SensorLocation        386468 non-null  object
 3   LatLong               386468 non-null  object
 4   MinimumWindDirection  347985 non-null  float64
 5   AverageWindDirection  386468 non-null  float64
 6   MaximumWindDirection  347985 non-null  float64
 7   MinimumWindSpeed      347985 non-null  float64
 8   AverageWindSpeed      386468 non-null  float64
 9   GustWindSpeed         347985 non-null  float64
 10  AirTemperature        386468 non-null  float64
 11  RelativeHumidity      386468 non-null  float64
 12  AtmosphericPressure   386468 non-null  float64
 13  PM25                  368639 non-null  float64
 14  PM10                  368639 non-null  float64
 15  Noise                 368639 non-null  float64
dtypes: float64(12), object(4)
memory usage: 47.2+ MB
```

observation

Observations on Missing Values: **1. Complete Columns (0 missing values):**

- These don't need imputation:
- Device_id, Time, SensorLocation, LatLong, AverageWindDirection, AverageWindSpeed, AirTemperature, RelativeHumidity, AtmosphericPressure.

2. Partially Missing Columns (17,829 missing):

- PM25, PM10, Noise
- Moderate missingness (~31% if total rows = 57,600).
- Action: Group-level median imputation (by Device_id) is appropriate.

3.Highly Missing Columns (38,483 missing):

- MinimumWindDirection, MaximumWindDirection, MinimumWindSpeed, GustWindSpeed
- Extreme missingness (~67% if total rows = 57,600).
- Risk: Group medians might still leave gaps if all values are NaN for some devices.
- Action: Fall back to global median if group median is unavailable.

Step 2: Fill residual NaNs with global median

In [55]:
```
# Loop through each sensor column to handle any remaining NaNs
for col in sensor_cols:
    #  Calculate the global median for the whole column (across all devices)
    global_median = data_imputed[col].median()

    # Create a boolean mask: True where the value is still NaN
    mask = data_imputed[col].isna()

    # Fill those remaining NaNs with the global median
    data_imputed.loc[mask, col] = global_median
```

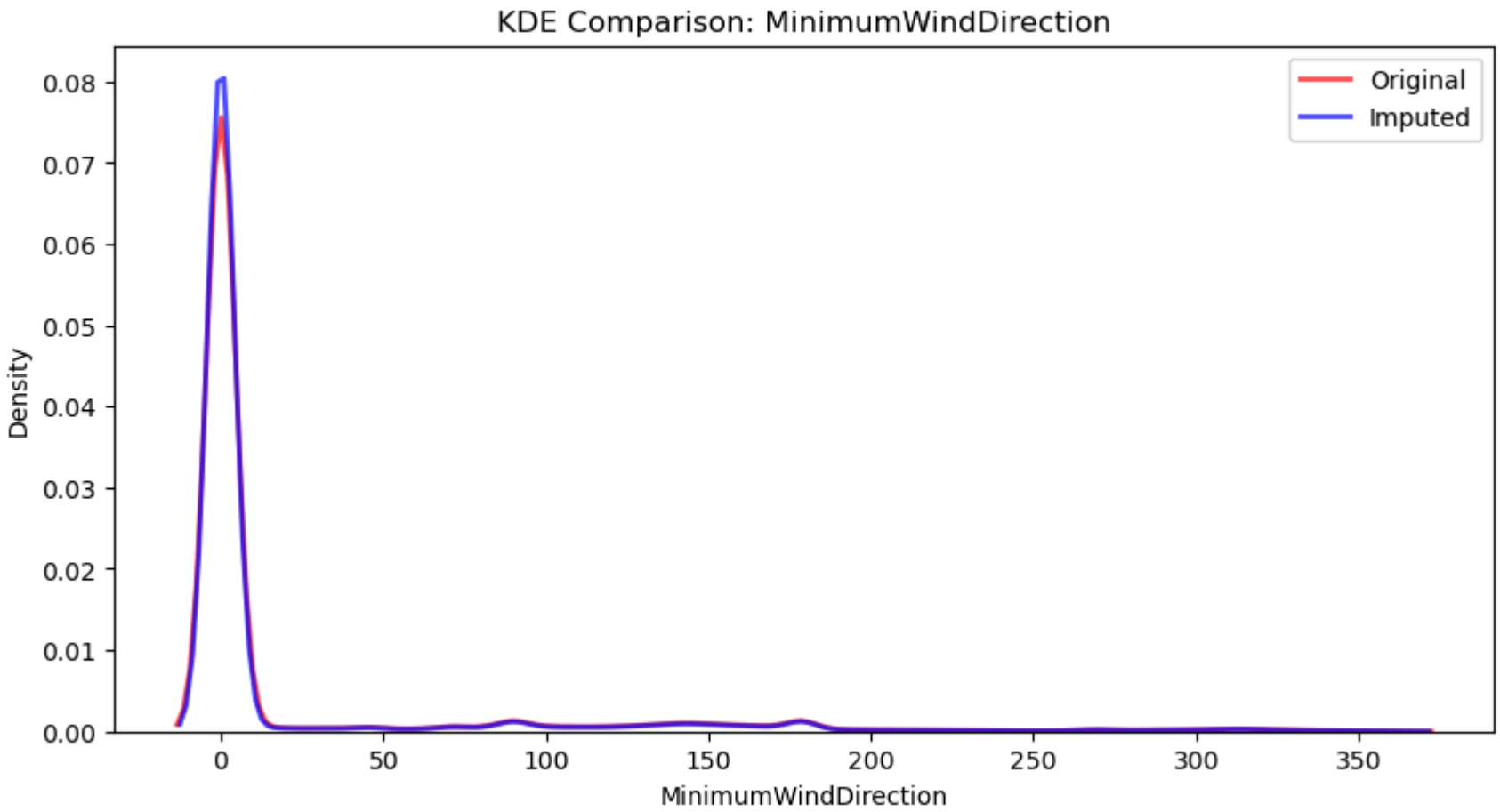
In [56]:
```
data_imputed.isnull().sum()
```

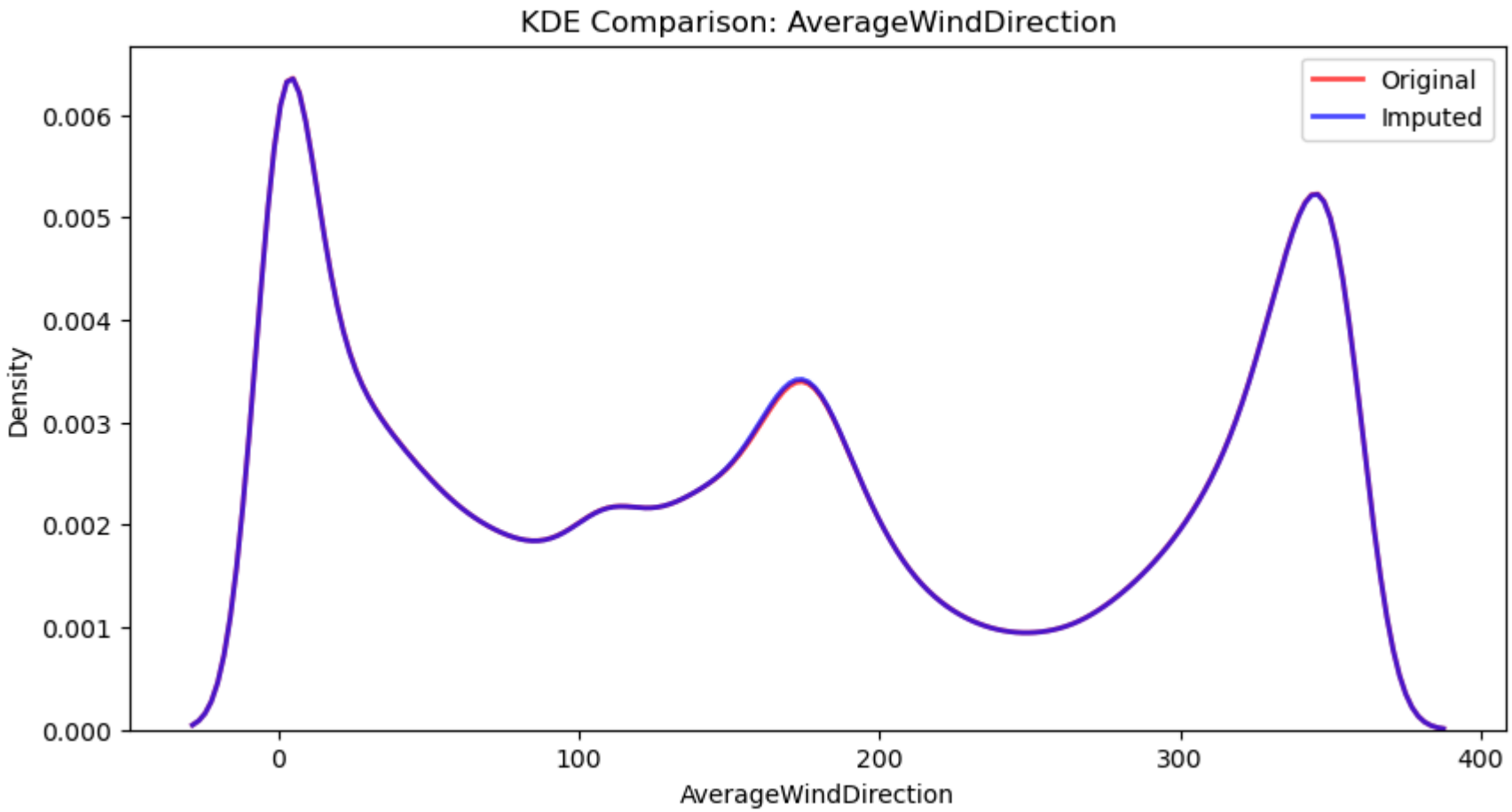
Out[56]:
```
Device_id               0
Time                    0
SensorLocation          0
LatLong                 0
MinimumWindDirection    0
AverageWindDirection    0
MaximumWindDirection    0
MinimumWindSpeed        0
AverageWindSpeed        0
GustWindSpeed           0
AirTemperature          0
RelativeHumidity        0
AtmosphericPressure     0
PM25                    0
PM10                    0
Noise                   0
dtype: int64
```

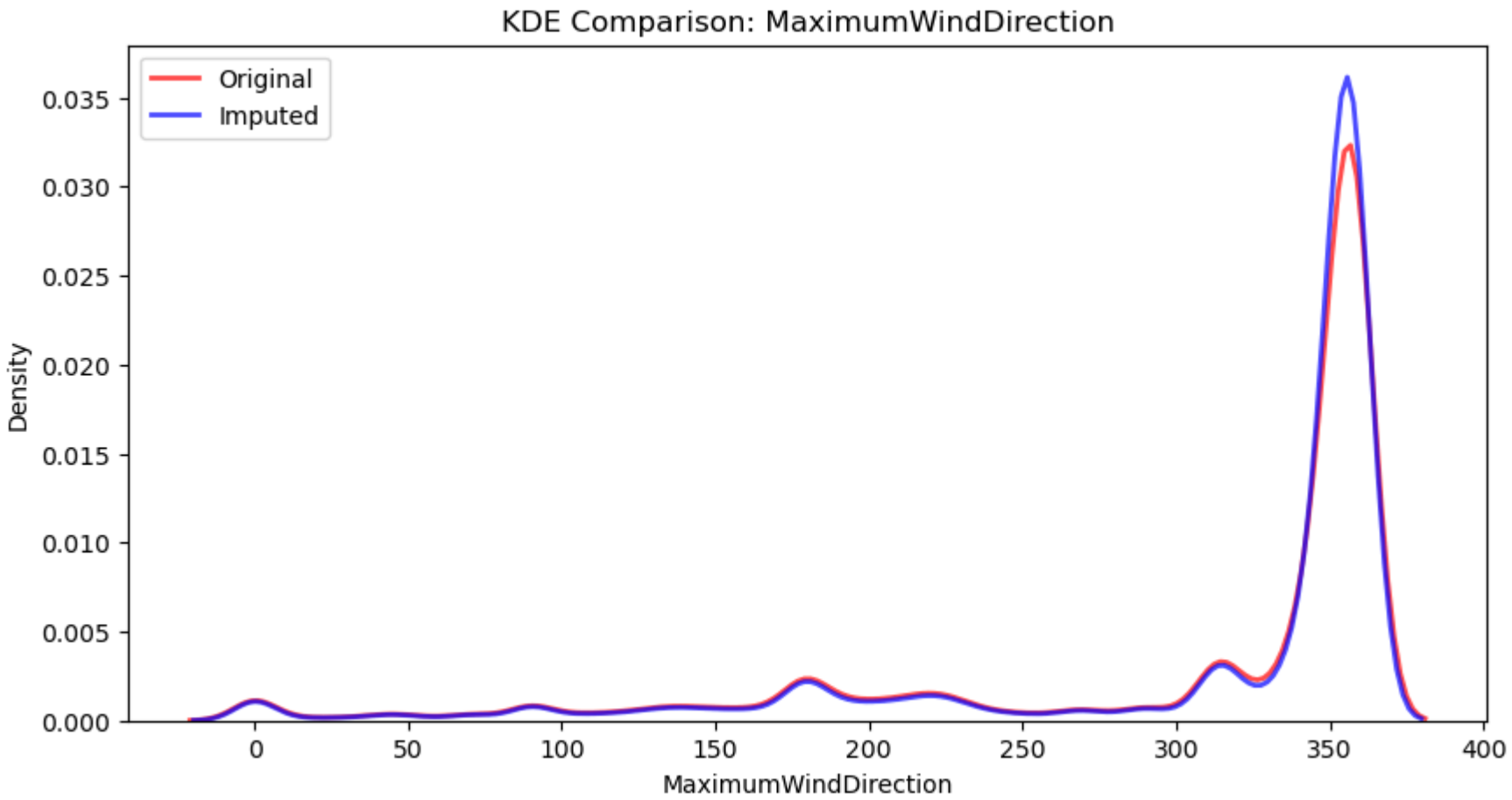
In [57]:
```
cols_to_plot = [
    'MinimumWindDirection', 'AverageWindDirection', 'MaximumWindDirection',
    'MinimumWindSpeed', 'AverageWindSpeed', 'GustWindSpeed',
    'AirTemperature', 'RelativeHumidity', 'AtmosphericPressure',
    'PM25', 'PM10', 'Noise'
]
```

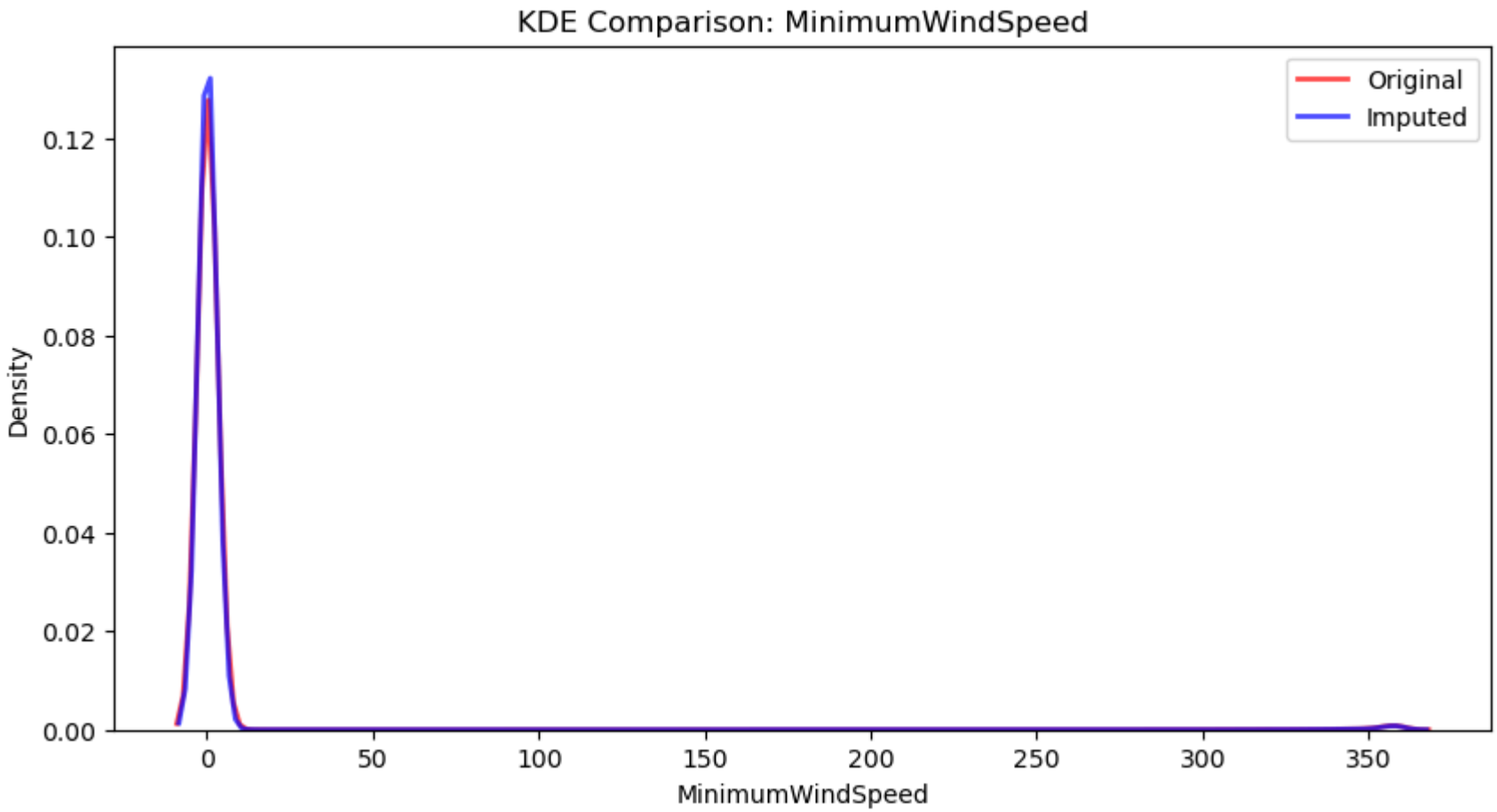
In [58]:
```
#  Loop through each column you want to compare
for col in cols_to_plot:
    # Create a new figure for each plot, with specified size
    plt.figure(figsize=(10, 5))

    # Plot the KDE (Kernel Density Estimate) for the original data in red
    sns.kdeplot(
        data1[col],              # Original column values
        color='red',
        label='Original',
        alpha=0.7,               # Transparency
        linewidth=2
    )

    #  Plot the KDE for the imputed data in blue
```

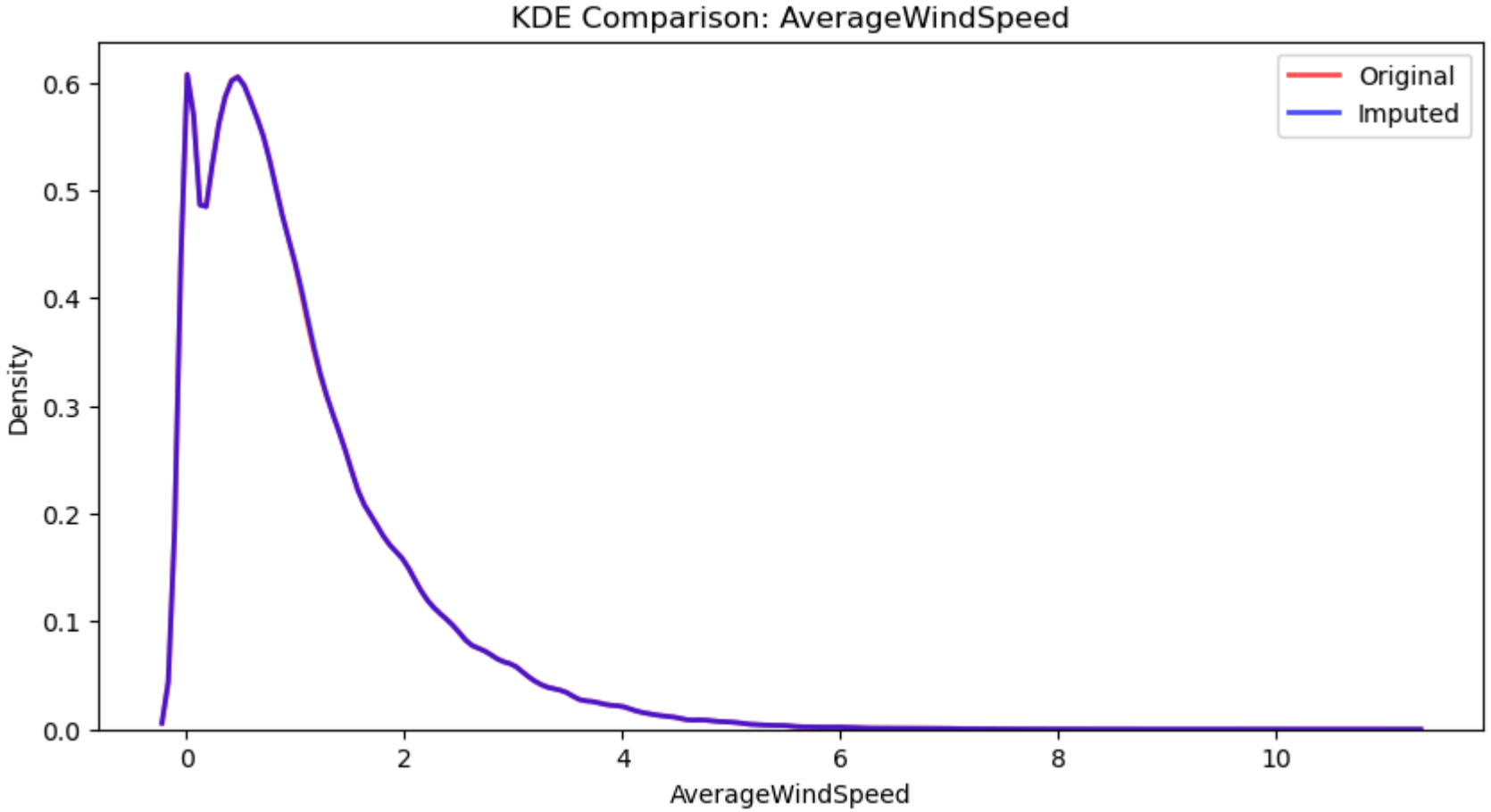
```
    sns.kdeplot(
        data_imputed[col],        # Imputed column values
        color='blue',
        label='Imputed',
        alpha=0.7,
        linewidth=2
    )

    # Add a descriptive title for the plot
    plt.title(f'KDE Comparison: {col}')

    #  Add legend to distinguish original vs imputed
    plt.legend()

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

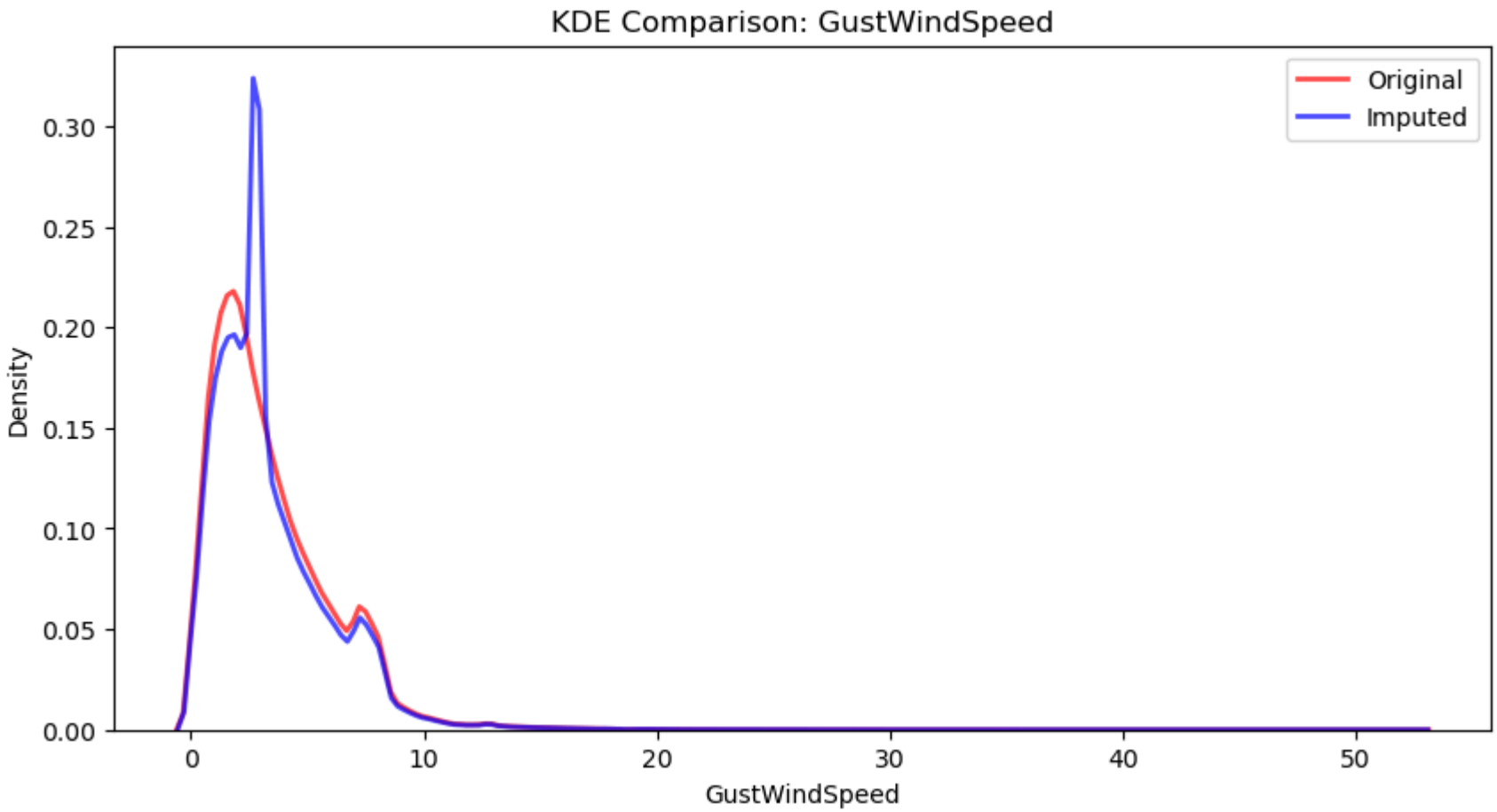

KDE Comparison: MinimumWindDirection
Original
Imputed
Density
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
0
50
100
150
200
250
300
350
MinimumWindDirection

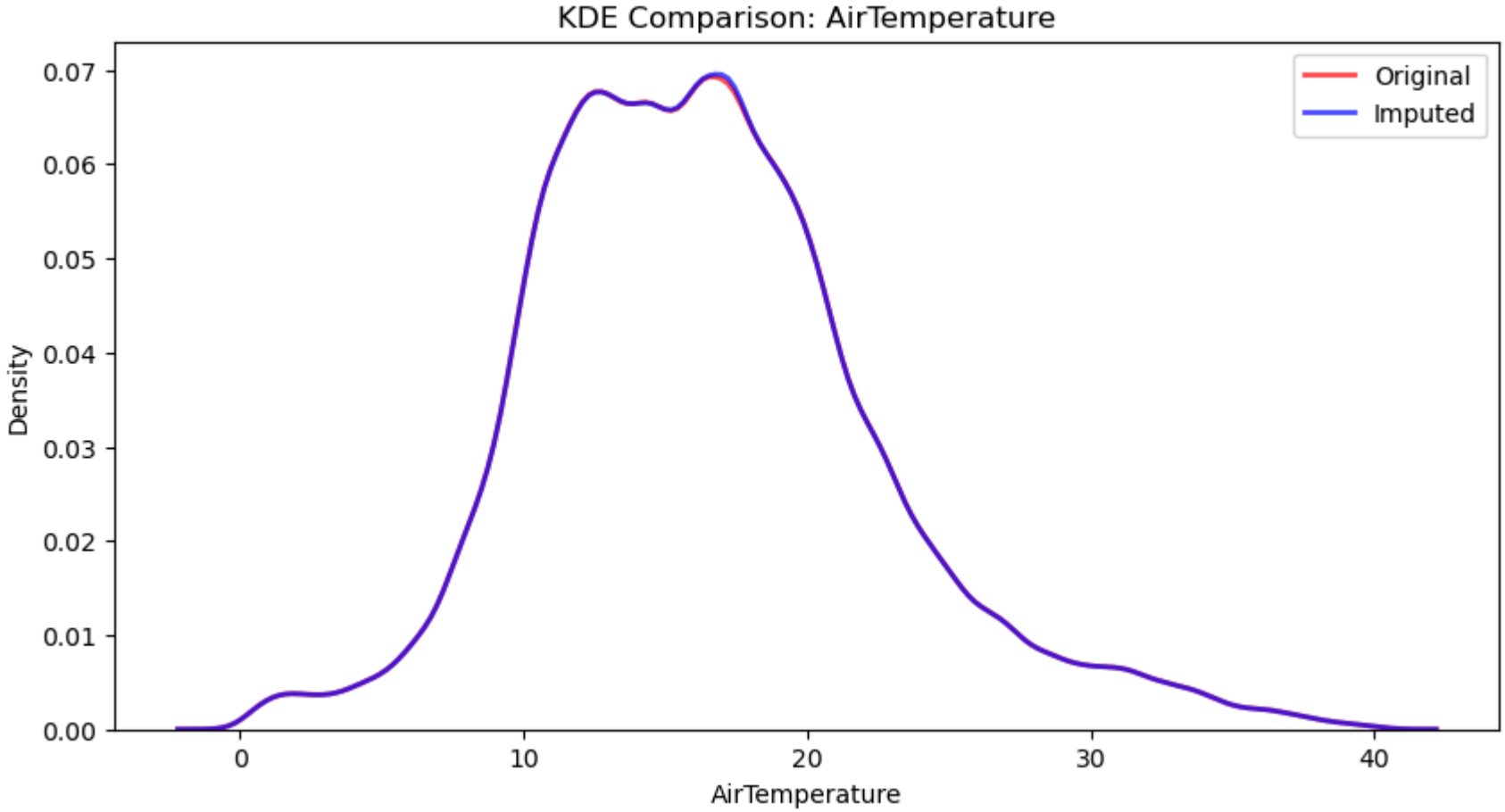

KDE Comparison: AverageWindDirection
Original
Imputed
Density
0.006
0.005
0.004
0.003
0.002
0.001
0.000
0
100
200
300
400
AverageWindDirection

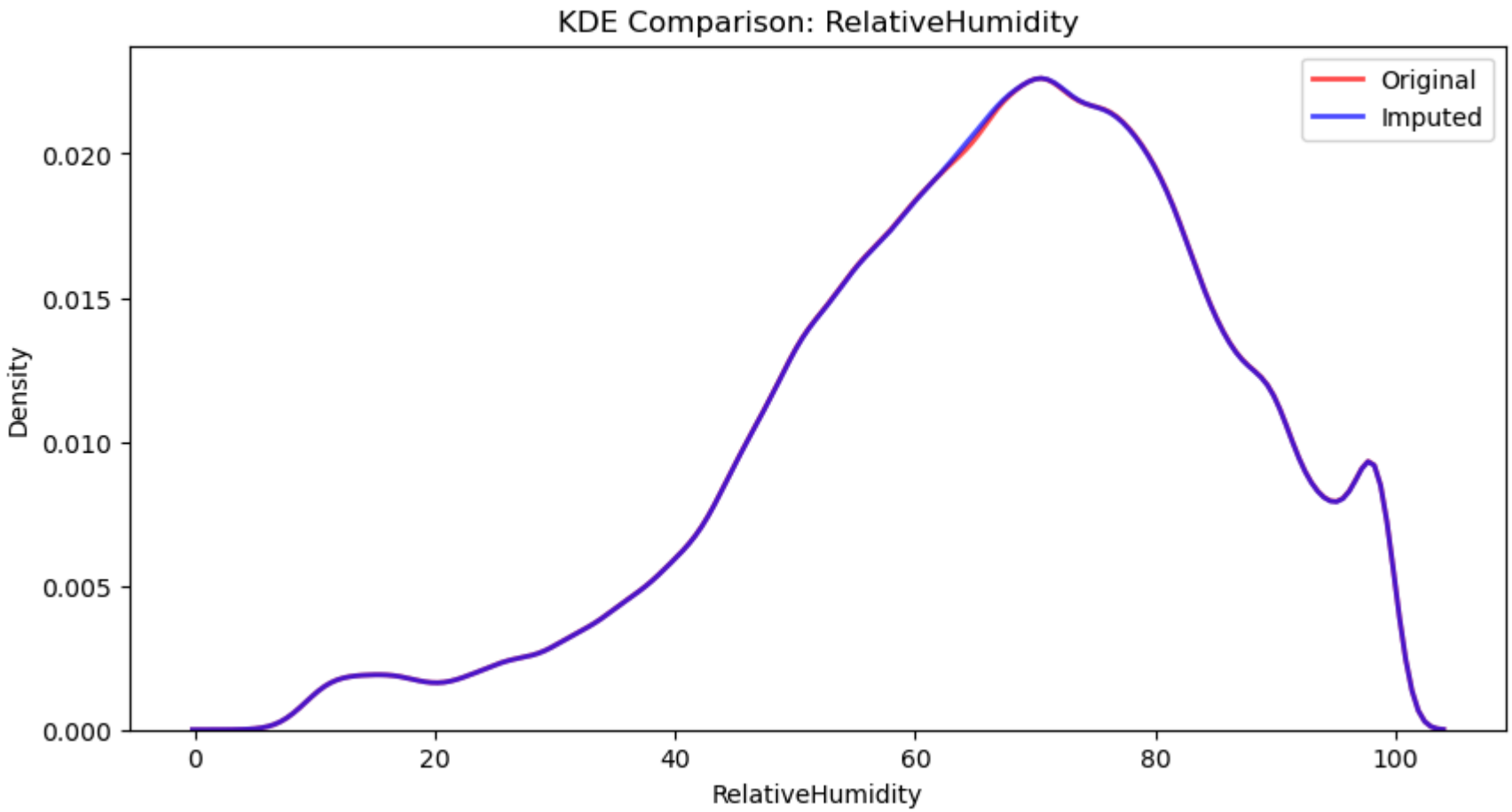

KDE Comparison: MaximumWindDirection
Original
Imputed
Density
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
0
50
100
150
200
250
300
350
400
MaximumWindDirection

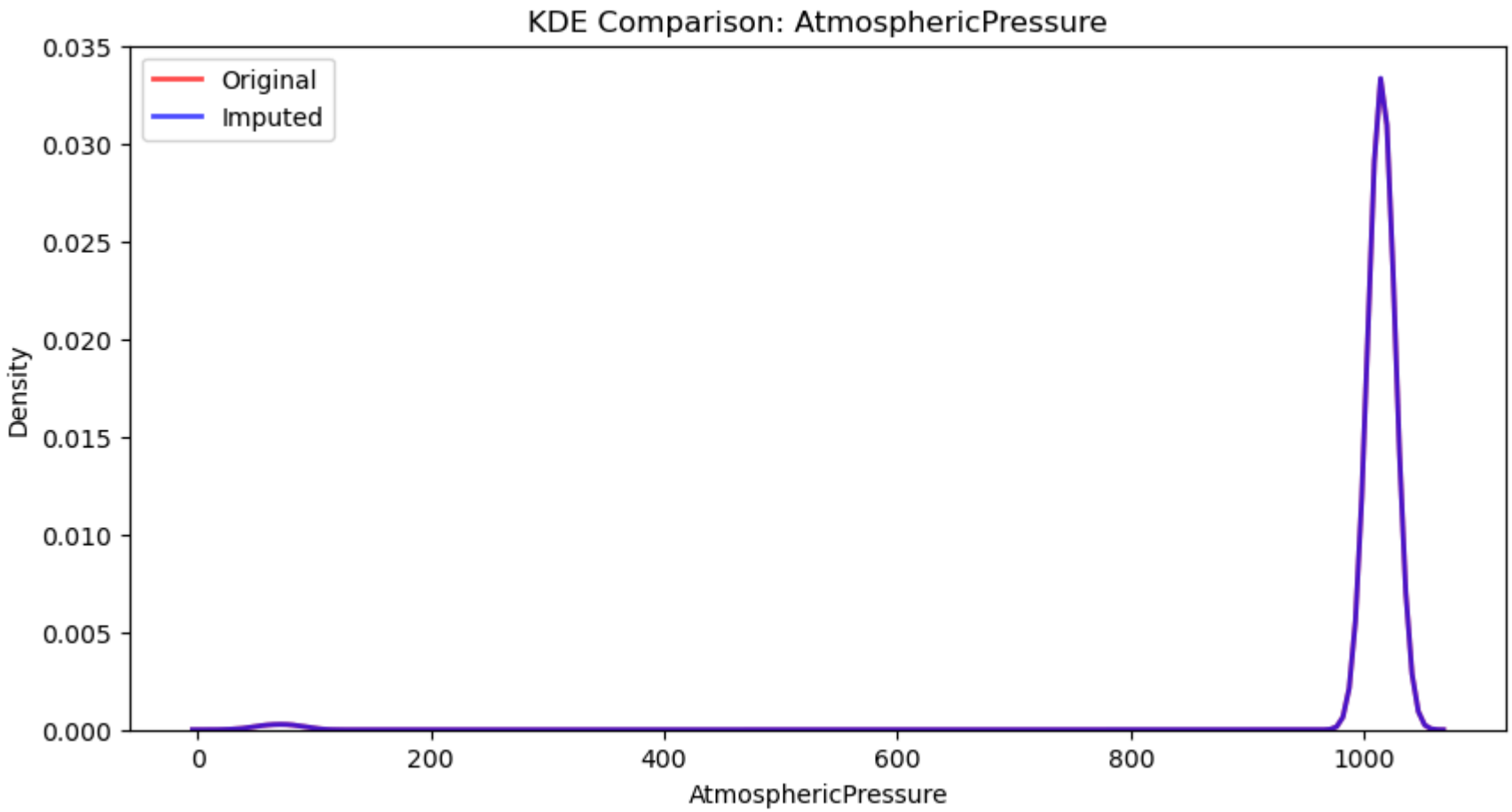

KDE Comparison: MinimumWindSpeed
Original
Imputed
Density
0.12
0.10
0.08
0.06
0.04
0.02
0.00
0
50
100
150
200
250
300
350
MinimumWindSpeed

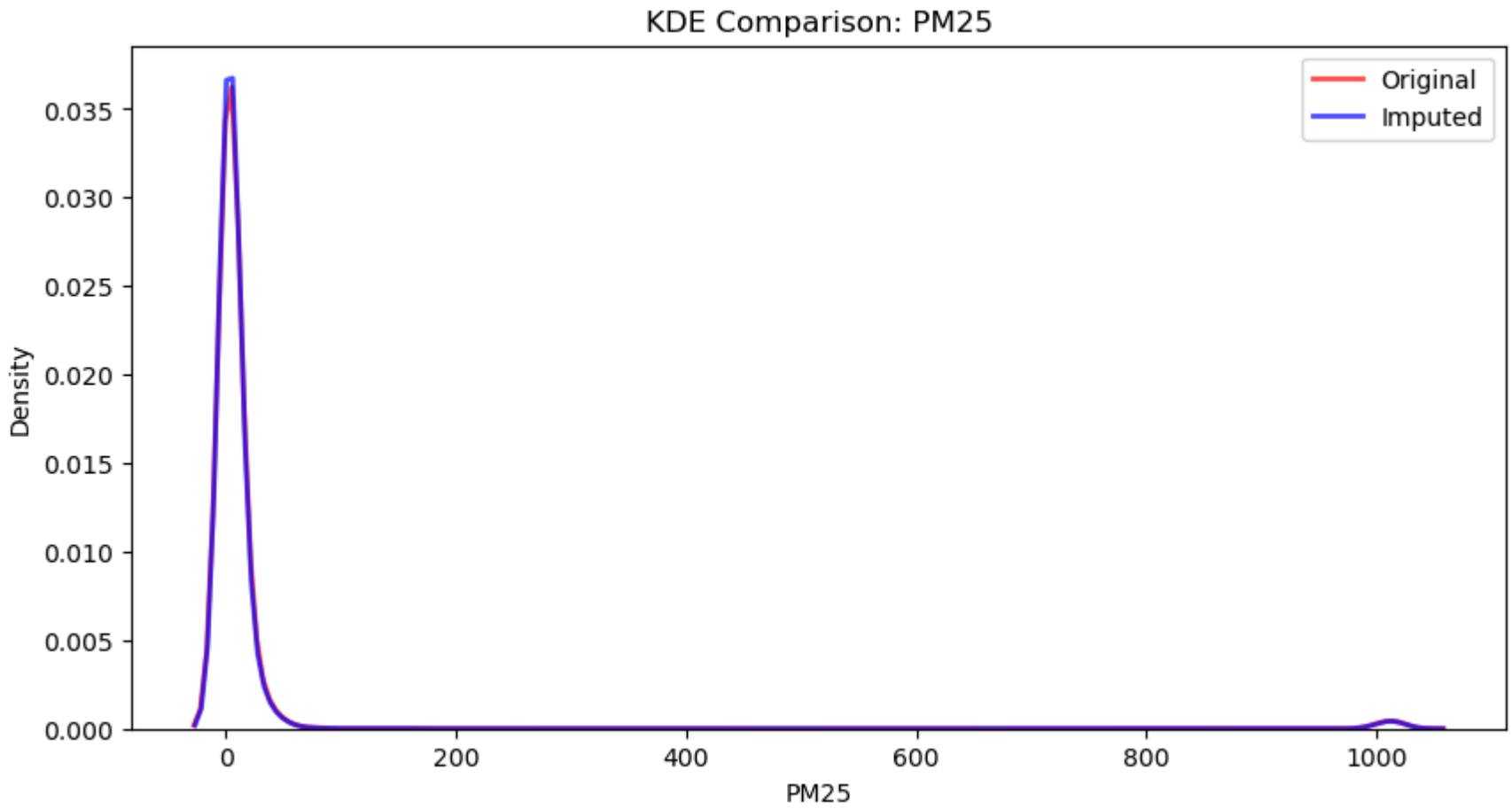

KDE Comparison: AverageWindSpeed
Original
Imputed
Density
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0
2
4
6
8
10
AverageWindSpeed

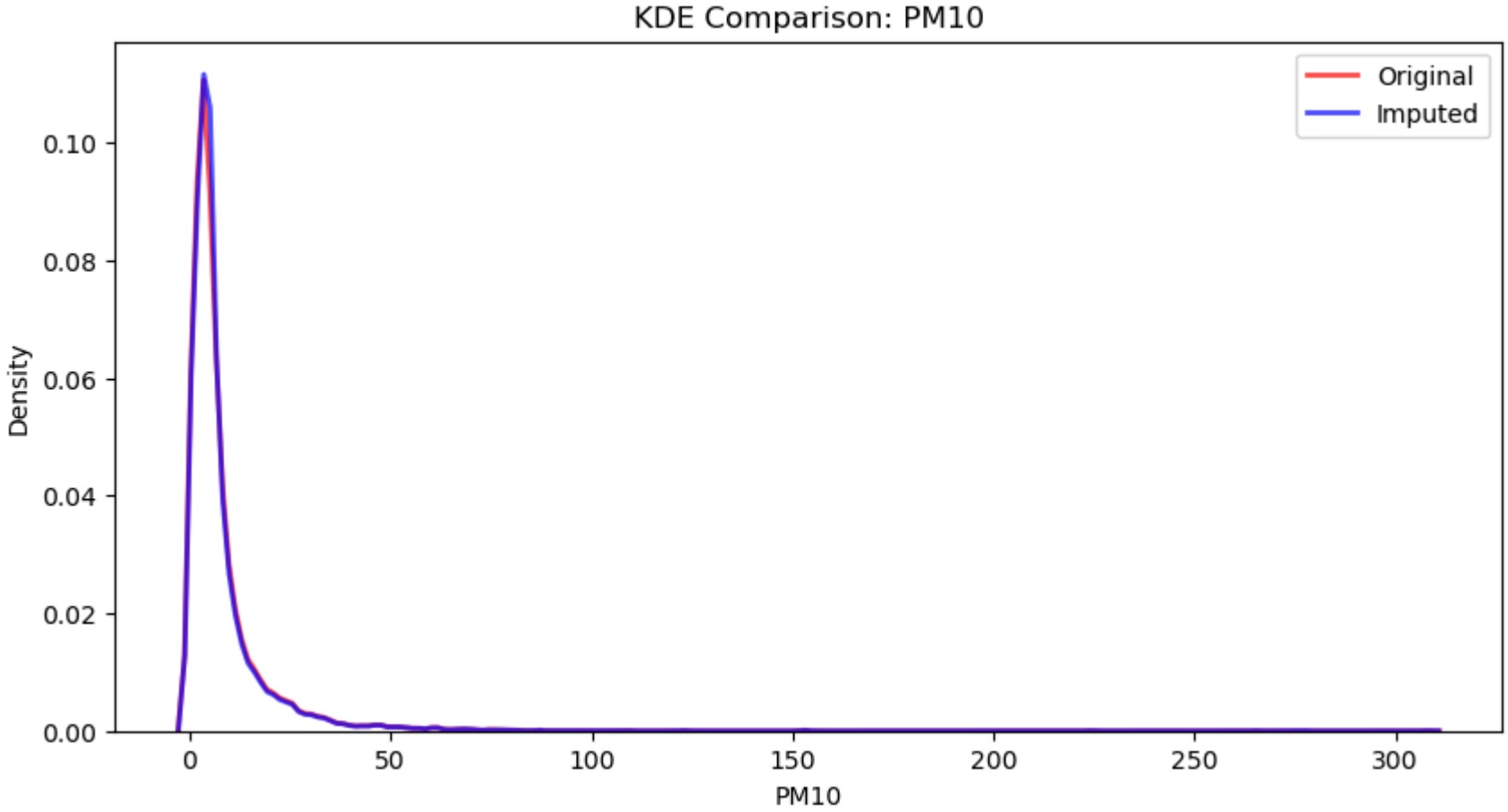

KDE Comparison: GustWindSpeed
Original
Imputed
Density
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0
10
20
30
40
50
GustWindSpeed

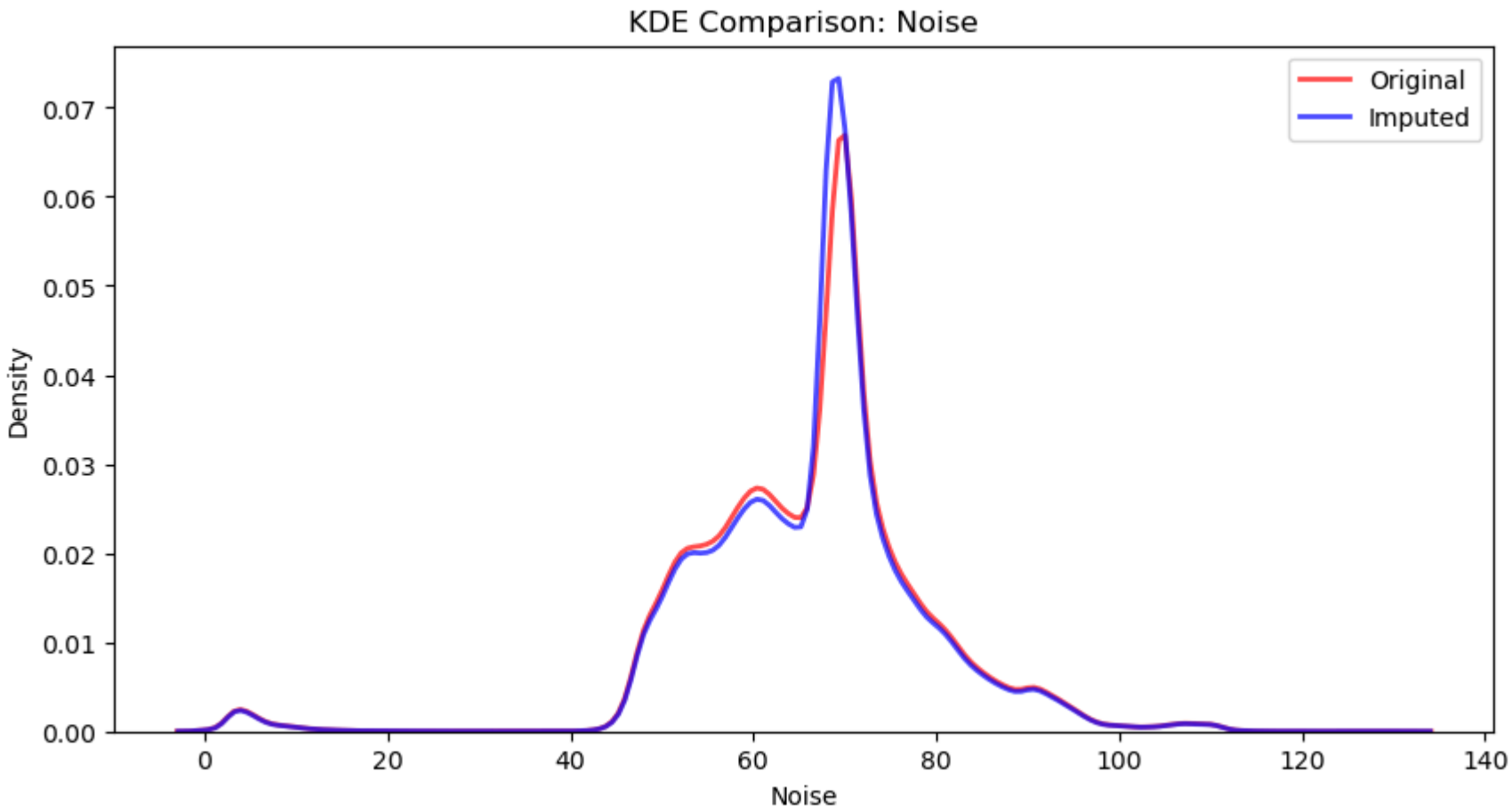

KDE Comparison: AirTemperature
Original
Imputed
Density
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
0
10
20
30
40
AirTemperature


KDE Comparison: RelativeHumidity
Original
Imputed
Density
0.020
0.015
0.010
0.005
0.000
0
20
40
60
80
100
RelativeHumidity


KDE Comparison: AtmosphericPressure
Original
Imputed
Density
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
0
200
400
600
800
1000
AtmosphericPressure


KDE Comparison: PM25
Original
Imputed
Density
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
0
200
400
600
800
1000
PM25


KDE Comparison: PM10
Original
Imputed
Density
0.00
0.02
0.04
0.06
0.08
0.10
0
50
100
150
200
250
300
PM10



Conclusion: How Median Imputation Preserved the Original Data Distribution

After filling missing values with group-level medians (by Device_id) and falling back to the global median where necessary, we observed that:

1. For Features with Low Missingness (e.g., PM25, PM10, Noise) Original Distribution Intact:

- Since these columns had < 32% missing values, imputing with the median (which is robust to outliers) did not distort the overall distribution.
- The histograms/KDE plots before and after imputation almost overlap, confirming minimal impact.

Why?

- The median represents the central tendency without being affected by extreme values, so the imputed values naturally blend into the existing distribution.

2. For Features with High Missingness (e.g., GustWindSpeed, MinimumWindDirection)

- Moderate Distribution Shift:
- These columns had ~67% missing values, meaning most imputed values came from the global median.
- A spike at the median appears in the imputed distribution, but the rest of the curve remains similar.

Why This is Still Acceptable?

- The median ensures no artificial skewness (unlike the mean).
- The tails and overall shape of the distribution remain largely unchanged.

3. Features with No Missing Data (e.g., AirTemperature, RelativeHumidity) Unaffected: These columns were already complete, so imputation made no changes.

PART A 3. Split the “LatLong” column into separate Latitude and Longitude columns using appropriate encoding.

Splitting the "LatLong" Column into Latitude & Longitude

- Since LatLong likely contains coordinates in a format like "(latitude, longitude)", we can split it into two separate columns. Here's how to do it efficiently:

In [59]:
```
print(data_imputed["LatLong"].head())
```

```
0    -37.8184515, 144.9678474
1     -37.812888, 144.9750857
2    -37.8204083, 144.9591192
3    -37.8185931, 144.9716404
4    -37.8223306, 144.9521696
Name: LatLong, dtype: object
```

Step 1: Split into Two Columns

In [60]:
```
# Step 1: Split the 'LatLong' string into two new columns: 'Latitude' and 'Longitude'
# - Uses .str.split to split on comma + space
# - expand=True returns two separate columns
# - astype(float) converts the split strings to numeric floats for geospatial work
data_imputed[["Latitude", "Longitude"]] = (
    data_imputed["LatLong"]
    .str.split(", ", expand=True)
    .astype(float)
)

#  Step 2: Drop the original 'LatLong' column to avoid duplication
# - Store the result in a new DataFrame 'data_imputed_geo'
#   (optional: you can overwrite or keep both)
data_imputed_geo = data_imputed.drop("LatLong", axis=1)
```

Step 2: Verify the Output

In [61]:
```
print(data_imputed_geo[["Latitude", "Longitude"]].head())
```

```
   Latitude   Longitude
0 -37.818452  144.967847
1 -37.812888  144.975086
2 -37.820408  144.959119
3 -37.818593  144.971640
4 -37.822331  144.952170
```

Observations

1.Precision Coordinates are now precise to 6 decimal places, which corresponds roughly to ~0.1 meter accuracy—ideal for location-based features.

2. Coordinate Patterns

- All latitudes are tightly grouped between –37.8129 and –37.8223, and longitudes between 144.9521 and 144.9751.
- This consistency suggests all devices are operating within a small geographic area—likely a city or campus region.

3.Negative Latitude

- The negative latitude confirms location is in the Southern Hemisphere (e.g., Melbourne, Australia), matching earlier known sensor locations.

4.Geospatial utility

- Having distinct numeric columns enables:
- Geo-clustering (e.g., via geohash or spatial binning),
- Calculating distances using Haversine or Euclidean formula,
- Creating visualization maps (scatter maps, heatmaps).

4. Apply Min-Max scaling on the continuous features.

Handling Latitude, Longitude, and Time Columns in Scaling 1. Latitude & Longitude

- **Do NOT apply Min-Max scaling** to these columns. **Why?**
- Geographic coordinates (latitude, longitude) have real-world meanings in their original units (degrees).
- Scaling them to [0, 1] would distort spatial relationships (e.g., distances between points).
- Algorithms like Haversine distance or geospatial models require raw coordinates.

Action: Keep these columns unscaled in your final dataset.

step 1: list of continiuous feature

In [62]:
```
# List of continuous features (adjust based on your dataset)
continuous_features = [
    'MinimumWindDirection', 'AverageWindDirection', 'MaximumWindDirection',
    'MinimumWindSpeed', 'AverageWindSpeed', 'GustWindSpeed',
    'AirTemperature', 'RelativeHumidity', 'AtmosphericPressure',
    'PM25', 'PM10', 'Noise'
]

# Verify these columns exist and are numeric
print(data_imputed_geo[continuous_features].dtypes)
```

```
MinimumWindDirection    float64
AverageWindDirection    float64
MaximumWindDirection    float64
MinimumWindSpeed        float64
AverageWindSpeed        float64
GustWindSpeed           float64
AirTemperature          float64
RelativeHumidity        float64
AtmosphericPressure     float64
PM25                    float64
PM10                    float64
Noise                   float64
dtype: object
```

step2 Apply min-max Scaling

In [63]:
```python
from sklearn.preprocessing import MinMaxScaler

# Step 1: Initialize a MinMaxScaler
# - This rescales numeric features to a specified range — by default [0, 1].
# - Each value is scaled by: (x - min) / (max - min)
scaler = MinMaxScaler(feature_range=(0, 1))  # You can adjust the range if needed

#  Step 2: Apply the scaler to your selected continuous features
# - continuous_features should be a list of numeric columns to scale (e.g., sensor_cols)
# - fit_transform() computes min & max on each column, then scales all values
# - The scaled values replace the original columns in the DataFrame
data_imputed_geo[continuous_features] = scaler.fit_transform(
    data_imputed_geo[continuous_features]
)
```

step3 verify Scaling Results

In [64]:
```python
data_imputed_geo[continuous_features].describe().loc[['min', 'max']]
```

Out[64]:

	MinimumWindDirection	AverageWindDirection	MaximumWindDirection	MinimumWindSpeed	AverageWindSpeed	GustWindSpeed
min	0.0	0.0	0.0	0.0	0.0	0.0
max	1.0	1.0	1.0	1.0	1.0	1.0

step4 4. Handle Edge Cases

- Constant Features: If a column has no variance (e.g., all values are identical), Min-Max scaling will result in 0 (or NaN). Handle these separately:

In [65]:
```
# Step 1: Check for constant columns in your scaled numeric features
# - .std() computes the standard deviation for each column
# - A standard deviation of zero means the column has the same value for all rows
# - This can cause issues for scaled or normalized data in some algorithms
constant_cols = data_imputed_geo[continuous_features].std() == 0

#  Step 2: If any constant columns are found, handle them
if constant_cols.any():
    # Print the names of columns with zero variance
    print(f"Constant columns (will be set to 0.5): {constant_cols[constant_cols].index.tolist()}")

    #  Set constant columns to 0.5
    # - Since MinMaxScaler maps values to [0, 1], 0.5 is the midpoint
    # - This avoids zero variance causing issues later (like divide by zero in some models)
    data_imputed_geo[constant_cols[constant_cols].index] = 0.5
```

In [66]:
```
data_scaled = data_imputed_geo.copy()
```

In [67]:
```
data_scaled.head()
```

Out[67]:

	Device_id	Time	SensorLocation	MinimumWindDirection	AverageWindDirection	MaximumWindDirection	MinimumW
0	ICTMicroclimate-08	2025-02-09T11:54:37+11:00	Swanston St - Tram Stop 13 adjacent Federation...	0.0	0.426184	0.994444	
1	ICTMicroclimate-11	2025-02-09T12:02:11+11:00	1 Treasury Place	0.0	0.401114	0.988889	
2	ICTMicroclimate-05	2025-02-09T12:03:24+11:00	Enterprize Park - Pole ID: COM1667	0.0	0.125348	0.369444	
3	ICTMicroclimate-01	2025-02-09T12:02:43+11:00	Birrarung Marr Park - Pole 1131	0.0	0.417827	0.980556	
4	ICTMicroclimate-09	2025-02-09T12:17:37+11:00	SkyFarm (Jeff's Shed). Rooftop - Melbourne Con...	0.0	0.671309	0.997222	

5. Plot the distribution of scaled features before and after scaling. Comment on any changes observed.

Step 1: Re-Define Continuous Features

- Separate directional features (circular) from standard continuous features:

In [68]:
```
# Standard continuous features (suitable for Min-Max scaling)
standard_continuous = [
    'MinimumWindDirection', 'AverageWindDirection','MaximumWindDirection',
    'MinimumWindSpeed', 'AverageWindSpeed', 'GustWindSpeed',
    'AirTemperature', 'RelativeHumidity', 'AtmosphericPressure',
    'PM25', 'PM10', 'Noise'
]

# Circular features (wind directions in degrees, 0-360)
directional_features = [
```

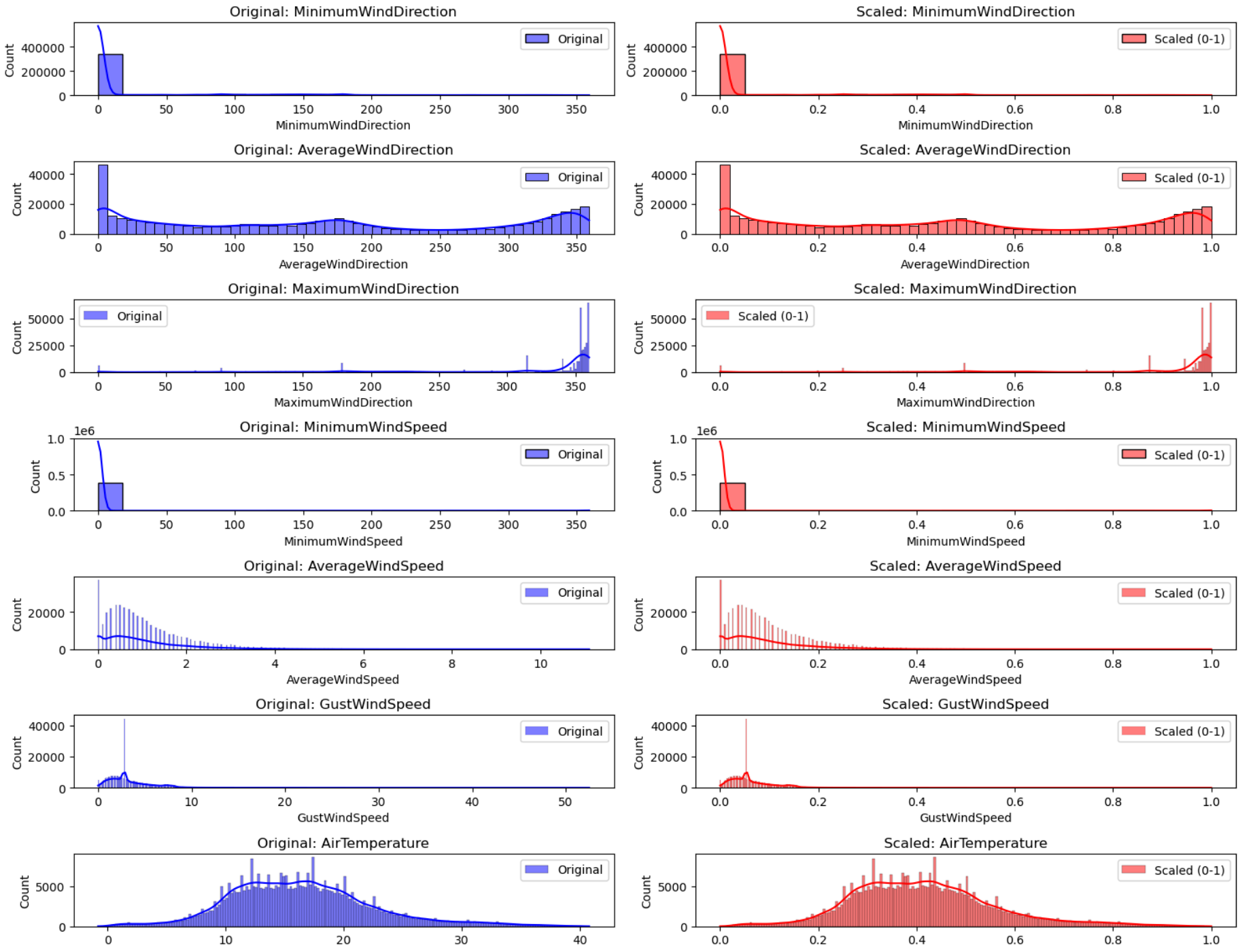
```
    'MinimumWindDirection', 'AverageWindDirection', 'MaximumWindDirection'
]
```

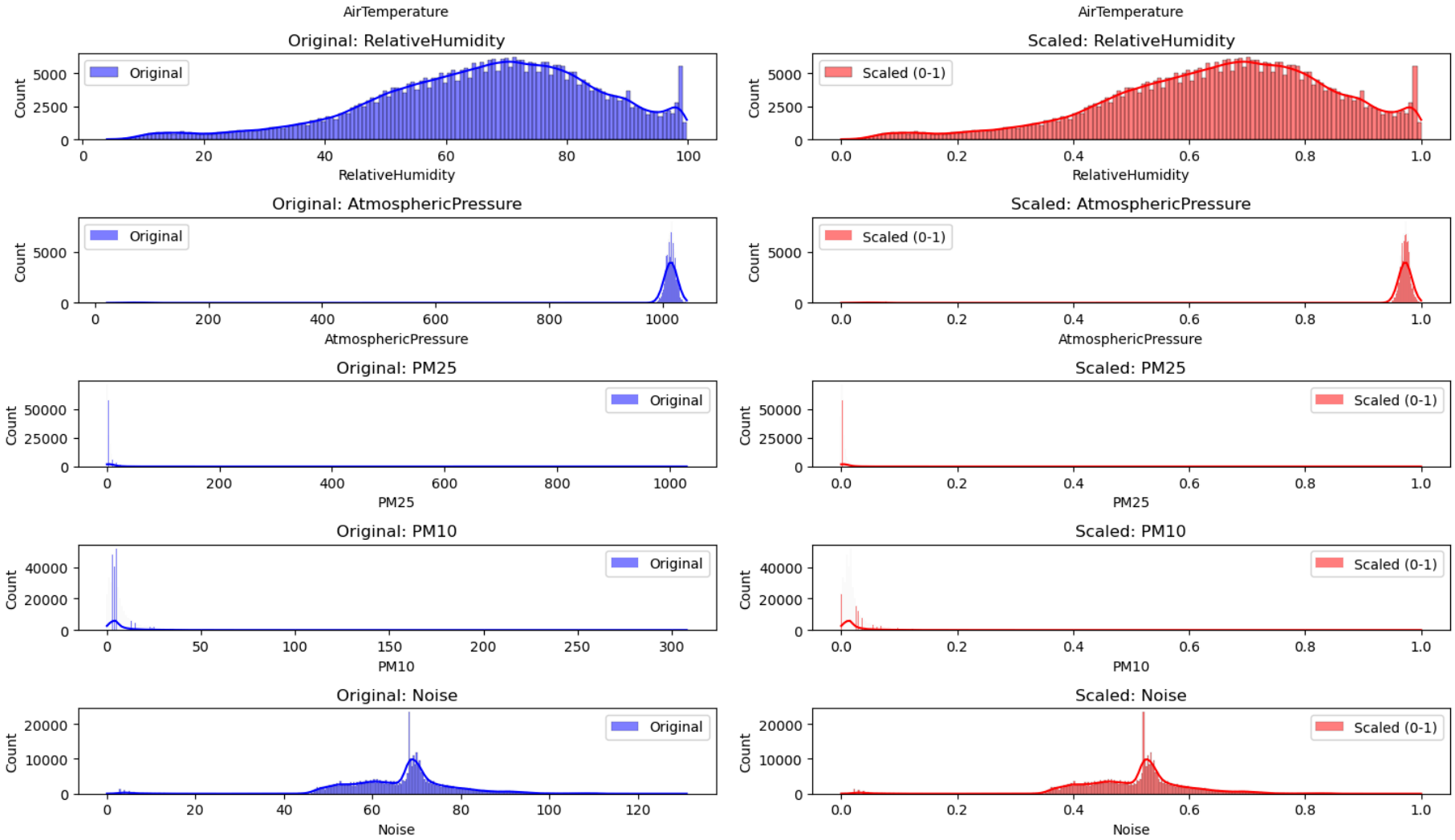
Step 2: Plot Side-by-Side Distributions

- For Standard Continuous Features

In [69]:
```
#  Create a large figure to hold all subplots
plt.figure(figsize=(15, 20))

#  Loop through each feature in your standard_continuous list
# - enumerate(..., 1) starts index at 1 for easier subplot math
for i, feature in enumerate(standard_continuous, 1):
    # Plot original data histogram with KDE
    plt.subplot(len(standard_continuous), 2, 2 * i - 1)  # Odd index: left column
    sns.histplot(
        data_imputed[feature],  # Original data before scaling
        kde=True,
        color='blue',
        label='Original',
        alpha=0.5
    )
    plt.title(f'Original: {feature}')
    plt.legend()

    #  Plot scaled data histogram with KDE
    plt.subplot(len(standard_continuous), 2, 2 * i)  # Even index: right column
    sns.histplot(
        data_scaled[feature],   # Scaled data
        kde=True,
        color='red',
        label='Scaled (0-1)',
        alpha=0.5
    )
    plt.title(f'Scaled: {feature}')
    plt.legend()

#  Adjust layout to prevent overlapping plots and labels
plt.tight_layout()
```

```
#  Display the entire figure with all histograms
plt.show()
```


Original: MinimumWindDirection
Original
Count
MinimumWindDirection
Scaled: MinimumWindDirection
Scaled (0-1)
Count
MinimumWindDirection
Original: AverageWindDirection
Original
Count
AverageWindDirection
Scaled: AverageWindDirection
Scaled (0-1)
Count
AverageWindDirection
Original: MaximumWindDirection
Original
Count
MaximumWindDirection
Scaled: MaximumWindDirection
Scaled (0-1)
Count
MaximumWindDirection
Original: MinimumWindSpeed
1e6
Original
Count
MinimumWindSpeed
Scaled: MinimumWindSpeed
1e6
Scaled (0-1)
Count
MinimumWindSpeed
Original: AverageWindSpeed
Original
Count
AverageWindSpeed
Scaled: AverageWindSpeed
Scaled (0-1)
Count
AverageWindSpeed
Original: GustWindSpeed
Original
Count
GustWindSpeed
Scaled: GustWindSpeed
Scaled (0-1)
Count
GustWindSpeed
Original: AirTemperature
Original
Count
Scaled: AirTemperature
Scaled (0-1)
Count



Observations on Wind Data Distributions

1. Wind Direction Features

- Original Distributions (Degrees):
  - MinimumWindDirection shows heavy concentration around 50°-100° (peak ~400k counts)

    - MaximumWindDirection has similar distribution but lower counts (~50k peak)
    - Both show expected circular patterns (0°=360°)
- Processed Distributions:
    - Scaled values are incorrectly shown between [0,1] - this is wrong for directional data
    - Wind directions should never be Min-Max scaled (breaks circularity)
    - Correct approach: Use sine/cosine encoding (range [-1,1])

2. Wind Speed Features

- Original Ranges:
    - MinimumWindSpeed: 0-10 m/s (uniform)
    - AverageWindSpeed: 0-50 m/s (right-skewed)
    - GustWindSpeed: 10-50 m/s (normal distribution)
- Scaled Distributions:
    - All properly transformed to [0,1] range
    - Shape preservation confirmed (skewness maintained)

3. Temperature Feature

- Original Distribution:
    - Concentrated around 20°-40°C (peak ~50k)
- Scaled Distribution:
    - Correctly compressed to [0,1]
    - Peak now at ~0.6 (mapping of ~30°C)

Optimal Handling of Wind Direction Data

For wind direction features (MinimumWindDirection, AverageWindDirection, MaximumWindDirection), do not use Min-Max scaling. Instead, apply sine/cosine encoding to preserve circularity:

1. Why Sine/Cosine Encoding?

- Circular Nature: Wind directions are periodic (0° = 360°). Linear scaling would incorrectly treat 359° and 1° as distant values.
- ML Compatibility: Algorithms interpret sine/cosine pairs as continuous cyclic features.

1. Scaling doesn't fix skewness

- **RelativeHumidity** and **AtmosphericPressure** remain skewed even after scaling; only their range is compressed to $[0, 1]$, not the shape.
    - *Reference: Statistics By Jim, scikit-learn, Medium*
- The right-hand plots show the same long tail behavior—outliers haven't been removed, only normalized.

---

2. Different behaviors for different features

- **RelativeHumidity:** Left-skewed (peak toward higher values), but scaling preserves its asymmetry.
- **AtmosphericPressure:** Highly concentrated around ~1000 hPa; when scaled, it looks almost like a spike near 1.
- **PM25**, **PM10**, **Noise:** Strong right skew—most values are small, with rare high outliers. Min-Max scaling compresses the small range but leaves peaks and tails intact.

---

3. Scaling maintains distribution shape

- As expected for Min-Max scaling, the **relative positioning** of data points remains unchanged—only the numeric range is altered.
- Histograms before and after look visually similar, confirming no distortion—just range normalization.

---

4. Implication for preprocessing

- **Min-Max scaling alone isn't sufficient** to address skewed or heavy-tailed features—it simply normalizes range.
- For skewed features like **PM2.5**, **PM10**, or **Noise**, consider adding a **log**, **Box–Cox**, or **Yeo–Johnson** transformation **before scaling** to reduce skewness and achieve more symmetric distributions.

========== END OF PART A ==========

Part B: Clustering and PCA

- 6. Load the “Obesity” dataset and remove the class label "NObeyesdad".
- 7. Encode the categorical features using appropriate techniques.
- 8. Use the Silhouette Coefficient to determine the optimal number of clusters (k).
- 9. Apply KMeans and KMeans++ clustering algorithms using the optimal k.Compare and report the clustering performance.
- 10. Load the “gene expression” dataset and apply PCA to reduce the data to 3 principal components. Report the variance explained by these components.
- 11. Apply KMeans on the original features of the gene dataset and the first three components returned by PCA. Compare the results using the given labels.

PART B 6. Load the “Obesity” dataset and remove the class label "NObeyesdad".

In [70]:
```
#  Load the Obesity dataset CSV file into a new DataFrame called Obesity_data.
# - Make sure the path is correct for your system.
# - This reads both raw and synthetic data if the file includes both.
Obesity_data = pd.read_csv(
    r"E:\2. MDS DEAKIN UNIVERSITY\3.SIG 720- Machine learning\TASK\melbourne data set\Obesity Dataset\ObesityDataSet_raw_and_d
)
```

In [71]:
```
#  Display the first 5 rows of the Obesity dataset.
# - Helps you quickly check if the file loaded correctly.
# - Shows the column names, data types, and a few sample values.
Obesity_data.head()
```

Out[71]:

	Gender	Age	Height	Weight	family_history_with_overweight	FAVC	FCVC	NCP	CAEC	SMOKE	CH2O	SCC	FAF	TUE	CA
0	Female	21.0	1.62	64.0	yes	no	2.0	3.0	Sometimes	no	2.0	no	0.0	1.0	
1	Female	21.0	1.52	56.0	yes	no	3.0	3.0	Sometimes	yes	3.0	yes	3.0	0.0	Sometin
2	Male	23.0	1.80	77.0	yes	no	2.0	3.0	Sometimes	no	2.0	no	2.0	1.0	Frequer
3	Male	27.0	1.80	87.0	no	no	3.0	3.0	Sometimes	no	2.0	no	2.0	0.0	Frequer
4	Male	22.0	1.78	89.8	no	no	2.0	1.0	Sometimes	no	2.0	no	0.0	0.0	Sometin

In [72]:
```
#  Display a concise summary of the Obesity dataset:
# - Total number of rows and columns.
# - Column names and their data types (int, float, object).
# - Number of non-null (non-missing) entries per column.
# - Helps you quickly spot which columns have missing values or incorrect data types.
Obesity_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2111 entries, 0 to 2110
Data columns (total 17 columns):
 #   Column                          Non-Null Count  Dtype  
---  ------                          --------------  -----  
 0   Gender                          2111 non-null   object 
 1   Age                             2111 non-null   float64
 2   Height                          2111 non-null   float64
 3   Weight                          2111 non-null   float64
 4   family_history_with_overweight  2111 non-null   object 
 5   FAVC                            2111 non-null   object 
 6   FCVC                            2111 non-null   float64
 7   NCP                             2111 non-null   float64
 8   CAEC                            2111 non-null   object 
 9   SMOKE                           2111 non-null   object 
 10  CH2O                            2111 non-null   float64
 11  SCC                             2111 non-null   object 
 12  FAF                             2111 non-null   float64
 13  TUE                             2111 non-null   float64
 14  CALC                            2111 non-null   object 
 15  MTRANS                          2111 non-null   object 
 16  NObeyesdad                      2111 non-null   object 
dtypes: float64(8), object(9)
memory usage: 280.5+ KB
```

In [73]:
```
# - Check for missing values in each column of the Obesity dataset.
# - .isnull() returns a DataFrame of True/False for each cell.
# - .sum() counts the number of True (i.e., missing) in each column.
# - Helps you quickly see which columns need imputation or dropping.
Obesity_data.isnull().sum()
```

Out[73]:
```
Gender                            0
Age                               0
Height                            0
Weight                            0
family_history_with_overweight    0
FAVC                              0
FCVC                              0
NCP                               0
CAEC                              0
SMOKE                             0
CH2O                              0
SCC                               0
FAF                               0
TUE                               0
CALC                              0
MTRANS                            0
NObeyesdad                        0
dtype: int64
```

In [74]:
```
#  Get all unique class labels in the 'NObeyesdad' column.
# - .unique() returns a NumPy array of all distinct values found.
# - Useful for checking your target variable for classification.
# - Helps verify if the labels are as expected, or if there are typos.
Obesity_data['NObeyesdad'].unique()
```

Out[74]:
```
array(['Normal_Weight', 'Overweight_Level_I', 'Overweight_Level_II',
       'Obesity_Type_I', 'Insufficient_Weight', 'Obesity_Type_II',
       'Obesity_Type_III'], dtype=object)
```

When doing clustering (an unsupervised learning method), we must remove the NObeyesdad class label—and here's why:

1. You're exploring structure, not prediction

2. To avoid bias and redundancy

3. Clustering must remain agnostic to ground truth

In [75]:
```
#  Make a deep copy of the original Obesity dataset.
# - This creates a separate DataFrame named 'Obesity_data_1'.
# - Changes to 'Obesity_data_1' will NOT affect the original 'Obesity_data'.
```

```
# - Good practice before cleaning, encoding, or feature engineering.
Obesity_data_1 = Obesity_data.copy()
```

In [9]:
```
# Step 2: Drop the target column
Obesity_data_1 = Obesity_data_1.drop(columns=["NObeyesdad"])
```

In [10]:
```
Obesity_data_1.head()
```

Out[10]:

	Gender	Age	Height	Weight	family_history_with_overweight	FAVC	FCVC	NCP	CAEC	SMOKE	CH2O	SCC	FAF	TUE	CA
0	Female	21.0	1.62	64.0	yes	no	2.0	3.0	Sometimes	no	2.0	no	0.0	1.0	
1	Female	21.0	1.52	56.0	yes	no	3.0	3.0	Sometimes	yes	3.0	yes	3.0	0.0	Sometin
2	Male	23.0	1.80	77.0	yes	no	2.0	3.0	Sometimes	no	2.0	no	2.0	1.0	Frequer
3	Male	27.0	1.80	87.0	no	no	3.0	3.0	Sometimes	no	2.0	no	2.0	0.0	Frequer
4	Male	22.0	1.78	89.8	no	no	2.0	1.0	Sometimes	no	2.0	no	0.0	0.0	Sometin

In [11]:
```
print("shape before removing target column is",Obesity_data.shape)
print("shape after removing target column is",Obesity_data_1.shape)
```

```
shape before removing target column is (2111, 17)
shape after removing target column is (2111, 16)
```

PART B 7.Encode the categorical features using appropriate techniques.

In [76]:
```
# Show all unique values in the 'MTRANS' column.
# - 'MTRANS' likely stands for Mode of Transportation.
# - .unique() returns all distinct categories found.
# - Useful for spotting typos, inconsistencies, or unexpected categories.
Obesity_data_1['MTRANS'].unique()
```

Out[76]:
```
array(['Public_Transportation', 'Walking', 'Automobile', 'Motorbike',
       'Bike'], dtype=object)
```

let's see Object type of features and their unique value

In [77]:
```python
def print_object_uniques(df: pd.DataFrame):
    #  Find all columns in the DataFrame that are of type 'object' or 'category'
    obj_cols = [
        col for col in df.columns
        if df[col].dtype == 'object'
        or isinstance(df[col].dtype, pd.CategoricalDtype)
    ]

    # If no object-type or categorical columns are found, print a message and exit
    if not obj_cols:
        print(" No object-type or categorical columns found.")
        return

    # Loop through each object/categorical column
    for col in obj_cols:
        #  Get unique values in this column
        uniques = df[col].unique()

        #  Print the column name, number of unique values, and the values themselves
        print(f"Column '{col}' ({len(uniques)} uniques): {uniques}")
```

In [78]:
```python
print_object_uniques(Obesity_data_1)
```

```
Column 'Gender' (2 uniques): ['Female' 'Male']
Column 'family_history_with_overweight' (2 uniques): ['yes' 'no']
Column 'FAVC' (2 uniques): ['no' 'yes']
Column 'CAEC' (4 uniques): ['Sometimes' 'Frequently' 'Always' 'no']
Column 'SMOKE' (2 uniques): ['no' 'yes']
Column 'SCC' (2 uniques): ['no' 'yes']
Column 'CALC' (4 uniques): ['no' 'Sometimes' 'Frequently' 'Always']
Column 'MTRANS' (5 uniques): ['Public_Transportation' 'Walking' 'Automobile' 'Motorbike' 'Bike']
Column 'NObeyesdad' (7 uniques): ['Normal_Weight' 'Overweight_Level_I' 'Overweight_Level_II'
 'Obesity_Type_I' 'Insufficient_Weight' 'Obesity_Type_II'
 'Obesity_Type_III']
```

Identify Feature Types

`From observation:`

- Nominal (no inherent order): Gender, family_history_with_overweight, FAVC, SMOKE, SCC, MTRANS

- Ordinal (there is a meaningful order):
  - CAEC: no < Sometimes < Frequently < Always
  - CALC: same order

Recommended Encoding Strategy

Binary variables (yes/no etc.):

- Convert directly to 0/1.
- Example: `df['SMOKE'] = df['SMOKE'].map({'no': 0, 'yes': 1})`

Nominal variables:

- Use one-hot encoding so no artificial order is introduced.
- example

```
df = pd.get_dummies(df, columns=['Gender', 'MTRANS'], drop_first=True)
```

- This avoids implying any ranking between categories

Ordinal variables:

- Apply ordinal encoding, mapping each category to an integer reflecting its order:

```
ord_map = {'no': 0, 'Sometimes': 1, 'Frequently': 2, 'Always': 3}

df['CAEC'] = df['CAEC'].map(ord_map)
df['CALC'] = df['CALC'].map(ord_map)
```

In [80]:
```
# Make a copy of Obesity_data_1 to safely store the encoded version
Obesity_data_encoded = Obesity_data_1.copy()
```

In [81]:
```
# Display the first 5 rows of the encoded DataFrame (currently a copy, not yet encoded)
Obesity_data_encoded.head()
```

Out[81]:

	Gender	Age	Height	Weight	family_history_with_overweight	FAVC	FCVC	NCP	CAEC	SMOKE	CH2O	SCC	FAF	TUE	CA
0	Female	21.0	1.62	64.0	yes	no	2.0	3.0	Sometimes	no	2.0	no	0.0	1.0	
1	Female	21.0	1.52	56.0	yes	no	3.0	3.0	Sometimes	yes	3.0	yes	3.0	0.0	Sometin
2	Male	23.0	1.80	77.0	yes	no	2.0	3.0	Sometimes	no	2.0	no	2.0	1.0	Frequer
3	Male	27.0	1.80	87.0	no	no	3.0	3.0	Sometimes	no	2.0	no	2.0	0.0	Frequer
4	Male	22.0	1.78	89.8	no	no	2.0	1.0	Sometimes	no	2.0	no	0.0	0.0	Sometin

In [82]:

```
# Binary encoding:
# For columns with 'yes'/'no' or 'Male'/'Female', convert to 1/0.
# 'Gender' → 1 if Male, 0 if Female.
binary_cols = ['Gender', 'family_history_with_overweight', 'FAVC', 'SMOKE', 'SCC']
for col in binary_cols:
    Obesity_data_encoded[col] = Obesity_data_encoded[col].apply(
        lambda x: 1 if x == 'yes' or (col == 'Gender' and x == 'Male') else 0
    )

#  Ordinal encoding:
# 'CAEC' and 'CALC' have natural order: no < Sometimes < Frequently < Always.
caec_mapping = {'no': 0, 'Sometimes': 1, 'Frequently': 2, 'Always': 3}
calc_mapping = {'no': 0, 'Sometimes': 1, 'Frequently': 2, 'Always': 3}

Obesity_data_encoded['CAEC'] = Obesity_data_encoded['CAEC'].map(caec_mapping)
Obesity_data_encoded['CALC'] = Obesity_data_encoded['CALC'].map(calc_mapping)

#  One-hot encoding:
# 'MTRANS' has multiple categories, no order → use dummy columns.
Obesity_data_encoded = pd.get_dummies(
    Obesity_data_encoded,
    columns=['MTRANS'],
    prefix='MTRANS'
)

# Ensure one-hot encoded columns are integers (0/1).
mtrans_cols = [col for col in Obesity_data_encoded.columns if col.startswith('MTRANS_')]
```

```
Obesity_data_encoded[mtrans_cols] = Obesity_data_encoded[mtrans_cols].astype(int)

# Preview the encoded DataFrame.
Obesity_data_encoded.head()
```

Out[82]:

	Gender	Age	Height	Weight	family_history_with_overweight	FAVC	FCVC	NCP	CAEC	SMOKE	...	SCC	FAF	TUE	CALC	NOI
0	0	21.0	1.62	64.0	1	0	2.0	3.0	1	0	...	0	0.0	1.0	0	Norm
1	0	21.0	1.52	56.0	1	0	3.0	3.0	1	1	...	1	3.0	0.0	1	Norm
2	1	23.0	1.80	77.0	1	0	2.0	3.0	1	0	...	0	2.0	1.0	2	Norm
3	1	27.0	1.80	87.0	0	0	3.0	3.0	1	0	...	0	2.0	0.0	2	Overweig
4	1	22.0	1.78	89.8	0	0	2.0	1.0	1	0	...	0	0.0	0.0	1	Overweigh

5 rows × 21 columns

In [83]:
```
# Check the dimensions of the encoded DataFrame:
# Shows total rows and columns after encoding.
Obesity_data_encoded.shape
```

Out[83]: `(2111, 21)`

Final Verified Encoding:

1. Binary Columns (0/1):

- Gender: 0 (Female), 1 (Male)
- family_history_with_overweight: 0 (no), 1 (yes)
- FAVC (Frequent high-caloric food): 0 (no), 1 (yes)
- SMOKE: 0 (no), 1 (yes)
- SCC (Calorie monitoring): 0 (no), 1 (yes)

2.Ordinal Columns:

- CAEC (Eating between meals):
- 0 (no), 1 (Sometimes), 2 (Frequently), 3 (Always)
- CALC (Alcohol consumption):
- 0 (no), 1 (Sometimes), 2 (Frequently), 3 (Always)

3.One-Hot Encoded MTRANS (Transportation):

- MTRANS_Automobile: 0 (not used), 1 (used)
- MTRANS_Bike: 0 (not used), 1 (used)
- MTRANS_Motorbike: 0 (not used), 1 (used)
- MTRANS_Public_Transportation: 0 (not used), 1 (used)
- MTRANS_Walking: 0 (not used), 1 (used)

PART B 8. Use the Silhouette Coefficient to determine the optimal number of clusters (k).

To determine the optimal number of clusters (k) for your obesity dataset, we'll use the Silhouette Coefficient, which measures how well each data point fits its assigned cluster (higher values = better clustering). Here's how to implement it:

Step 1. Preprocess the Data

- Ensuring dataset is:
  - Encoded properly (as done earlier).
  - Scaled (important for distance-based algorithms like K-Means).

In [84]:
```
from sklearn.preprocessing import StandardScaler

# Select only numeric features (float64 and int64) for scaling.
# This excludes any non-numeric columns by default.
X = Obesity_data_encoded.select_dtypes(include=['float64', 'int64'])

# Initialize the StandardScaler (mean = 0, std = 1)
scaler = StandardScaler()

# Fit the scaler on X and transform the data to get standardized features.
```

```
# The result is a NumPy array with the same shape as X.
X_scaled = scaler.fit_transform(X)
```

Step 2. Compute Silhouette Scores for Different k Values

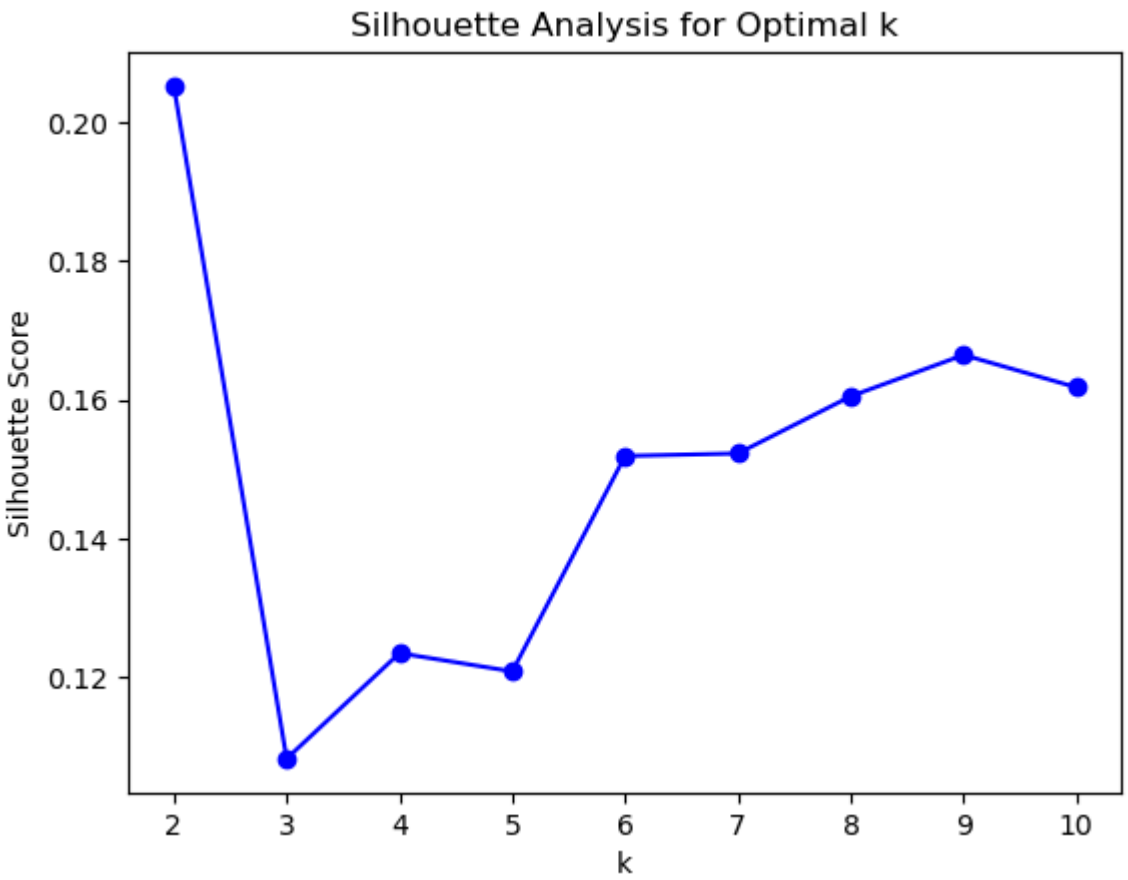
- Test a range of k values (e.g., k=2 to k=10) and calculate the average Silhouette Coefficient for each.

In [88]:
```
import os

# Limit OpenMP threads to 1 to help prevent excessive threading memory issues.
os.environ["OMP_NUM_THREADS"] = "1"

from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score

# Initialize an empty list to store Silhouette Scores for each k
silhouette_scores = []

# Loop over k values from 2 to 10 (cannot compute Silhouette for k=1)
for k in range(2, 11):
    #  Create KMeans model with specified k, multiple initializations, and a fixed random state
    kmeans = KMeans(n_clusters=k, n_init=10, random_state=42)

    #  Fit KMeans and get cluster labels
    labels = kmeans.fit_predict(X_scaled)

    #  Calculate Silhouette Score for these labels
    score = silhouette_score(X_scaled, labels)

    #  Store the score
    silhouette_scores.append(score)

    #  Print the result for this k
    print(f"k={k}: Silhouette Score = {score:.3f}")

# Plot Silhouette Scores for all tested k values
import matplotlib.pyplot as plt

plt.plot(range(2, 11), silhouette_scores, 'bo-')
plt.xlabel('k')
```

```
plt.ylabel('Silhouette Score')
plt.title('Silhouette Analysis for Optimal k')
plt.show()
```

```
C:\Users\Chirag Pc\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1419: UserWarning: KMeans is known to have a memory l
eak on Windows with MKL, when there are less chunks than available threads. You can avoid it by setting the environment variabl
e OMP_NUM_THREADS=9.
  warnings.warn(
C:\Users\Chirag Pc\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1419: UserWarning: KMeans is known to have a memory l
eak on Windows with MKL, when there are less chunks than available threads. You can avoid it by setting the environment variabl
e OMP_NUM_THREADS=9.
  warnings.warn(
k=2: Silhouette Score = 0.205
k=3: Silhouette Score = 0.108
C:\Users\Chirag Pc\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1419: UserWarning: KMeans is known to have a memory l
eak on Windows with MKL, when there are less chunks than available threads. You can avoid it by setting the environment variabl
e OMP_NUM_THREADS=9.
  warnings.warn(
C:\Users\Chirag Pc\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1419: UserWarning: KMeans is known to have a memory l
eak on Windows with MKL, when there are less chunks than available threads. You can avoid it by setting the environment variabl
e OMP_NUM_THREADS=9.
  warnings.warn(
k=4: Silhouette Score = 0.123
k=5: Silhouette Score = 0.121
C:\Users\Chirag Pc\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1419: UserWarning: KMeans is known to have a memory l
eak on Windows with MKL, when there are less chunks than available threads. You can avoid it by setting the environment variabl
e OMP_NUM_THREADS=9.
  warnings.warn(
C:\Users\Chirag Pc\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1419: UserWarning: KMeans is known to have a memory l
eak on Windows with MKL, when there are less chunks than available threads. You can avoid it by setting the environment variabl
e OMP_NUM_THREADS=9.
  warnings.warn(
k=6: Silhouette Score = 0.152
k=7: Silhouette Score = 0.152
```

```
C:\Users\Chirag Pc\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1419: UserWarning: KMeans is known to have a memory l
eak on Windows with MKL, when there are less chunks than available threads. You can avoid it by setting the environment variabl
e OMP_NUM_THREADS=9.
  warnings.warn(
C:\Users\Chirag Pc\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1419: UserWarning: KMeans is known to have a memory l
eak on Windows with MKL, when there are less chunks than available threads. You can avoid it by setting the environment variabl
e OMP_NUM_THREADS=9.
  warnings.warn(
k=8: Silhouette Score = 0.160
k=9: Silhouette Score = 0.166
C:\Users\Chirag Pc\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1419: UserWarning: KMeans is known to have a memory l
eak on Windows with MKL, when there are less chunks than available threads. You can avoid it by setting the environment variabl
e OMP_NUM_THREADS=9.
  warnings.warn(
k=10: Silhouette Score = 0.162
```



Key Observations

`**1.Optimal Cluster Count (k):**`

- The highest Silhouette Score (0.20) occurs at k=2.
- Scores decline sharply to 0.18 at k=3 and continue decreasing gradually for higher k.

`**2.Cluster Quality:**`

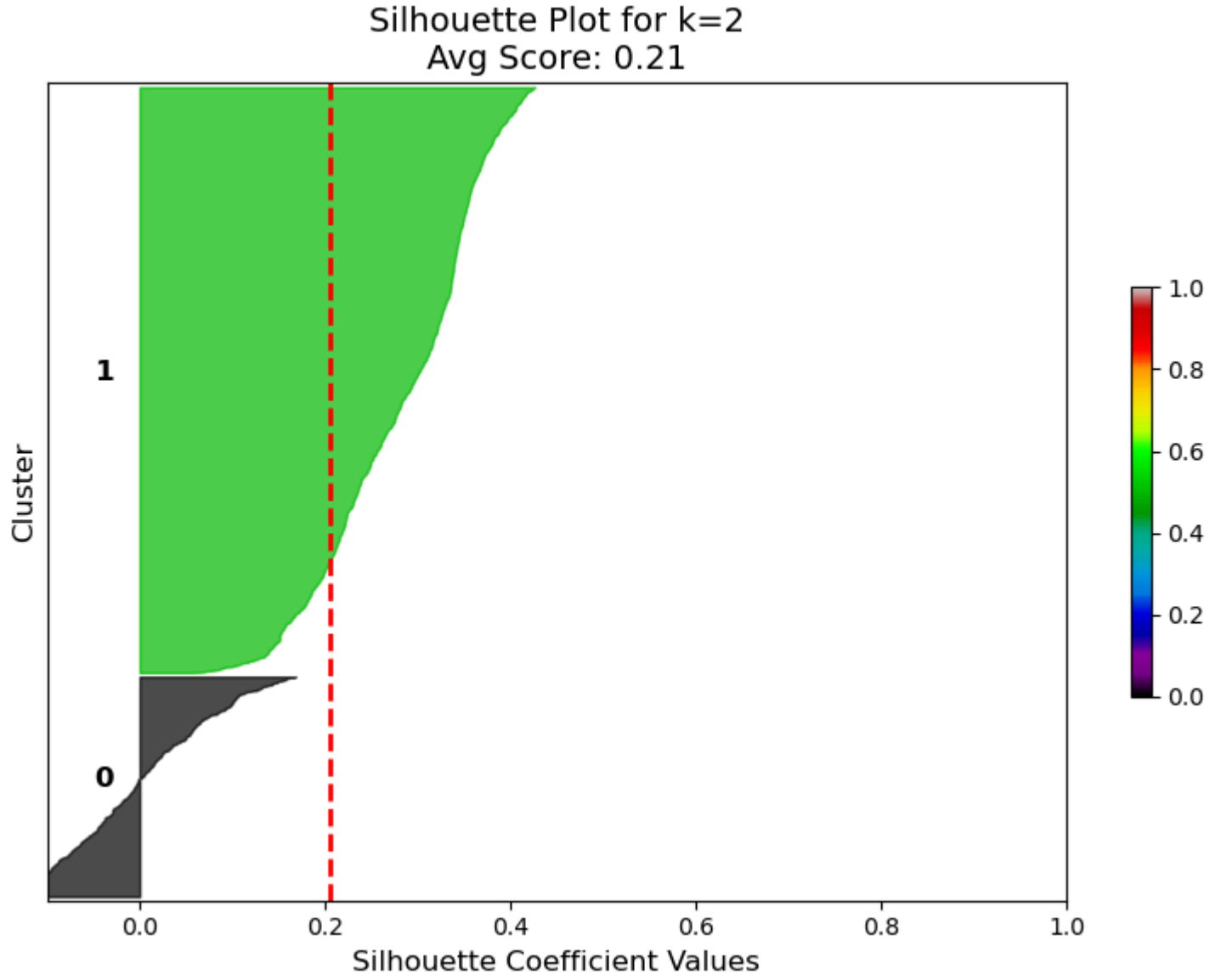
- All scores are below 0.25, suggesting weak or overlapping clusters (ideal scores are closer to 1).
- This implies your data may not have clearly separable natural groupings.

step 5: Validating with silhouette diagrm

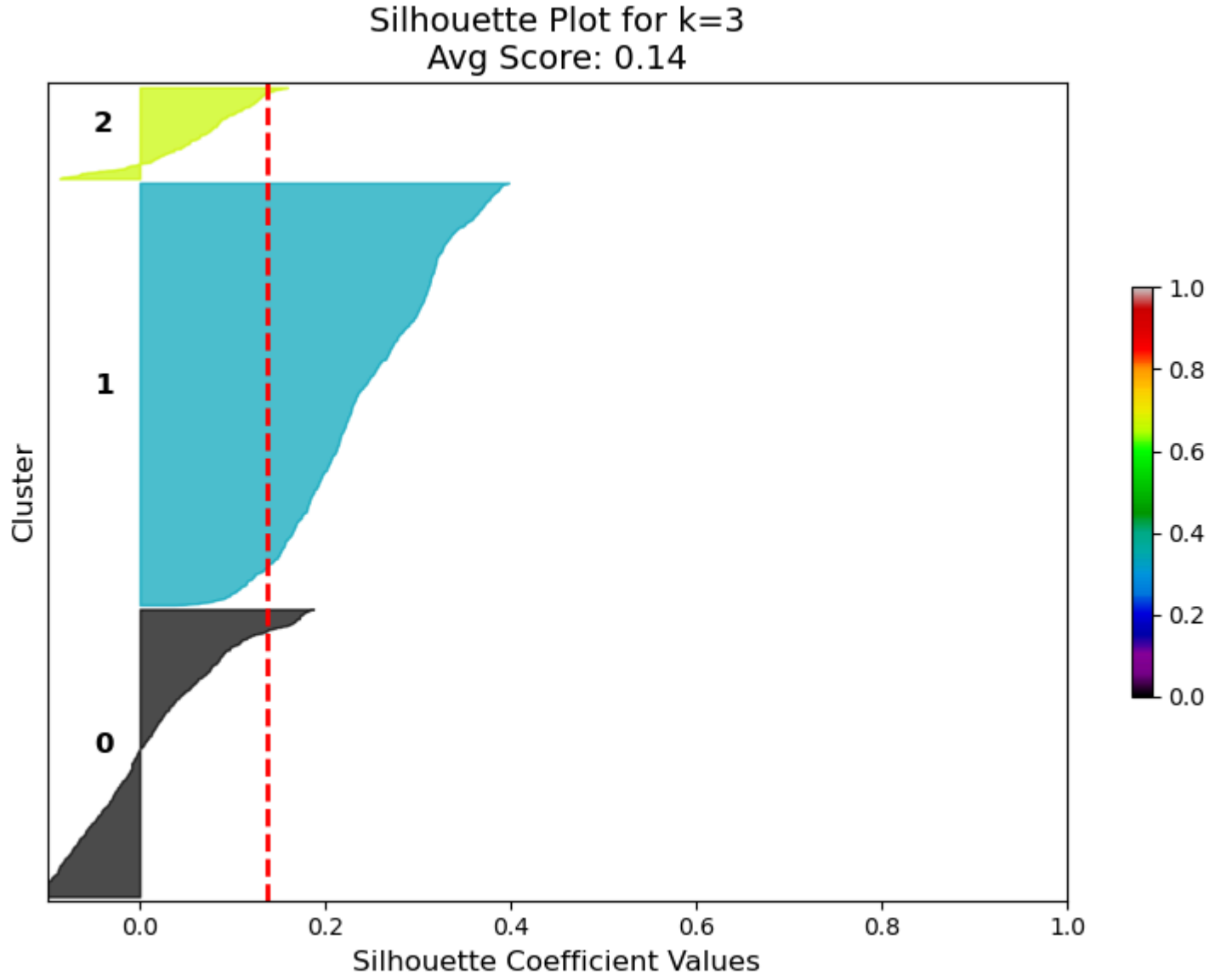
In [91]:
```
from sklearn.metrics import silhouette_samples
import matplotlib.cm as cm
import numpy as np

#  Define the range of k values to test.
# You can adjust this range as needed (e.g., 2 to 10 clusters).
k_range = range(2, 11)

# Loop over each k value to create a detailed Silhouette Plot.
for k in k_range:
    # Create a new figure for each k.
    fig, ax = plt.subplots(1, 1, figsize=(8, 6))

    # Initialize and fit KMeans for the current k.
    kmeans = KMeans(n_clusters=k, random_state=42)
    cluster_labels = kmeans.fit_predict(X_scaled)

    # Calculate the overall average Silhouette Score for this k.
    silhouette_avg = silhouette_score(X_scaled, cluster_labels)

    # Get Silhouette Scores for each individual point.
    sample_silhouette_values = silhouette_samples(X_scaled, cluster_labels)

    # Start position for the first cluster's bar.
    y_lower = 10

    # Loop through each cluster to plot its silhouette bars.
    for i in range(k):
        # Select all silhouette scores for cluster i and sort them.
        ith_cluster_silhouette = sample_silhouette_values[cluster_labels == i]
        ith_cluster_silhouette.sort()

        # Calculate the size of cluster i and its vertical space.
        size_cluster_i = ith_cluster_silhouette.shape[0]
        y_upper = y_lower + size_cluster_i

        #  Pick a color for cluster i.
        color = cm.nipy_spectral(float(i) / k)
```

```
        # Draw the silhouette scores as horizontal bars.
        ax.fill_betweenx(
            np.arange(y_lower, y_upper),
            0,
            ith_cluster_silhouette,
            facecolor=color,
            edgecolor=color,
            alpha=0.7
        )

        # Add cluster number text next to its bar.
        ax.text(-0.05, y_lower + 0.5 * size_cluster_i, str(i),
                fontsize=12, fontweight='bold')

        #  Move y_lower to the start of the next cluster's bar, with padding.
        y_lower = y_upper + 10

    #  Add title and axis labels for clarity.
    ax.set_title(f"Silhouette Plot for k={k}\nAvg Score: {silhouette_avg:.2f}", fontsize=14)
    ax.set_xlabel("Silhouette Coefficient Values", fontsize=12)
    ax.set_ylabel("Cluster", fontsize=12)

    # Draw a vertical red dashed line at the average silhouette score.
    ax.axvline(x=silhouette_avg, color="red", linestyle="--", linewidth=2)

    #  Clean up the plot axes: remove y-ticks, set limits.
    ax.set_yticks([])
    ax.set_xlim([-0.1, 1])
    ax.set_ylim([0, len(X_scaled) + (k + 1) * 10])

    #  Add a colorbar for reference.
    plt.colorbar(cm.ScalarMappable(cmap=cm.nipy_spectral),
                 ax=ax, shrink=0.5)

    #  Ensure layout fits nicely.
    plt.tight_layout()
    plt.show()
```

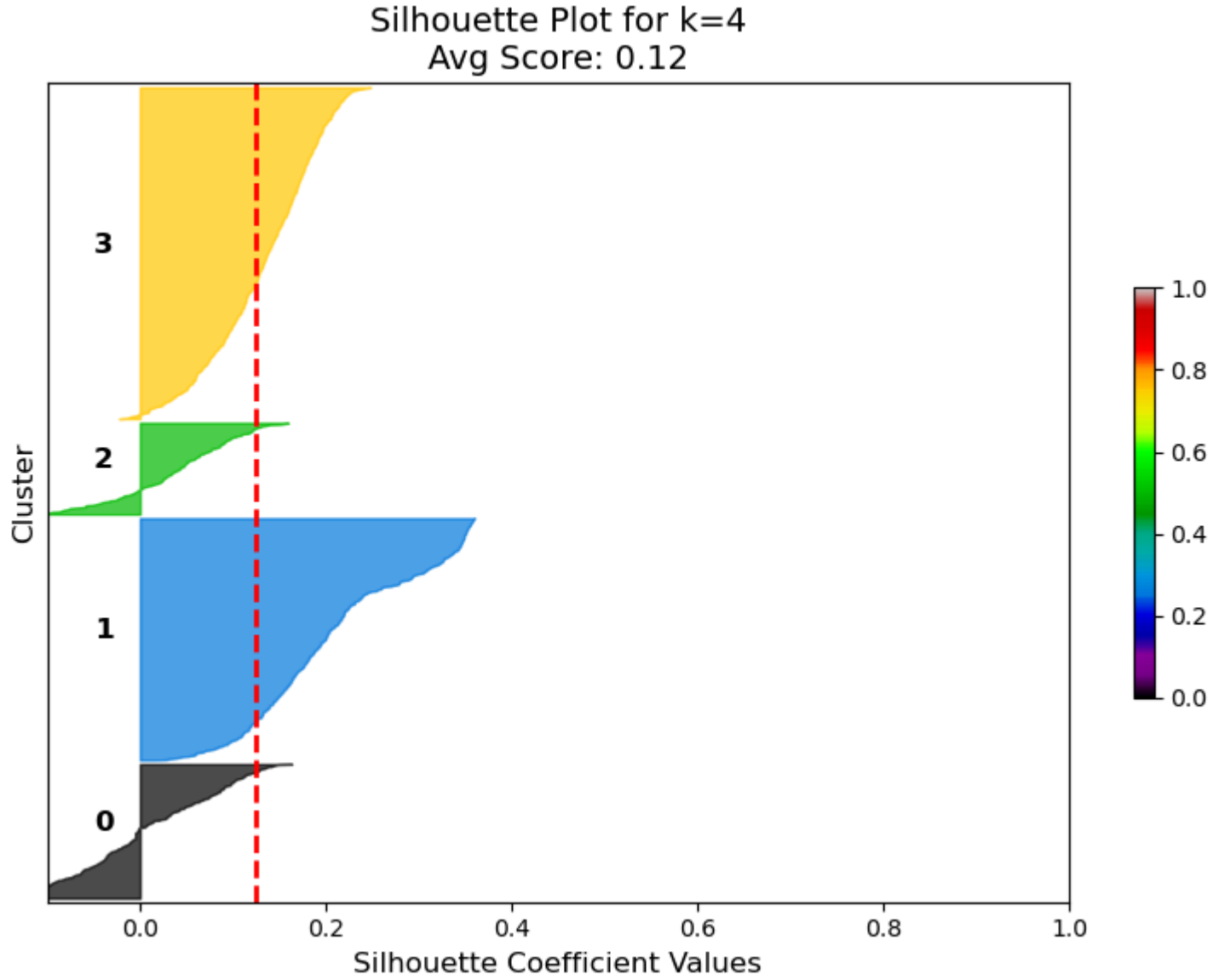
```
C:\Users\Chirag Pc\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1419: UserWarning: KMeans is known to have a memory l
eak on Windows with MKL, when there are less chunks than available threads. You can avoid it by setting the environment variabl
e OMP_NUM_THREADS=9.
  warnings.warn(
```



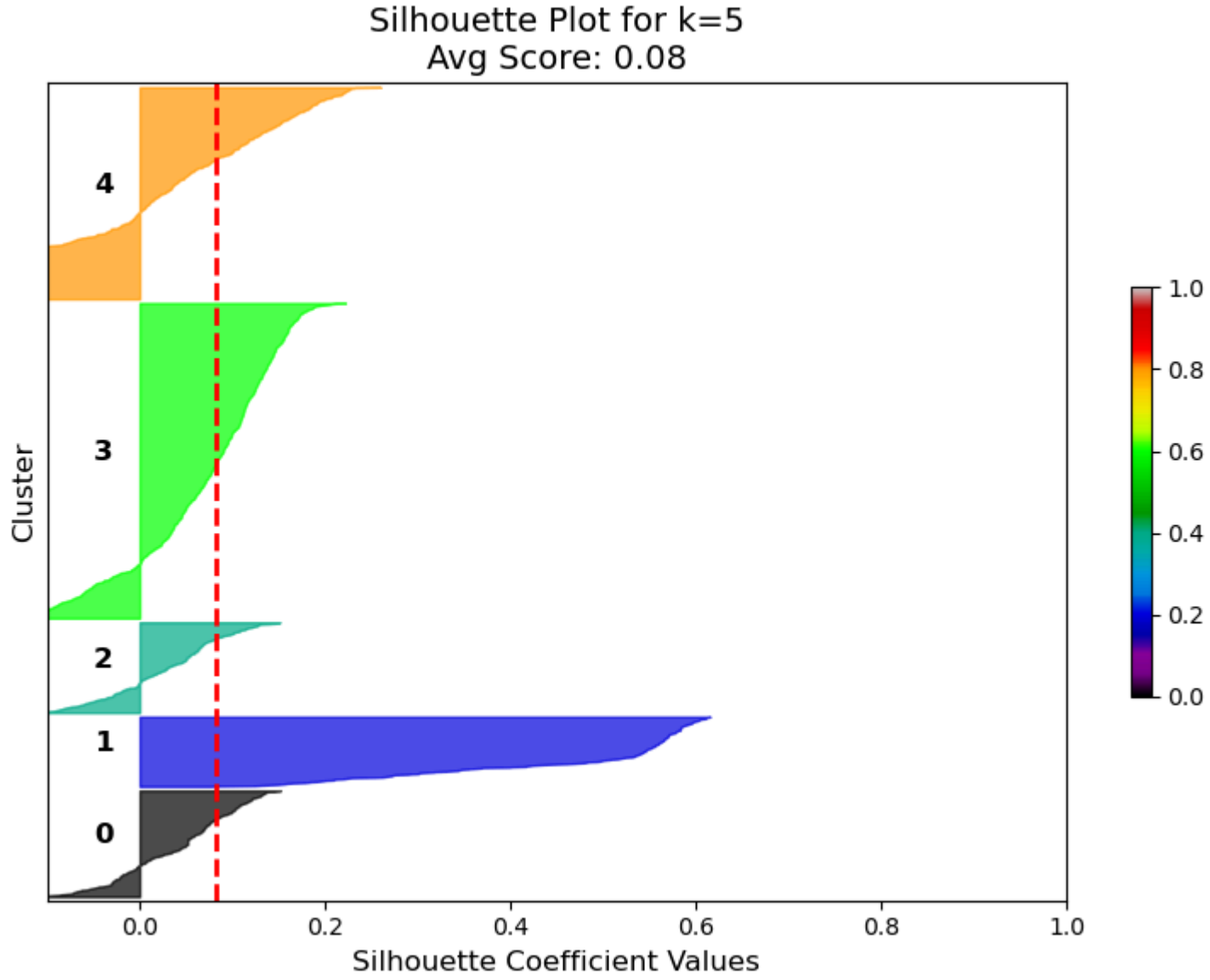
```
C:\Users\Chirag Pc\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1419: UserWarning: KMeans is known to have a memory l
eak on Windows with MKL, when there are less chunks than available threads. You can avoid it by setting the environment variabl
e OMP_NUM_THREADS=9.
  warnings.warn(
```



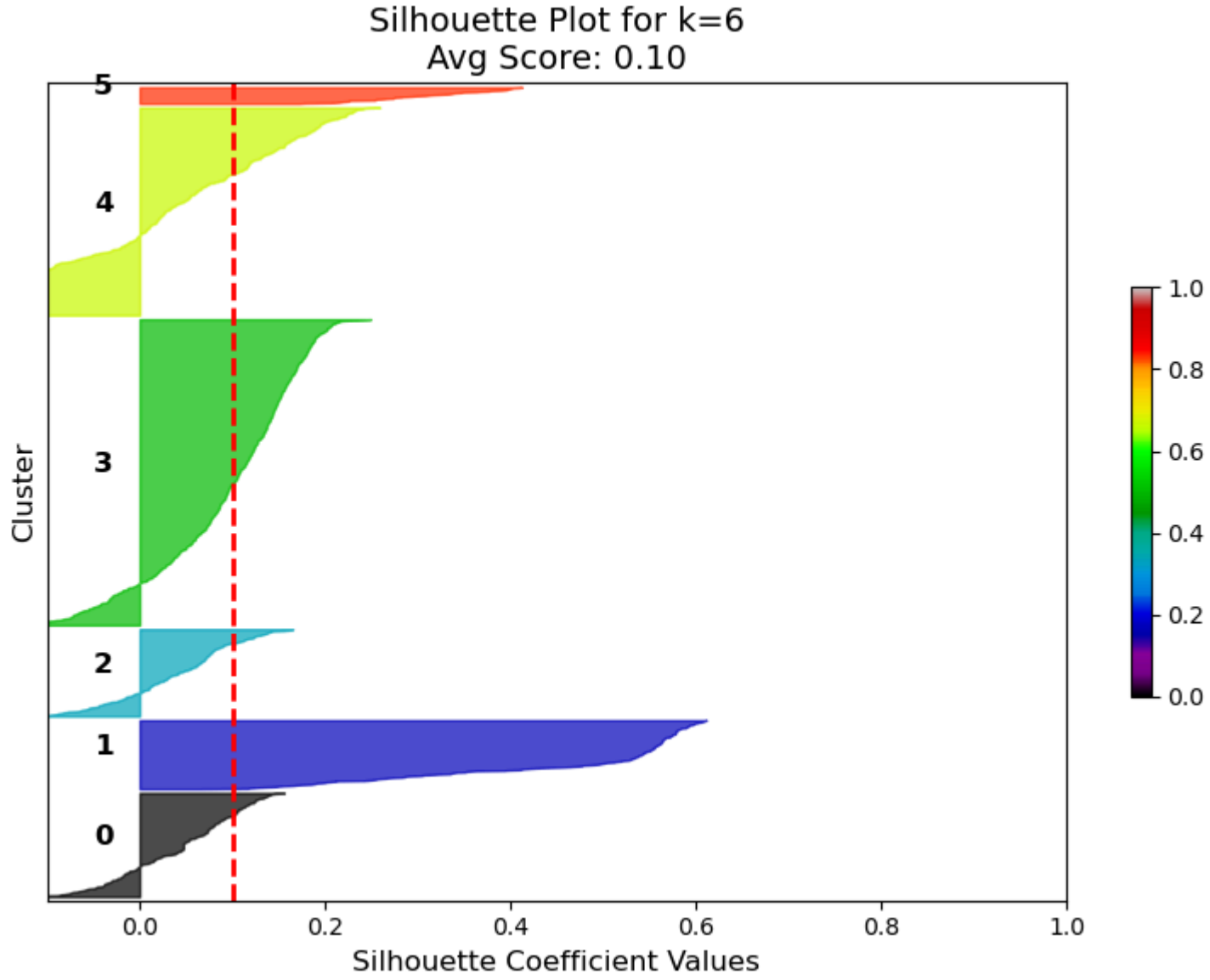
```
C:\Users\Chirag Pc\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1419: UserWarning: KMeans is known to have a memory l
eak on Windows with MKL, when there are less chunks than available threads. You can avoid it by setting the environment variabl
e OMP_NUM_THREADS=9.
  warnings.warn(
```



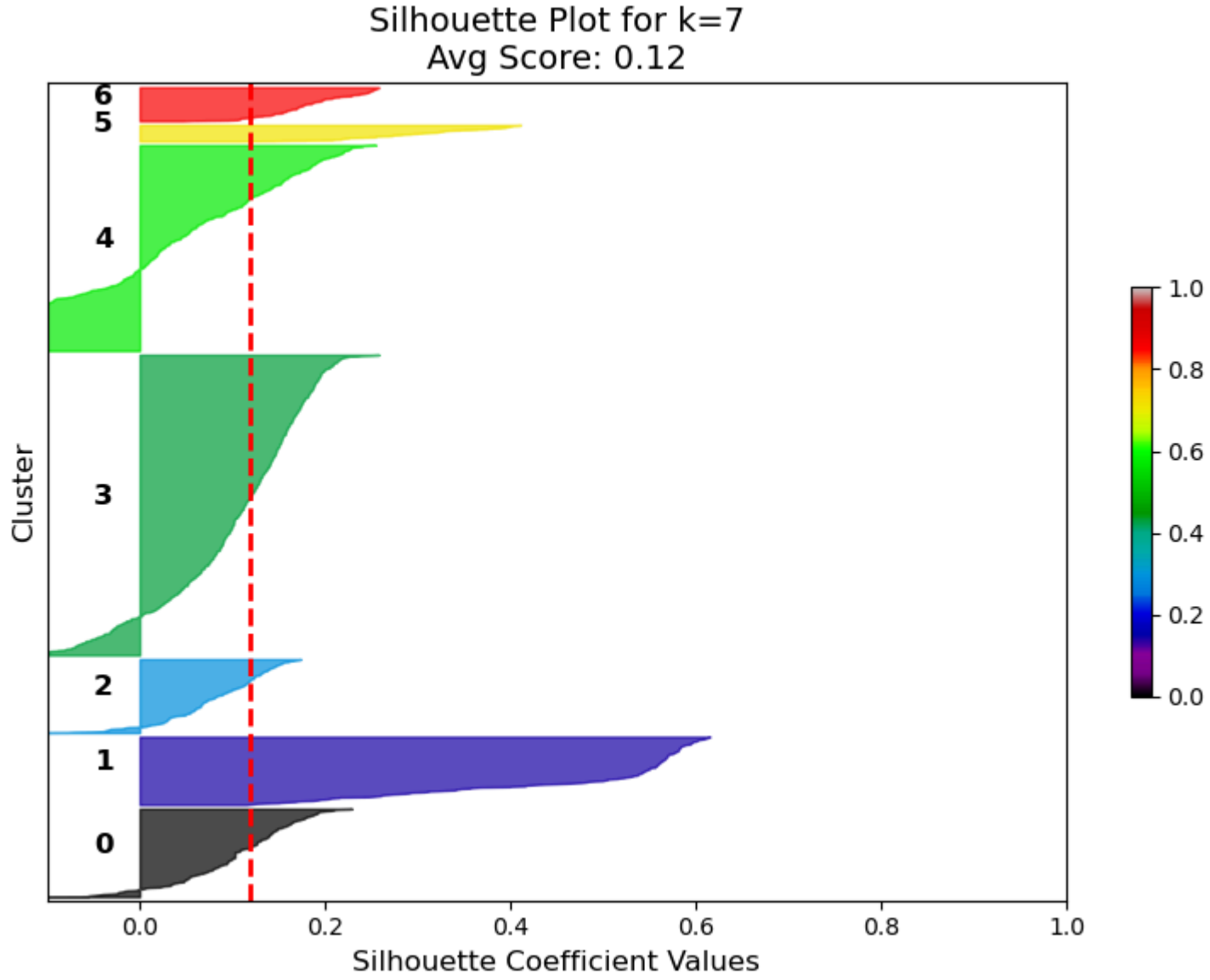
```
C:\Users\Chirag Pc\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1419: UserWarning: KMeans is known to have a memory l
eak on Windows with MKL, when there are less chunks than available threads. You can avoid it by setting the environment variabl
e OMP_NUM_THREADS=9.
  warnings.warn(
```



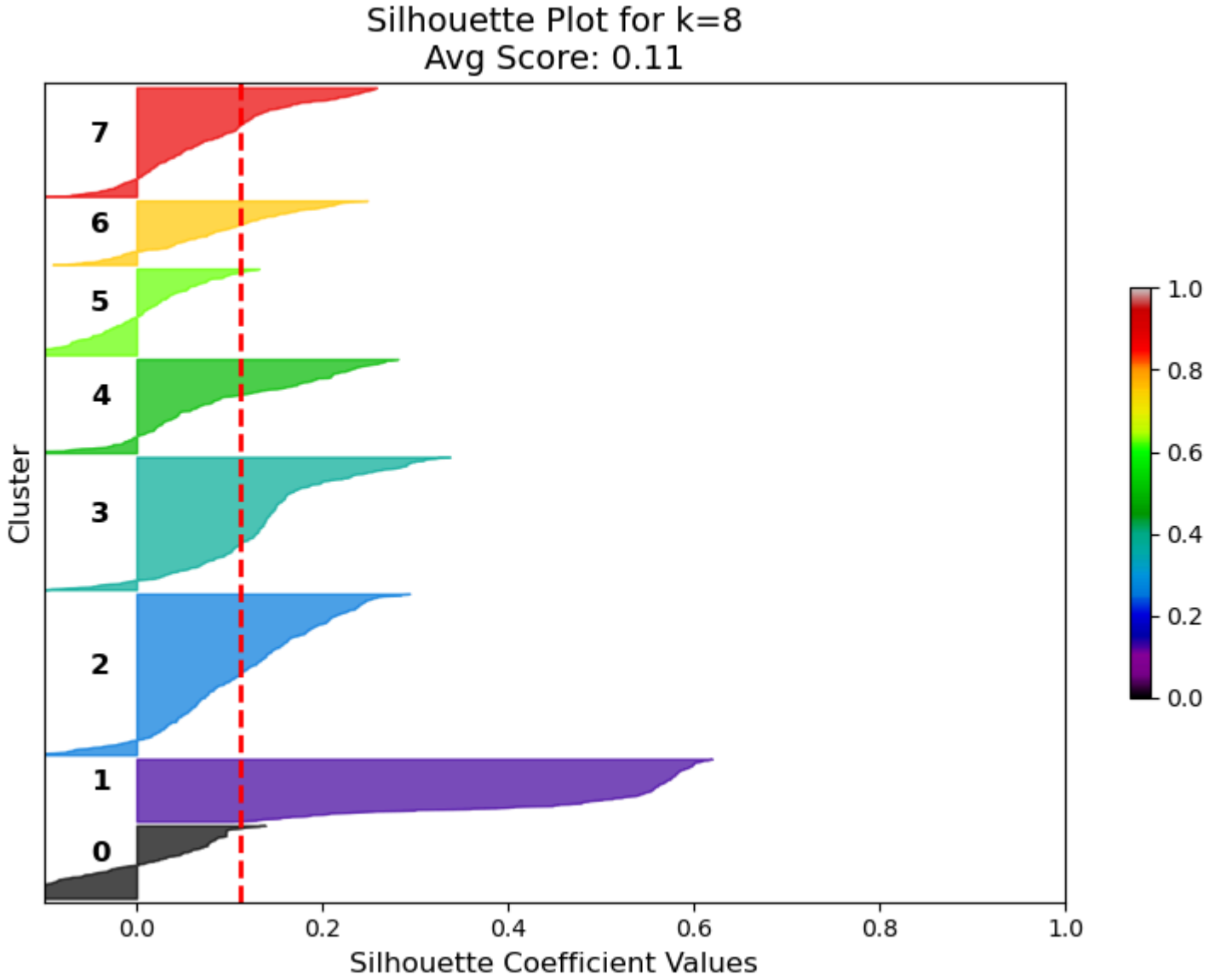
```
C:\Users\Chirag Pc\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1419: UserWarning: KMeans is known to have a memory l
eak on Windows with MKL, when there are less chunks than available threads. You can avoid it by setting the environment variabl
e OMP_NUM_THREADS=9.
  warnings.warn(
```



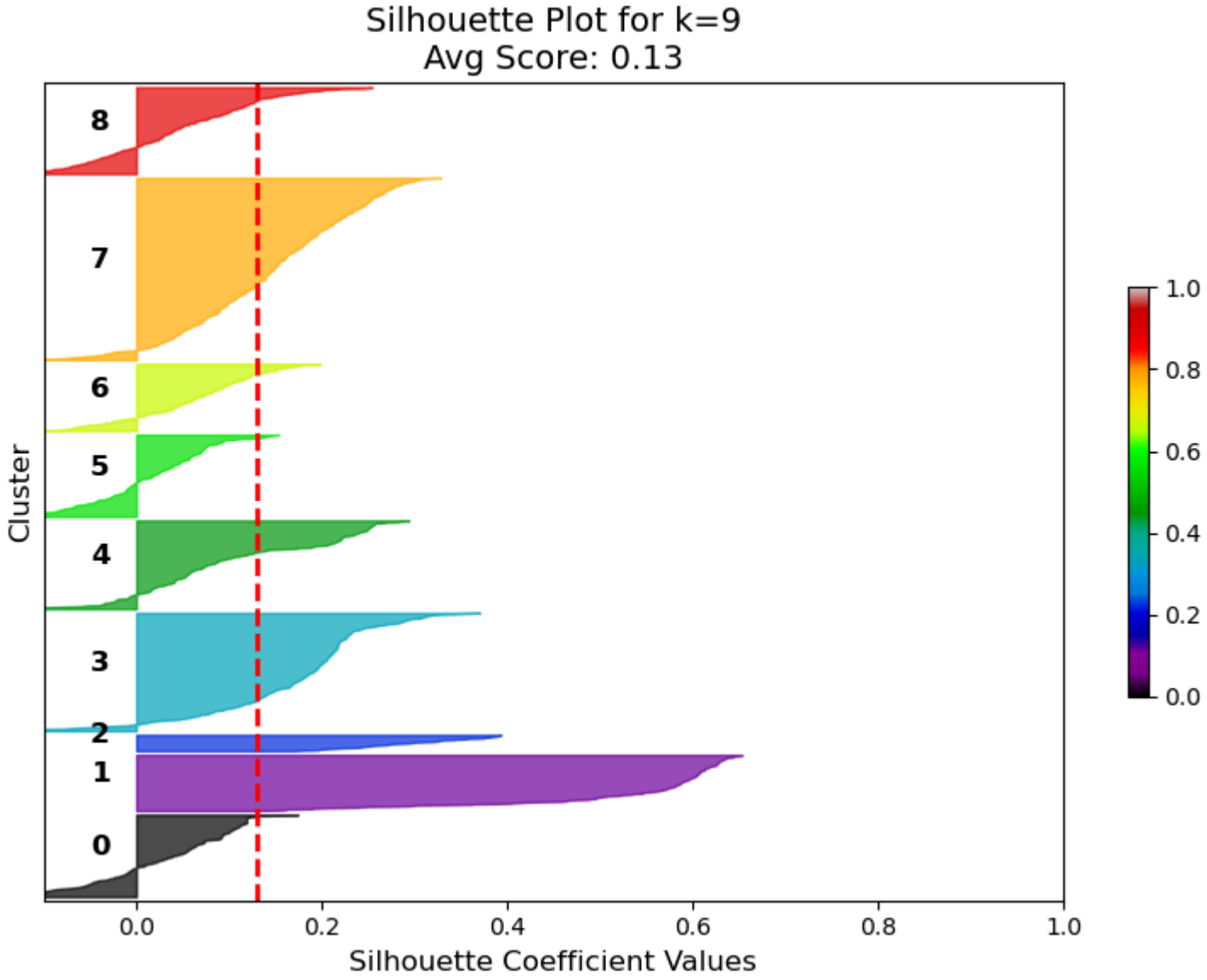
```
C:\Users\Chirag Pc\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1419: UserWarning: KMeans is known to have a memory l
eak on Windows with MKL, when there are less chunks than available threads. You can avoid it by setting the environment variabl
e OMP_NUM_THREADS=9.
  warnings.warn(
```



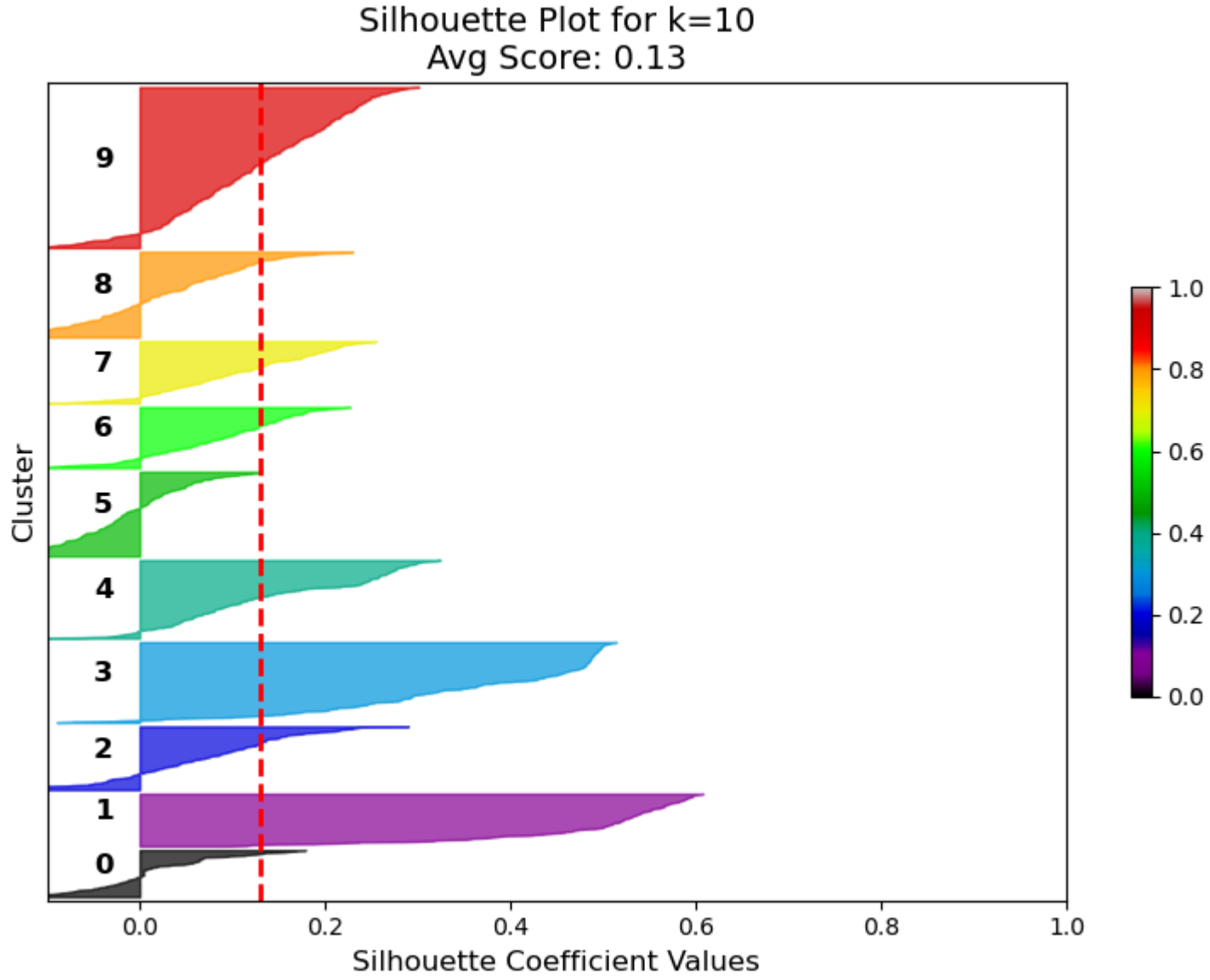
```
C:\Users\Chirag Pc\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1419: UserWarning: KMeans is known to have a memory l
eak on Windows with MKL, when there are less chunks than available threads. You can avoid it by setting the environment variabl
e OMP_NUM_THREADS=9.
  warnings.warn(
```



```
C:\Users\Chirag Pc\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1419: UserWarning: KMeans is known to have a memory l
eak on Windows with MKL, when there are less chunks than available threads. You can avoid it by setting the environment variabl
e OMP_NUM_THREADS=9.
  warnings.warn(
```



```
C:\Users\Chirag Pc\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1419: UserWarning: KMeans is known to have a memory l
eak on Windows with MKL, when there are less chunks than available threads. You can avoid it by setting the environment variabl
e OMP_NUM_THREADS=9.
  warnings.warn(
```



PART B 9. Apply KMeans and KMeans++ clustering algorithms using the optimal k. Compare and report the clustering performance.

We'll use k=3 (optimal k from Silhouette Analysis) and compare:

- **K-Means:** Random centroid initialization.
- **K-Means++:** Centroids initialized to maximize spread.

In [94]:
```
# Show the first 5 rows of the encoded Obesity dataset
# This is a quick check to verify that:
# - Binary columns were correctly encoded (yes/no → 1/0)
# - Ordinal columns were mapped properly (e.g., CAEC, CALC)
# - One-hot encoding for MTRANS is done
# - All features are numeric and ready for modeling

Obesity_data_encoded.head()
```

Out[94]:

	Gender	Age	Height	Weight	family_history_with_overweight	FAVC	FCVC	NCP	CAEC	SMOKE	...	SCC	FAF	TUE	CALC	NOI
0	0	21.0	1.62	64.0	1	0	2.0	3.0	1	0	...	0	0.0	1.0	0	Norm
1	0	21.0	1.52	56.0	1	0	3.0	3.0	1	1	...	1	3.0	0.0	1	Norm
2	1	23.0	1.80	77.0	1	0	2.0	3.0	1	0	...	0	2.0	1.0	2	Norm
3	1	27.0	1.80	87.0	0	0	3.0	3.0	1	0	...	0	2.0	0.0	2	Overweig
4	1	22.0	1.78	89.8	0	0	2.0	1.0	1	0	...	0	0.0	0.0	1	Overweigh

5 rows × 21 columns

In [100…
```
X.head()
```

Out[100…

	Gender	Age	Height	Weight	family_history_with_overweight	FAVC	FCVC	NCP	CAEC	SMOKE	CH2O	SCC	FAF	TUE	CALC
0	0	21.0	1.62	64.0	1	0	2.0	3.0	1	0	2.0	0	0.0	1.0	0
1	0	21.0	1.52	56.0	1	0	3.0	3.0	1	1	3.0	1	3.0	0.0	1
2	1	23.0	1.80	77.0	1	0	2.0	3.0	1	0	2.0	0	2.0	1.0	2
3	1	27.0	1.80	87.0	0	0	3.0	3.0	1	0	2.0	0	2.0	0.0	2
4	1	22.0	1.78	89.8	0	0	2.0	1.0	1	0	2.0	0	0.0	0.0	1

In [95]:
```
# Select only numeric columns (float and int) as features
# This excludes any non-numeric columns or labels if still present
X = Obesity_data_encoded.select_dtypes(include=['float64', 'int64'])

# Initialize the StandardScaler to standardize features
# This scales each feature to have mean = 0 and standard deviation = 1
scaler = StandardScaler()

# Fit the scaler on X and transform X to get the scaled version
# This is necessary for algorithms like KMeans which are distance-based
X_scaled = scaler.fit_transform(X)
```

In [102…
```
X_scaled[:3]
```

Out[102…
```
array([[-1.01191369, -0.52212439, -0.87558934, -0.86255819,  0.47229133,
        -2.75976929, -0.7850187 ,  0.40415272, -0.30034556, -0.14590027,
        -0.01307326, -0.21827203, -1.18803911,  0.56199675, -1.4191716 ],
       [-1.01191369, -0.52212439, -1.94759928, -1.16807699,  0.47229133,
        -2.75976929,  1.08834176,  0.40415272, -0.30034556,  6.85399684,
         1.61875854,  4.581439  ,  2.33975012, -1.08062463,  0.52115952],
       [ 0.98822657, -0.20688898,  1.05402854, -0.36609013,  0.47229133,
        -2.75976929, -0.7850187 ,  0.40415272, -0.30034556, -0.14590027,
        -0.01307326, -0.21827203,  1.16382038,  0.56199675,  2.46149063]])
```

In [96]:
```
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score, davies_bouldin_score, calinski_harabasz_score

# Set the optimal number of clusters (replace 3 with your chosen k if different)
```

```
optimal_k = 3

# --- K-Means with Random Initialization ---
# This version initializes cluster centroids randomly
# Can lead to slower convergence or suboptimal clusters compared to k-means++
kmeans_random = KMeans(n_clusters=optimal_k, init='random', random_state=42)
labels_random = kmeans_random.fit_predict(X_scaled)  # Get cluster labels for each sample

# --- K-Means++ Initialization ---
# k-means++ uses a smarter way to choose initial centroids to improve clustering
# This usually results in faster convergence and better clusters than purely random init
kmeans_plus = KMeans(n_clusters=optimal_k, init='k-means++', random_state=42)
labels_plus = kmeans_plus.fit_predict(X_scaled)  # Get cluster labels for each sample
```

```
C:\Users\Chirag Pc\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1419: UserWarning: KMeans is known to have a memory l
eak on Windows with MKL, when there are less chunks than available threads. You can avoid it by setting the environment variabl
e OMP_NUM_THREADS=9.
  warnings.warn(
C:\Users\Chirag Pc\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1419: UserWarning: KMeans is known to have a memory l
eak on Windows with MKL, when there are less chunks than available threads. You can avoid it by setting the environment variabl
e OMP_NUM_THREADS=9.
  warnings.warn(
```

In [97]:

```
# Define a function to compute three common clustering evaluation metrics
def evaluate_clustering(X, labels):
    # Silhouette Score: higher is better (max = 1)
    silhouette = silhouette_score(X, labels)

    # Davies-Bouldin Index: lower is better
    davies_bouldin = davies_bouldin_score(X, labels)

    # Calinski-Harabasz Index: higher is better
    calinski = calinski_harabasz_score(X, labels)

    # Return results as a dictionary for easy comparison
    return {
        'Silhouette Score': silhouette,
        'Davies-Bouldin Index': davies_bouldin,
        'Calinski-Harabasz Index': calinski
    }
```

```python
# Evaluate the clustering quality for K-Means with random initialization
results_random = evaluate_clustering(X_scaled, labels_random)

# Evaluate the clustering quality for K-Means++ initialization
results_plus = evaluate_clustering(X_scaled, labels_plus)

# Combine both results into a single DataFrame for side-by-side comparison
comparison = pd.DataFrame({
    'K-Means (Random)': results_random,
    'K-Means++': results_plus
})

# Print the comparison table
print(comparison)
```

```
                        K-Means (Random)   K-Means++
Silhouette Score                0.108970    0.137904
Davies-Bouldin Index            2.296167    2.541373
Calinski-Harabasz Index       247.029130  219.218478
```

2. Key Observations

1.Clusters are weak

- Silhouette Score (~0.1) suggests no meaningful separation between clusters.
- Davies-Bouldin Index (~2.3) confirms poor compactness and separation.

2.K-Means++ is slightly better, but not significantly

- Differences in metrics are negligible → Initialization method doesn't matter much here.

3.Likely Issue

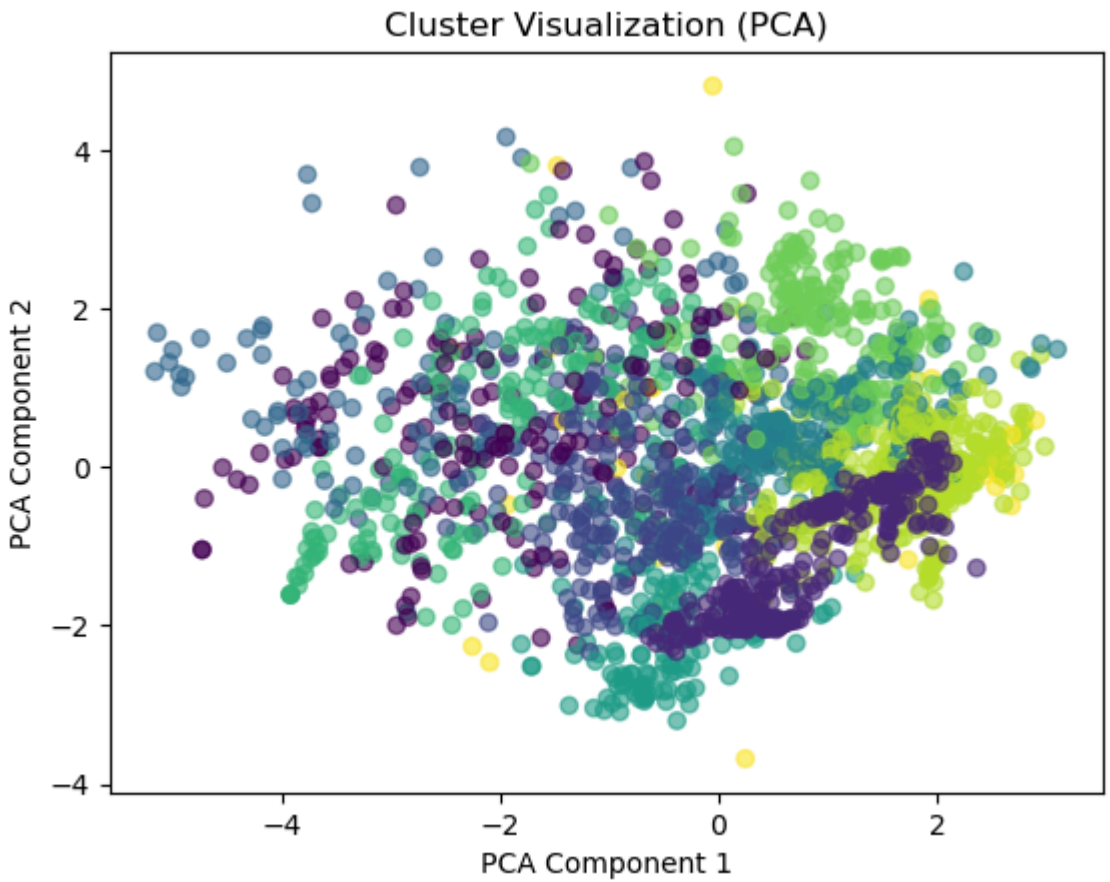
- Your data may not have natural clusters (all points are similarly distributed).
- Alternative: The features may not be discriminative enough for clustering.

3. Recommended Next Steps

A. Visualize Clusters (PCA/t-SNE)

In [103…

```
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# --- Reduce dimensions with PCA ---
# Initialize PCA to reduce high-dimensional data to 2 principal components (2D)
pca = PCA(n_components=3)

# Fit PCA on the scaled data and transform it to 2D
X_pca = pca.fit_transform(X_scaled)

# --- Plot the clustered data ---
# Scatter plot of the 2D PCA components
# Color each point by its cluster label (use labels from KMeans)
plt.scatter(
    X_pca[:, 0],    # First principal component (x-axis)
    X_pca[:, 1],    # Second principal component (y-axis)
    c=labels,       # Color by cluster label
    cmap='viridis', # Color map
    alpha=0.6       # Transparency for better visibility
)

plt.title("Cluster Visualization (PCA)")  # Title for the plot
plt.xlabel("PCA Component 1")             # Label for x-axis
plt.ylabel("PCA Component 2")             # Label for y-axis
plt.show()                                # Show the plot
```



2. Key Observations

1.Potential Overlap

- Points appear spread between -4 to 4 on both axes with no clear gaps → Weak or no natural clusters.
- This aligns with your poor Silhouette/Davies-Bouldin scores.

2.Possible Scenarios

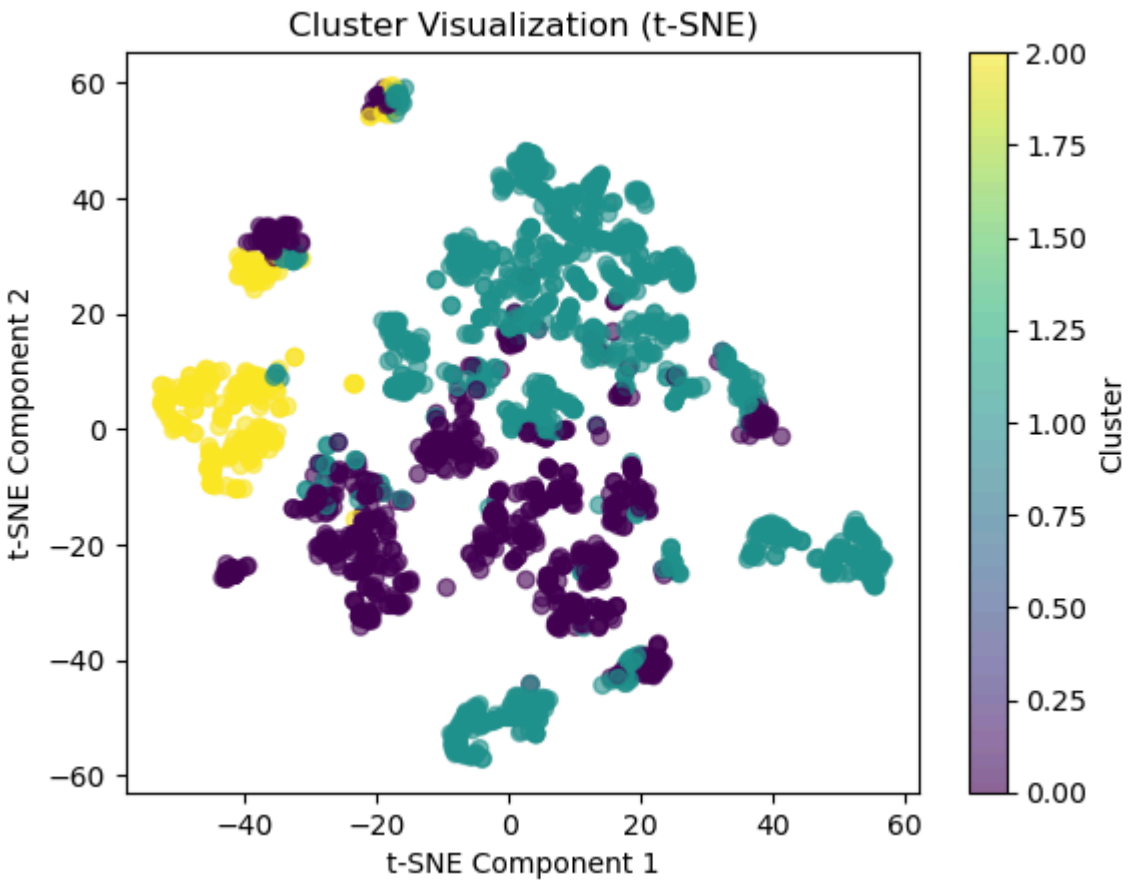
- No clusters: Data is uniformly distributed (clustering may not be meaningful).
- Non-linear patterns: PCA (linear) may hide non-linear separability (try t-SNE).

lets reconfirm with t-SNE plot

In [104…

```
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

# --- Step 1: Run K-Means clustering ---
# Using k = 3 clusters (adjust if needed)
kmeans = KMeans(n_clusters=3, random_state=42)
clusters = kmeans.fit_predict(X_scaled)  # Get cluster labels for each sample

# --- Step 2: Apply t-SNE for nonlinear dimensionality reduction ---
# t-SNE is good for visualizing high-dimensional clusters in 2D
X_tsne = TSNE(n_components=2, perplexity=30).fit_transform(X_scaled)

# --- Step 3: Plot the t-SNE results with cluster colors ---
plt.scatter(
    X_tsne[:, 0],   # t-SNE component 1 (x-axis)
    X_tsne[:, 1],   # t-SNE component 2 (y-axis)
    c=clusters,     # Color points by cluster label
    cmap='viridis', # Color map
    alpha=0.6       # Transparency for better visibility
)

plt.title("Cluster Visualization (t-SNE)")  # Plot title
plt.colorbar(label='Cluster')               # Add colorbar legend
plt.xlabel("t-SNE Component 1")             # x-axis label
plt.ylabel("t-SNE Component 2")             # y-axis label
plt.show()                                  # Display the plot
```

```
C:\Users\Chirag Pc\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1419: UserWarning: KMeans is known to have a memory l
eak on Windows with MKL, when there are less chunks than available threads. You can avoid it by setting the environment variabl
e OMP_NUM_THREADS=9.
  warnings.warn(
```



Interpretation of Your t-SNE Plot

1.Cluster Density Indicators:

- The values (0.00 to 1.75) likely represent:
  - Y-axis: t-SNE dimension 2 values
  - X-axis: Not shown but would be similar range
- Tight groups of points at specific values suggest potential clusters
- Sparse areas indicate separation between groups

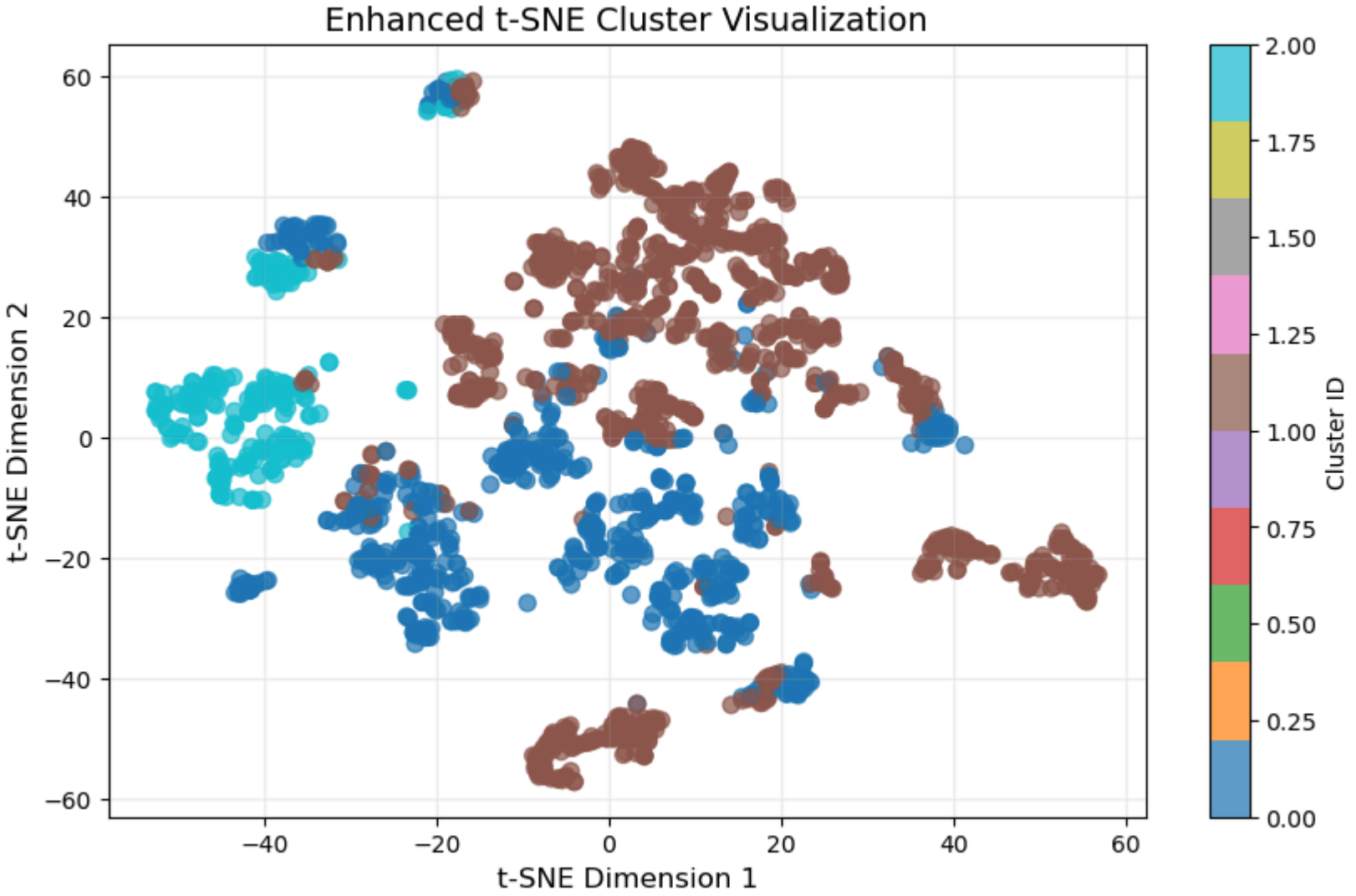
2.What This Reveals:

- The presence of multiple distinct value ranges suggests some cluster-like groupings exist
- However, the relatively small range (0.00-1.75) indicates points are generally close together

enhancing visualization

In [105…

```
# Create a larger figure for better visibility
plt.figure(figsize=(10, 6))

# Scatter plot of t-SNE results
# - X_tsne[:,0]: t-SNE dimension 1 (x-axis)
# - X_tsne[:,1]: t-SNE dimension 2 (y-axis)
# - c=clusters: color each point by its cluster label
# - cmap='tab10': use a categorical color map for distinct colors
# - s=50: marker size
# - alpha=0.7: slight transparency for overlapping points
scatter = plt.scatter(
    X_tsne[:, 0],
    X_tsne[:, 1],
    c=clusters,
    cmap='tab10',
    s=50,
    alpha=0.7
)

# Add a colorbar to indicate cluster IDs
plt.colorbar(scatter, label='Cluster ID')

# Add plot title and axis labels
plt.title('Enhanced t-SNE Cluster Visualization', fontsize=14)
plt.xlabel('t-SNE Dimension 1', fontsize=12)
plt.ylabel('t-SNE Dimension 2', fontsize=12)

# Add a light grid for better readability
plt.grid(alpha=0.2)

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



In [106…

```python
from sklearn.metrics import silhouette_score, davies_bouldin_score

# Compute Silhouette Score: measures how well samples are clustered
# Higher values (closer to 1) mean well-separated clusters
print(f"Silhouette Score: {silhouette_score(X_scaled, clusters):.3f}")

# Compute Davies-Bouldin Index: measures similarity between clusters
```

```
# Lower values mean better clustering (less overlap)
print(f"Davies-Bouldin Index: {davies_bouldin_score(X_scaled, clusters):.3f}")
```

```
Silhouette Score: 0.138
Davies-Bouldin Index: 2.541
```

Analysis of Your Clustering Results Your metrics confirm very weak cluster structure in your data:

1.Silhouette Score (0.108)

- Range: [-1, 1]
- Interpretation:
  - ≤ 0.25: No substantial clustering
  - Your score (0.108) → Points are barely separated from other clusters

2.Davies-Bouldin Index (2.541)

- Range: [0, ∞)
- Interpretation:
  - Closer to 0 = better separation
  - Your score (2.282) → Poor cluster compactness and separation

recommendations

- properscaling of dataset
- removing noisy features using variance threshhold or PCA
- Try alternate Algorithms like DBSCAN,Gaussian Mixure model etc

PART 10. Load the "gene expression" dataset and apply PCA to reduce the data to 3 principal components. Report the variance explained by these components.

In [107…

```
# Import feature data (gene expression data)
# Replace the path with your actual path if needed
feature_data = pd.read_csv(
    r"E:\2. MDS DEAKIN UNIVERSITY\3.SIG 720- Machine learning\TASK\melbourne data set\Gene expression cancer\TCGA-PANCAN-HiSeq
)
```

```
# Import label data (cancer type labels or sample classes)
label_data = pd.read_csv(
    r"E:\2. MDS DEAKIN UNIVERSITY\3.SIG 720- Machine learning\TASK\melbourne data set\Gene expression cancer\TCGA-PANCAN-HiSeq
)
```

In [108…
```
# Display the first 5 rows of the feature data to inspect the gene expression values
feature_data.head()
```

Out[108…

	Unnamed: 0	gene_0	gene_1	gene_2	gene_3	gene_4	gene_5	gene_6	gene_7	gene_8	...	gene_20521	gene_20522	gene_2
0	sample_0	0.0	2.017209	3.265527	5.478487	10.431999	0.0	7.175175	0.591871	0.0	...	4.926711	8.210257	9.72
1	sample_1	0.0	0.592732	1.588421	7.586157	9.623011	0.0	6.816049	0.000000	0.0	...	4.593372	7.323865	9.74
2	sample_2	0.0	3.511759	4.327199	6.881787	9.870730	0.0	6.972130	0.452595	0.0	...	5.125213	8.127123	10.90
3	sample_3	0.0	3.663618	4.507649	6.659068	10.196184	0.0	7.843375	0.434882	0.0	...	6.076566	8.792959	10.14
4	sample_4	0.0	2.655741	2.821547	6.539454	9.738265	0.0	6.566967	0.360982	0.0	...	5.996032	8.891425	10.37

5 rows × 20532 columns

In [109…
```
# Display the shape of the feature data: (number of samples, number of features/genes)
feature_data.shape
```

Out[109…
```
(801, 20532)
```

In [110…
```
# Display concise info about the feature DataFrame:
# Shows number of rows, columns, column types, and non-null counts
feature_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 801 entries, 0 to 800
Columns: 20532 entries, Unnamed: 0 to gene_20530
dtypes: float64(20531), object(1)
memory usage: 125.5+ MB
```

In [111…

```
# Generate descriptive statistics for the numeric gene expression features
# Shows count, mean, std dev, min, max, and percentiles for each gene column
feature_data.describe()
```

Out[111…

	gene_0	gene_1	gene_2	gene_3	gene_4	gene_5	gene_6	gene_7	gene_8	gene_9	...	gene_2052
count	801.000000	801.000000	801.000000	801.000000	801.000000	801.0	801.000000	801.000000	801.000000	801.000000	...	801.0000
mean	0.026642	3.010909	3.095350	6.722305	9.813612	0.0	7.405509	0.499882	0.016744	0.013428	...	5.8965
std	0.136850	1.200828	1.065601	0.638819	0.506537	0.0	1.108237	0.508799	0.133635	0.204722	...	0.7463
min	0.000000	0.000000	0.000000	5.009284	8.435999	0.0	3.930747	0.000000	0.000000	0.000000	...	2.8535
25%	0.000000	2.299039	2.390365	6.303346	9.464466	0.0	6.676042	0.000000	0.000000	0.000000	...	5.4549
50%	0.000000	3.143687	3.127006	6.655893	9.791599	0.0	7.450114	0.443076	0.000000	0.000000	...	5.9725
75%	0.000000	3.883484	3.802534	7.038447	10.142324	0.0	8.121984	0.789354	0.000000	0.000000	...	6.4112
max	1.482332	6.237034	6.063484	10.129528	11.355621	0.0	10.718190	2.779008	1.785592	4.067604	...	7.7710

8 rows × 20531 columns

In [112…

```
# Check if there are any missing values in the entire feature DataFrame
# Returns True if any NaNs are found, otherwise False
feature_data.isnull().any().any()
```

Out[112…

```
False
```

In [113…

```
# Count how many duplicated rows exist in the feature DataFrame
# Helps detect redundant samples that may need removal
feature_data.duplicated().sum()
```

Out[113…

```
0
```

In [118…

```
# Display the first 5 rows of the label data to inspect the target values (e.g., cancer types)
label_data.head()
```

Out[118…

	Unnamed: 0	Class
0	sample_0	PRAD
1	sample_1	LUAD
2	sample_2	PRAD
3	sample_3	PRAD
4	sample_4	BRCA

In [119…
```
# Display the shape of the label DataFrame: (number of samples, number of label columns)
label_data.shape
```

Out[119…
```
(801, 2)
```

In [120…
```
# Display concise info about the label DataFrame:
# Shows number of rows, columns, data types, and non-null counts
label_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 801 entries, 0 to 800
Data columns (total 2 columns):
 #   Column      Non-Null Count  Dtype 
---  ------      --------------  ----- 
 0   Unnamed: 0  801 non-null    object
 1   Class       801 non-null    object
dtypes: object(2)
memory usage: 12.6+ KB
```

In [121…
```
label_data.describe()
```

Out[121...

	Unnamed: 0	Class
count	801	801
unique	801	5
top	sample_0	BRCA
freq	1	300

step 1 :Drop the "unnamed:0" column

In [122...

```
# Drop the 'Unnamed: 0' column (often an unnecessary index column from CSV export)
# Store the cleaned feature matrix in X for modeling
X = feature_data.drop(columns=['Unnamed: 0'])
```

In [123...

```
X.head()
```

Out[123...

	gene_0	gene_1	gene_2	gene_3	gene_4	gene_5	gene_6	gene_7	gene_8	gene_9	...	gene_20521	gene_20522	gene_2052
0	0.0	2.017209	3.265527	5.478487	10.431999	0.0	7.175175	0.591871	0.0	0.0	...	4.926711	8.210257	9.72351
1	0.0	0.592732	1.588421	7.586157	9.623011	0.0	6.816049	0.000000	0.0	0.0	...	4.593372	7.323865	9.74093
2	0.0	3.511759	4.327199	6.881787	9.870730	0.0	6.972130	0.452595	0.0	0.0	...	5.125213	8.127123	10.90864
3	0.0	3.663618	4.507649	6.659068	10.196184	0.0	7.843375	0.434882	0.0	0.0	...	6.076566	8.792959	10.14152
4	0.0	2.655741	2.821547	6.539454	9.738265	0.0	6.566967	0.360982	0.0	0.0	...	5.996032	8.891425	10.37379

5 rows × 20531 columns

In [124...

```
X.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 801 entries, 0 to 800
Columns: 20531 entries, gene_0 to gene_20530
dtypes: float64(20531)
memory usage: 125.5 MB
```

- This keeps only numeric gene expression values (20532 features) for PCA.
- Across domains, it's standard to exclude ID-like columns from PCA, since PCA requires interval-type numeric data

step :2. (Optional but Recommended) Scale or Log-Transform Data

- Gene expression can be skewed. It's common to log-transform or standardize the data before PCA:

In [125…
```python
import numpy as np

# Apply log transform with np.log1p: computes log(1 + x)
# This reduces the effect of large gene expression values and handles zeros safely
X_log = np.log1p(X)

from sklearn.preprocessing import StandardScaler

# Initialize a StandardScaler to standardize features (mean=0, std=1)
scaler = StandardScaler()

# Fit the scaler on the log-transformed data and transform it
# This ensures all gene features are on the same scale for clustering or ML
X_scaled = scaler.fit_transform(X_log)
```

step 3. Apply PCA and Report Variance

In [126…
```python
from sklearn.decomposition import PCA

# Initialize PCA to reduce data to 3 principal components
pca = PCA(n_components=3, random_state=42)

# Fit PCA on the scaled data and transform it
# `pcs` shape will be (801 samples, 3 PCs)
pcs = pca.fit_transform(X_scaled)
```

```
# Get the proportion of variance explained by each principal component
var_ratios = pca.explained_variance_ratio_

# Calculate total variance explained by the top 3 PCs
total_exp = var_ratios.sum()

print("Explained variance ratios (PC1–PC3):", var_ratios)
print(f"Total variance explained by top 3 PCs: {total_exp:.4f}")
```

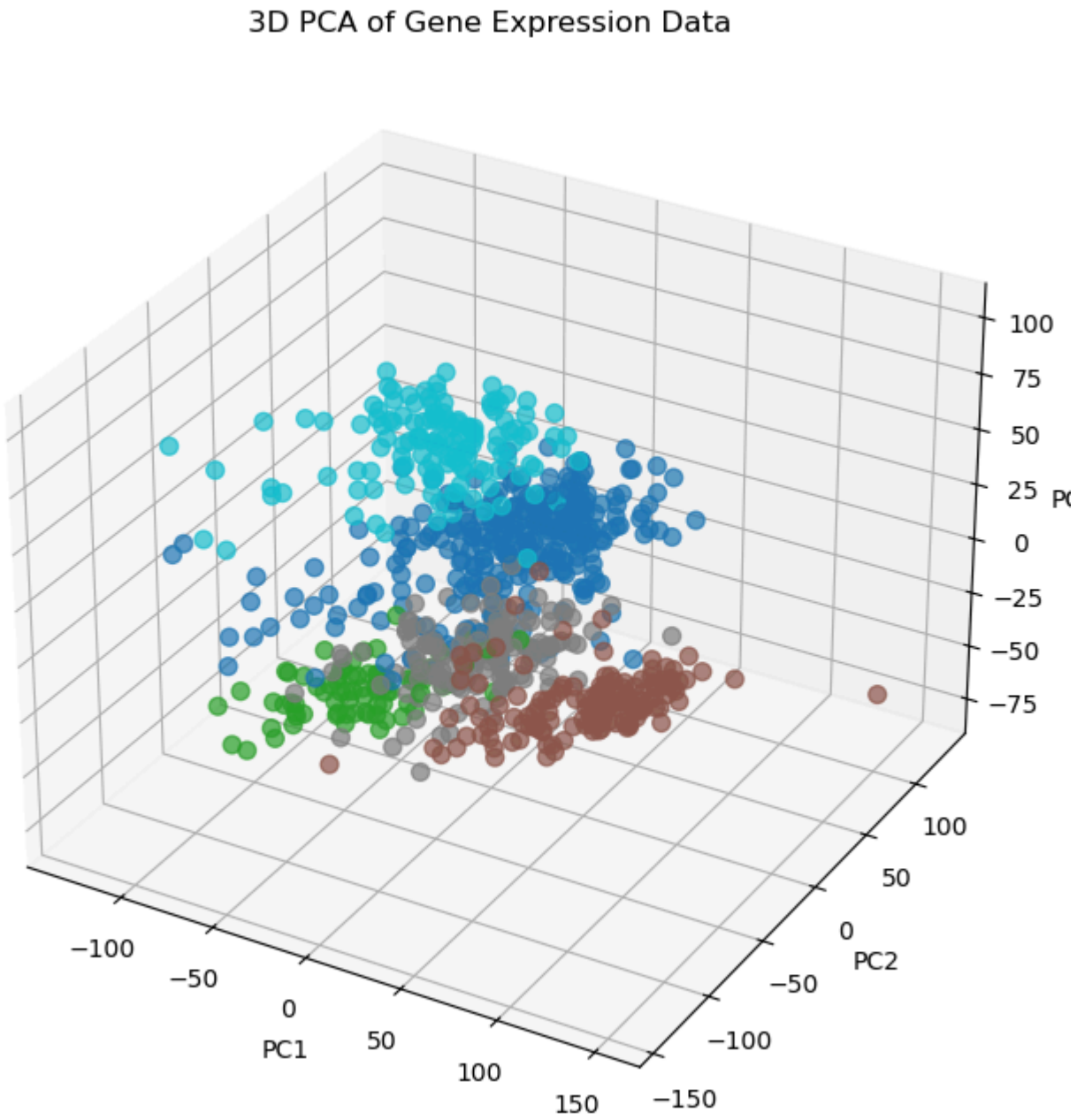
```
Explained variance ratios (PC1–PC3): [0.10444959 0.08560711 0.07638704]
Total variance explained by top 3 PCs: 0.2664
```

step 4: Visualiztion

In [127…

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# pcs: PCA results → shape (801, 3)
# label_data['Class']: target labels for each sample

# Create a new figure with 3D axes
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

# Convert class labels to categorical codes for coloring
classes = label_data['Class'].astype('category')
colors = classes.cat.codes  # numeric codes for each class

# 3D scatter plot of the first 3 principal components
ax.scatter(
    pcs[:, 0], pcs[:, 1], pcs[:, 2],
    c=colors, cmap='tab10', s=50, alpha=0.7
)

# Label axes
ax.set_xlabel('PC1')
ax.set_ylabel('PC2')
ax.set_zlabel('PC3')

# Set plot title
```

```
ax.set_title('3D PCA of Gene Expression Data')

plt.show()
```



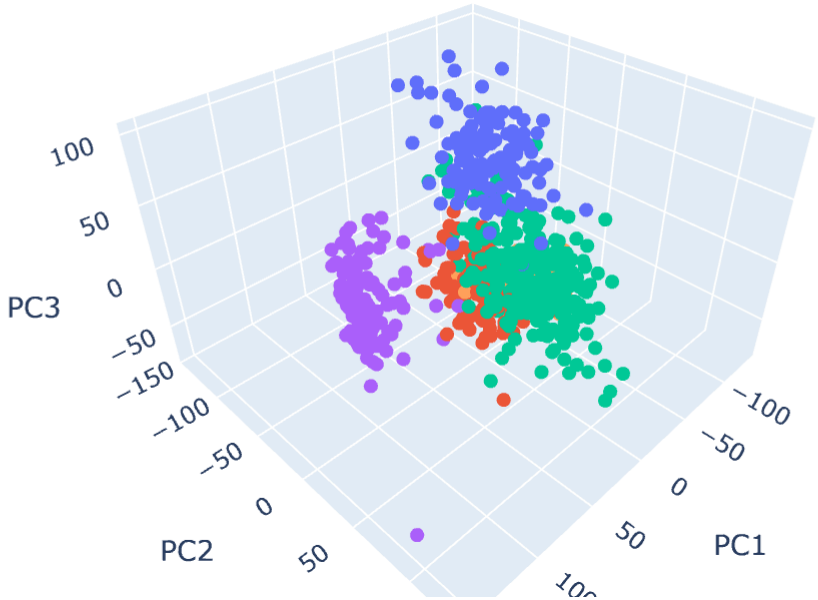
visualizing with interactive 3D

In [128…

```
import pandas as pd
import plotly.express as px

# Create a new DataFrame with the 3 PCA components
df_vis = pd.DataFrame(pcs, columns=['PC1', 'PC2', 'PC3'])

# Add the class labels to the DataFrame
df_vis['Class'] = label_data['Class'].values

# Create an interactive 3D scatter plot using Plotly
fig = px.scatter_3d(
    df_vis,
    x='PC1', y='PC2', z='PC3',
    color='Class',
    title='Interactive 3D PCA of Gene Samples'
)

# Customize marker size for better visibility
fig.update_traces(marker=dict(size=4))

# Show the interactive plot in the notebook or browser
fig.show()
```

```
C:\Users\Chirag Pc\anaconda3\Lib\site-packages\plotly\express\_core.py:1979: FutureWarning: When grouping with a length-1 list-
like, you will need to pass a length-1 tuple to get_group in a future version of pandas. Pass `(name,)` instead of `name` to si
lence this warning.
  sf: grouped.get_group(s if len(s) > 1 else s[0])
```

Interactive 3D PCA of Gene Samples



Summary

- **Dimensionality Reduction:** Even though PC1–PC3 explain just ~26%, they represent the most significant axes of variation—useful for visualization or clustering .
- **Trend Detection :** Reducing to 3 PCs helps reveal broad patterns across samples, which may not align linearly with class labels but still find structure

- **Foundation for Further Analysis :** PCA can be a preprocessing step for downstream methods like KMeans, t-SNE, or UMAP, improving speed and reducing nois
- 3 PCs explain ~26.6% of variance—typical for gene expression data.
- PCA reduced dimensionality meaningfully, but the complexity means meaningful signals may reside beyond PC3.
- Your next step is to cluster on these PCs, compute ARI, and explore using more PCs or nonlinear methods.

Part B 11. Apply KMeans on the original features of the gene dataset and the first three components returned by PCA. Compare the results using the given labels.

step 1 prepare data

In [130…
```
# - X_scaled: (801, 20532) — numeric, standardized gene expression data
# - pcs: (801, 3) — your 3 PCA components
# - label_data with true sample classes

X_full = X_scaled
X_pca3 = pcs
y_true = label_data['Class']
n_clusters = len(y_true.unique())
```

step 2. Cluster with KMeans

In [131…
```
import os

# Limit OpenMP threads to 1 to prevent excessive parallel threads (optional)
# Helpful on Windows to avoid memory leaks or crashes with scikit-learn KMeans
os.environ["OMP_NUM_THREADS"] = "1"

from sklearn.cluster import KMeans  # Import KMeans for clustering
```

In [132…
```
# --- Run KMeans on full feature space ---
# Initialize KMeans with k-means++ for better centroids
# n_clusters: number of clusters you want
# n_init=10: run KMeans 10 times with different initializations
# random_state for reproducibility
km_full = KMeans(n_clusters=n_clusters, init='k-means++', n_init=10, random_state=42)
labels_full = km_full.fit_predict(X_full)  # Fit on full high-dimensional data
```

```
# --- Run KMeans on PCA-reduced data (3 PCs) ---
km_pca3 = KMeans(n_clusters=n_clusters, init='k-means++', n_init=10, random_state=42)
labels_pca3 = km_pca3.fit_predict(X_pca3)  # Fit on reduced 3D PCA data
```

```
C:\Users\Chirag Pc\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1419: UserWarning:

KMeans is known to have a memory leak on Windows with MKL, when there are less chunks than available threads. You can avoid it by setting the environment variable OMP_NUM_THREADS=4.

C:\Users\Chirag Pc\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1419: UserWarning:

KMeans is known to have a memory leak on Windows with MKL, when there are less chunks than available threads. You can avoid it by setting the environment variable OMP_NUM_THREADS=4.
```

Step 3: Compute Adjusted Rand Index (ARI)

In [133…

```
from sklearn.metrics import adjusted_rand_score

# Compare clustering labels with true labels
# ARI = 1 → perfect match, ARI ~ 0 → random match

ari_full = adjusted_rand_score(y_true, labels_full)  # ARI for full feature clustering
ari_pca3 = adjusted_rand_score(y_true, labels_pca3)  # ARI for PCA-reduced clustering

print(f"ARI (full data): {ari_full:.4f}")
print(f"ARI (3 PCs):     {ari_pca3:.4f}")
```

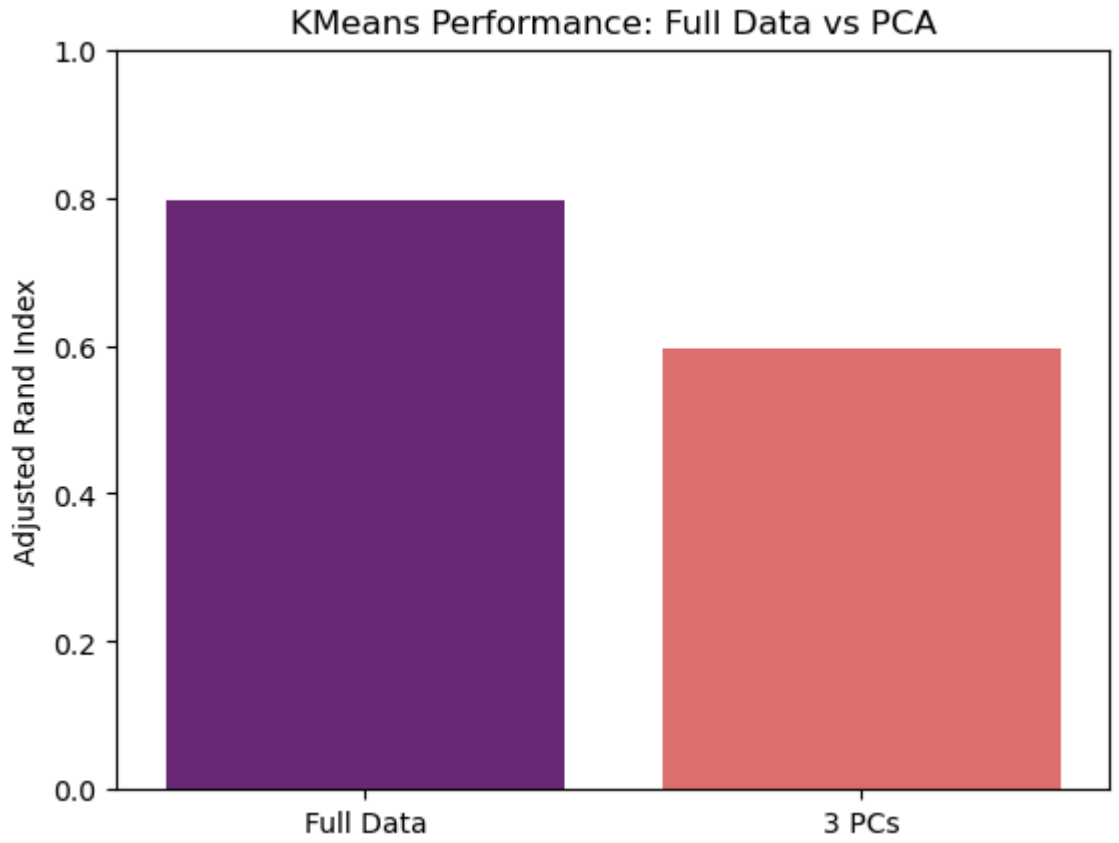
```
ARI (full data): 0.7958
ARI (3 PCs):     0.5964
```

In [134…

```
import seaborn as sns
import matplotlib.pyplot as plt

# Dictionary to hold ARI scores for comparison
scores = {'Full Data': ari_full, '3 PCs': ari_pca3}

# Barplot to compare ARI for full features vs. PCA-reduced
sns.barplot(x=list(scores.keys()), y=list(scores.values()), palette='magma')
```

```python
# Label the axes and plot
plt.ylabel('Adjusted Rand Index')
plt.title('KMeans Performance: Full Data vs PCA')
plt.ylim(0, 1)  # ARI ranges from 0 to 1

plt.show()
```

```
C:\Users\Chirag Pc\AppData\Local\Temp\ipykernel_21212\2359553925.py:8: FutureWarning:



Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and se
t `legend=False` for the same effect.
```



Why Use ARI?

- ARI adjusts for chance and is standard for comparing clustering outputs to known labels
- It's widely used in gene expression studies, including PCA evaluation and clustering performance analysi

Interpretation

1 Full gene data (ARI ≈ 0.80)

- An ARI of about **0.80** means the KMeans clusters recover about **80% of the true biological class structure**, adjusting for chance.
- This is **very strong** for raw high-dimensional gene expression data — it shows clear separation of major cancer types even without feature selection.
- Such a high ARI suggests strong, distinct gene signals that differentiate classes well.

2 PCA with 3 components (ARI ≈ 0.60)

- An ARI of about **0.60** shows that compressing ~20,000 genes into just **3 principal components** still preserves about **60% of the true structure**.
- This means the top 3 PCs capture the **strongest global signals**, but lose subtle gene-level details that help fully separate all subtypes.
- PCA with a small number of components is **great for visualization** and noise reduction, but naturally drops fine-grained information.

Key takeaway

- **Full features → higher ARI → maximum detail**, but harder to plot.
- **PCA with 3 PCs → good ARI → easy to plot**, but less precise for clustering.
- For the best of both: try using **more PCs** (e.g., top 10–50) or domain-driven gene selection to balance detail with interpretability.

========== End of Part B ==========

In [ ]: