

```
from IPython.display import Image, display
display(Image(filename='Lecture 2.png'))
```



MIDOCEAN
UNIVERSITY

Cleaning Dataset

Lecture 02

Prepared by

Dr. Hager Saleh

What is missing values

Missing values (or NaNs) are data points that are not recorded or available in a dataset.

Common methods to address missing values include

- Remove columns with missing values
- Removing incomplete rows.
- Imputing values using the mean, median, or mode,
- Applying forward or backward filling for time-series data,
- Using model-based imputation.

What is an Outlier?

An outlier is a data point that significantly differs from other values in a dataset. Outliers can be either much larger or much smaller than most of the other data points. Example [160, 162, 165, 168, 170, 163, 164, 166, 167, 300]

In this dataset, most values are around 160-170 cm, but 300 cm is an extreme value compared to the rest of the data. This value is likely an outlier.

What is mean?

The mean, often referred to as the average, is a measure of central tendency that gives an idea of the typical value in a dataset.

$$\bar{x} = \frac{1}{n}(x_1 + x_2 + \dots + x_n) = \frac{1}{n} \sum_{i=1}^n x_i$$

Where

- (n) is the sample size

Example

The data values are:

1792, 1666, 1362, 1614, 1460, 1867, 1439

$$\bar{x} = \frac{1792 + 1666 + 1362 + 1614 + 1460 + 1867 + 1439}{7}$$

$$= \frac{11200}{7}$$

$$= 1600$$

What is median (M)?

- The **median** is the middle value of a data set when the numbers are arranged in order.
- If the data set has an odd number of values, the median is the middle number.
- If the data set has an even number of values, the median is the average of the two middle numbers.

Example 1:

Data: (2, 4, 6)

Median = 4

Example 2:

Data: (2, 4, 6, 8)

Median = 4+6/2 = 5

Coding

```
import pandas as pd
```

```
df = pd.read_csv('Diabetes Missing Data.csv')
```

```
df.head()
```

	Pregnant	Glucose	Diastolic_BP	Skin_Fold	Serum_Insulin	BMI	\
0	6	148.0	72.0	35.0	NaN	33.6	
1	1	85.0	66.0	29.0	NaN	26.6	

2	8	183.0	64.0	NaN	NaN	23.3
3	1	89.0	66.0	23.0	94.0	28.1
4	0	137.0	40.0	35.0	168.0	43.1

	Diabetes_Pedigree	Age	Class
0	0.627	50	1
1	0.351	31	0
2	0.672	32	1
3	0.167	21	0
4	2.288	33	1

```
len(df)
```

```
768
```

Check Missing Values

It is crucial to check for missing data and understand how much of the data is missing in each column.

```
# Check for missing values
df.isnull().sum()
```

```
Pregnant      0
Glucose       5
Diastolic_BP  35
Skin_Fold     227
Serum_Insulin 374
BMI           11
Diabetes_Pedigree  0
Age           0
Class         0
dtype: int64
```

How to handle missing values?

When to remove columns with missing values

- Use Case: When a column has a large amount of missing data, say more than 30-50% of its values are missing.

```
missing_percentage = df.isnull().mean() * 100
```

```
# Filter columns with more than 30% (or 50%) missing values
columns_with_missing_30 = missing_percentage[missing_percentage > 30]
```

```
# Display the columns with more than 30% missing values
print("Columns with more than 30% missing values:")
print(columns_with_missing_30)
```

```
Columns with more than 30% missing values:  
Serum_Insulin    48.697917  
dtype: float64
```

How Remove column

```
missing_percentage = df.isnull().mean() * 100  
  
threshold = 30  
  
# Filter columns with more than the threshold percentage of missing values  
columns_to_drop = missing_percentage[missing_percentage > threshold].index  
  
# Drop the columns  
df_cleaned = df.drop(columns=columns_to_drop)  
  
# Display the resulting dataframe after removing columns with too many missing values  
print(f"Columns dropped: {list(columns_to_drop)}")  
print(f"Remaining columns: {df_cleaned.columns}")  
  
Columns dropped: ['Serum_Insulin']  
Remaining columns: Index(['Pregnant', 'Glucose', 'Diastolic_BP',  
                          'Skin_Fold', 'BMI',  
                          'Diabetes_Pedigree', 'Age', 'Class'],  
                          dtype='object')
```

Imputation with Mean/Median/Mode

When to Use: When the missing values are numerical and the missing data is random (MCAR), this method is suitable.

- Mean Imputation: Replace missing values with the mean of the column.
- Median Imputation: Use the median when data is skewed or has outliers.
- Mode Imputation: For categorical data, use the mode.

```
data = pd.DataFrame({  
    'A': [1, 2, None, 4],  
    'B': [5, None, 7, 8],  
    'C': ['cat', None, 'dog', 'cat']  
})  
  
# Mean imputation for numerical columns  
data['A'].fillna(data['A'].mean(), inplace=True)  
  
# Median imputation for numerical columns  
data['B'].fillna(data['B'].median(), inplace=True)
```

```
# Mode imputation for categorical columns
data['C'].fillna(data['C'].mode()[0], inplace=True)
```

```
print(data)
```

	A	B	C
0	1.000000	5.0	cat
1	2.000000	7.0	cat
2	2.333333	7.0	dog
3	4.000000	8.0	cat

C:\Users\Compumarts\AppData\Local\Temp\ipykernel_20332\174284825.py:8: FutureWarning: A value is trying to be set on a copy of a DataFrame or Series through chained assignment using an inplace method. The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are setting values always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] = df[col].method(value) instead, to perform the operation inplace on the original object.

```
data['A'].fillna(data['A'].mean(), inplace=True)
```

C:\Users\Compumarts\AppData\Local\Temp\ipykernel_20332\174284825.py:11: FutureWarning: A value is trying to be set on a copy of a DataFrame or Series through chained assignment using an inplace method. The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are setting values always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] = df[col].method(value) instead, to perform the operation inplace on the original object.

```
data['B'].fillna(data['B'].median(), inplace=True)
```

C:\Users\Compumarts\AppData\Local\Temp\ipykernel_20332\174284825.py:14: FutureWarning: A value is trying to be set on a copy of a DataFrame or Series through chained assignment using an inplace method. The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are setting values always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] =

`df[col].method(value)` instead, to perform the operation inplace on the original object.

```
data['C'].fillna(data['C'].mode()[0], inplace=True)
```

Imputation Using Forward/Backward Fill

When to Use: In time series data or ordered data, where the previous or next observation can provide a good estimate for the missing value.

- Forward Fill: Use the previous valid value to fill the missing data.
- Backward Fill: Use the next valid value to fill missing data.

```
# Example dataset (ordered)
data = pd.DataFrame({
    'A': [1, None, 3, None, 5],
    'B': [None, 6, None, 8, None]
})
```

```
# Forward fill
data_forward_fill = data.fillna(method='ffill')
```

```
# Backward fill
data_backward_fill = data.fillna(method='bfill')
```

```
print(data_forward_fill)
print(data_backward_fill)
```

	A	B
0	1.0	NaN
1	1.0	6.0
2	3.0	6.0
3	3.0	8.0
4	5.0	8.0

	A	B
0	1.0	6.0
1	3.0	6.0
2	3.0	8.0
3	5.0	8.0
4	5.0	NaN

```
C:\Users\Compumarts\AppData\Local\Temp\ipykernel_20332\711065662.py:8:
FutureWarning: DataFrame.fillna with 'method' is deprecated and will
raise in a future version. Use obj.ffill() or obj.bfill() instead.
```

```
data_forward_fill = data.fillna(method='ffill')
```

```
C:\Users\Compumarts\AppData\Local\Temp\
ipykernel_20332\711065662.py:11: FutureWarning: DataFrame.fillna with
'method' is deprecated and will raise in a future version. Use
```

```
obj.ffill() or obj.bfill() instead.  
data_backward_fill = data.fillna(method='bfill')
```

K-Nearest Neighbors (KNN) Imputation

- KNN imputation identifies the k-nearest samples and uses them to estimate the missing values.
- Pros: Captures relationships between variables and provides a more robust estimate than mean/median imputation.
- Cons: Computationally expensive, especially with large datasets; results may vary depending on the choice of k.

```
from sklearn.impute import KNNImputer  
  
# Example dataset  
data = pd.DataFrame({  
    'A': [1, 2, None, 4],  
    'B': [5, None, 7, 8],  
    'C': [9, 10, 11, None]  
})  
  
# KNN imputation (use n_neighbors=2, you can adjust this)  
imputer = KNNImputer(n_neighbors=2)  
data_imputed_knn = pd.DataFrame(imputer.fit_transform(data),  
                                columns=data.columns)  
  
print(data_imputed_knn)
```

	A	B	C
0	1.0	5.0	9.0
1	2.0	6.0	10.0
2	3.0	7.0	11.0
3	4.0	8.0	10.5

Key Considerations to Choose the Best Approach:

- Percentage of Missing Data:

Low (<5%): Consider deletion or simple imputation (mean/median/mode). Moderate (5-20%): Consider more sophisticated methods like KNN.

Type of Data:

- Numerical: Mean, median, KNN.
 - Categorical: Mode imputation, KNN imputation.
 - Time Series: Interpolation or forward/backward fill.
-

What is an outlier?

An outlier is a data point that significantly differs from other values in a dataset. Outliers can be either much larger or much smaller than most of the other data points. Example [160, 162, 165, 168, 170, 163, 164, 166, 167, 300]

In this dataset, most values are around 160-170 cm, but 300 cm is an extreme value compared to the rest of the data. This value is likely an outlier.

Ways to solve detect outliers?

Interquartile range (IQR) and z-score

Interquartile range (IQR)

- The **interquartile range (IQR)** is a measure of the spread of the middle 50% of the data.
- It is calculated as the difference between the **third quartile (Q3)** and the **first quartile (Q1)**.
 - **Q1 (First Quartile):** The median of the lower half of the data (25th percentile).
 - **Q3 (Third Quartile):** The median of the upper half of the data (75th percentile).

Formula:

$$\text{IQR} = \text{Q3} - \text{Q1}$$

Steps to Calculate IQR:

1. Arrange the data in ascending order.
2. Split the data into lower and upper halves.
 - If the number of data points is odd, exclude the median when splitting the data.
3. Find (Q1) (the median of the lower half) and (Q3) (the median of the upper half).
4. $\text{IQR} = \text{Q3} - \text{Q1}$

Example

Data set: 3, 5, 7, 8, 12, 13, 14, 18, 19, 21

Step-by-Step Process:

- Step 1: Arrange the data in ascending order: 3, 5, 7, 8, 12, 13, 14, 18, 19, 21
- Step 2: Find the first quartile (Q1) and third quartile (Q3):
Q1 is the median of the lower half:

Lower half = 3, 5, 7, 8, 12

Q1 = 7 (middle value of the lower half).

Q3 is the median of the upper half:

Upper half = 13, 14, 18, 19, 21

Q3 = 18 (middle value of the upper half).

- Step 3: Calculate the IQR = $Q3 - Q1 = 18 - 7 = 11$
- Step 4: Determine the outlier boundaries:

Lower Bound: Lower Bound = $Q1 - 1.5 \times IQR = 7 - 1.5 \times 11 = 7 - 16.5 = -9.5$

Upper Bound: Upper Bound = $Q3 + 1.5 \times IQR = 18 + 1.5 \times 11 = 34.5$

- Step 5: Identify the outliers:

Any value less than -9.5 or greater than 34.5 is considered an outlier.

In this case, no values are below -9.5 or above 34.5, so there are no outliers.

The z-score

z-score is a statistical measure that tells how many standard deviations a data point is from the mean of the dataset.

It is commonly used to detect **outliers**, which are data points that deviate significantly from the rest of the dataset.

□ Formula

$$z = \frac{x - \mu}{\sigma}$$

Where:

- x = the value to standardize
- μ = mean of the dataset
- σ = standard deviation of the dataset

Steps to compute the z-score for outlier detection:

1. Calculate the **mean** of your dataset.
2. Calculate the **standard deviation** of your dataset.
3. Compute the **z-score** for each data point using the formula above.

4. Set a **threshold**:

- Common thresholds are $(|z| > 3)$ or $(|z| > 2.5)$, depending on how strict you want to be in identifying outliers.

Example:

- If $(|z| > 3)$, the data point is considered an **outlier**.

Coding

```
import numpy as np
import pandas as pd
```

```
df=pd.read_csv("Data-Melbourne_F_fixed.csv")
df.head()
```

	Average Outflow	Average Inflow	Energy Consumption	Ammonia \
0	2.941	2.589	175856	27.0
1	2.936	2.961	181624	25.0
2	2.928	3.225	202016	42.0
3	2.928	3.354	207547	36.0
4	2.917	3.794	202824	46.0

	Biological Oxygen Demand	Chemical Oxygen Demand	Total Nitrogen \
0	365.0	730.0	60.378
1	370.0	740.0	60.026
2	418.0	836.0	64.522
3	430.0	850.0	63.000
4	508.0	1016.0	65.590

	Average Temperature
0	19.3
1	17.1
2	16.8
3	14.6
4	13.4

Common Methods to Detect Outliers

Using IQR (Interquartile Range):

The IQR method uses quartiles to determine the range of normal data. An outlier is typically defined as a point that lies below $Q1 - 1.5 * IQR$ or above $Q3 + 1.5 * IQR$.

```
data=df.copy()
def detect_outliers_iqr(column):
    Q1 = column.quantile(0.25)
    Q3 = column.quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
```

```

upper_bound = Q3 + 1.5 * IQR
# Return a boolean series indicating if the value is an outlier
return (column < lower_bound) | (column > upper_bound)

# Apply the IQR method to each feature in the DataFrame
for col in data.columns:
    data[f'{col}_Outlier'] = detect_outliers_iqr(data[col])

# Filter the DataFrame to display rows where any outlier flag is True
outliers_only = data[data.filter(like='_Outlier').any(axis=1)]

# Display the filtered DataFrame with outliers
outliers_only

```

	Average Outflow	Average Inflow	Energy Consumption	Ammonia \
9	3.912	3.425	202411	42.0
10	2.971	3.775	171123	47.0
11	3.833	3.763	193509	43.0
12	2.984	3.487	205761	38.0
15	2.935	3.481	173825	47.0
...	...	...	...	...
1342	0.002	5.208	257362	49.0
1358	5.502	4.453	285131	26.0
1359	5.392	4.930	328307	24.0
1362	4.846	4.219	343734	19.0
1364	3.954	4.907	259208	33.0

	Biological Oxygen Demand	Chemical Oxygen Demand	Total Nitrogen
\			
9	530.0	830.0	66.354
10	435.0	870.0	65.531
11	295.0	590.0	59.548
12	600.0	1200.0	68.820
15	470.0	980.0	76.000
...	...	...	...
1342	330.0	780.0	62.160
1358	190.0	610.0	48.000
1359	200.0	560.0	53.295
1362	170.0	640.0	59.797
1364	180.0	720.0	61.190

Average Temperature	Average Outflow_Outlier	Average
Inflow_Outlier \		
9	35.5	False
False		
10	32.1	False
False		
11	33.9	False
False		
12	19.2	False
False		
15	16.3	False
False		
...	...	...
...		
1342	15.4	True
False		
1358	10.3	False
False		
1359	9.1	False
False		
1362	10.6	False
False		
1364	8.1	False
False		

Energy Consumption_Outlier	Ammonia_Outlier \
9	False
10	False
11	False
12	False
15	False
...	...
1342	False
1358	False
1359	False
1362	False
1364	False

Biological Oxygen Demand_Outlier	Chemical Oxygen Demand_Outlier
\	
9	False
10	False
11	False
12	True
15	False

...	...	...
1342	False	False
1358	True	False
1359	False	False
1362	True	False
1364	True	False

	Total Nitrogen_Outlier	Average Temperature_Outlier
9	False	True
10	False	True
11	False	True
12	False	False
15	True	False
...	...	...
1342	False	False
1358	True	False
1359	True	False
1362	False	False
1364	False	False

[291 rows x 16 columns]

Using Visualization:

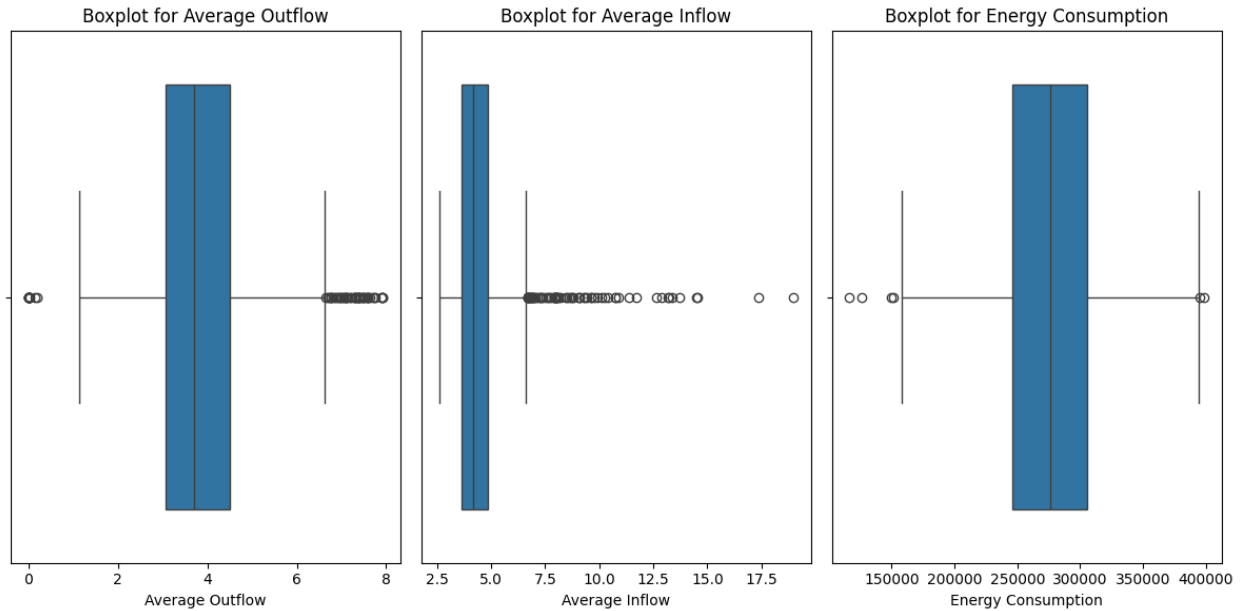
Visual methods like box plots and scatter plots can help in identifying outliers visually.

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

plt.figure(figsize=(12, 6))

for i, col in enumerate(df.columns[:3]): # Only plotting the first 3
    features
    plt.subplot(1, 3, i+1) # Create subplots for each feature
    sns.boxplot(x=df[col])
    plt.title(f"Boxplot for {col}")

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



Using Z-Score:

The Z-score measures how many standard deviations a data point is from the mean. A common threshold is a Z-score of ± 3 , which indicates that the data point is an outlier.

```
# Function to detect outliers in a column using Z-Score
data=df.copy()
def detect_outliers_zscore(column, threshold=3):
    mean = np.mean(column)
    std_dev = np.std(column)
    z_scores = (column - mean) / std_dev
    # Return a boolean series where True indicates an outlier
    return np.abs(z_scores) > threshold

# Apply the Z-Score method to each feature in the DataFrame
for col in data.columns:
    data[f'{col}_Outlier'] = detect_outliers_zscore(data[col])

# Filter the DataFrame to display rows where any outlier flag is True
outliers_only = data[data.filter(like='_Outlier').any(axis=1)]

outliers_only
```

	Average Outflow	Average Inflow	Energy Consumption	Ammonia	\
9	3.912000	3.425	202411	42.0	
10	2.971000	3.775	171123	47.0	
11	3.833000	3.763	193509	43.0	
15	2.935000	3.481	173825	47.0	
22	2.997000	3.526	214635	43.0	
...	...	...	...	...	

1320	0.000004	5.556	232271	35.0
1321	0.000004	5.299	263513	35.0
1341	0.000004	6.372	269271	43.0
1342	0.002000	5.208	257362	49.0
1358	5.502000	4.453	285131	26.0
Biological Oxygen Demand Chemical Oxygen Demand Total Nitrogen				
\				
9	530.0	830.0	66.354	
10	435.0	870.0	65.531	
11	295.0	590.0	59.548	
15	470.0	980.0	76.000	
22	344.0	688.0	63.521	
...	...	...	...	
1320	410.0	1000.0	64.477	
1321	280.0	720.0	61.866	
1341	320.0	880.0	61.000	
1342	330.0	780.0	62.160	
1358	190.0	610.0	48.000	
Average Temperature Average Outflow_Outlier Average				
Inflow_Outlier \				
9	35.5	False		
False				
10	32.1	False		
False				
11	33.9	False		
False				
15	16.3	False		
False				
22	31.7	False		
False				
...	...	...		
...				
1320	22.1	True		
False				
1321	14.1	True		
False				
1341	18.9	True		

False		
1342	15.4	True
False		
1358	10.3	False
False		

	Energy Consumption_Outlier	Ammonia_Outlier	\
9	False	False	
10	False	False	
11	False	False	
15	False	False	
22	False	False	
...	...	...	
1320	False	False	
1321	False	False	
1341	False	False	
1342	False	False	
1358	False	False	

	Biological Oxygen Demand_Outlier	Chemical Oxygen Demand_Outlier
\		
9	False	False
10	False	False
11	False	False
15	False	False
22	False	False
...	...	...
1320	False	False
1321	False	False
1341	False	False
1342	False	False
1358	False	False

	Total Nitrogen_Outlier	Average Temperature_Outlier
9	False	True
10	False	True
11	False	True
15	True	False
22	False	True
...	...	...

1320	False	False
1321	False	False
1341	False	False
1342	False	False
1358	True	False

[127 rows x 16 columns]

Handling Outliers

Replace outliers with NaN, then use methods for handling missing values to fill the NaN values.

```
data=df.copy()
def replace_outliers_with_nan(column):
    Q1 = column.quantile(0.25)
    Q3 = column.quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    # Replace outliers with NaN
    return column.where((column >= lower_bound) & (column <=
upper_bound), np.nan)

# Replace outliers with NaN for each feature
for col in data.columns:
    data[col] = replace_outliers_with_nan(data[col])

null_counts = data.isnull().sum()
print("Sum of null values in each column:")
print(null_counts)

Sum of null values in each column:
Average Outflow          74
Average Inflow           69
Energy Consumption        6
Ammonia                   11
Biological Oxygen Demand  58
Chemical Oxygen Demand    44
Total Nitrogen           103
Average Temperature       8
dtype: int64
```

We use K-Nearest Neighbors (KNN) Imputation to handle missing values

```
from sklearn.impute import KNNImputer
imputer = KNNImputer(n_neighbors=2)
data_imputed_knn = pd.DataFrame(imputer.fit_transform(data),
columns=data.columns)
data_imputed_knn.isnull().sum()
```

```
Average Outflow      0
Average Inflow        0
Energy Consumption    0
Ammonia               0
Biological Oxygen Demand 0
Chemical Oxygen Demand 0
Total Nitrogen        0
Average Temperature   0
dtype: int64
```

Remove outliers

```
df=pd.read_csv("Data-Melbourne_F_fixed.csv")
```

Using the Interquartile Range (IQR) Method

```
data=df.copy()

def remove_outliers_iqr(data):
    # Calculate Q1 (25th percentile) and Q3 (75th percentile) for each feature
    Q1 = data.quantile(0.25)
    Q3 = data.quantile(0.75)
    IQR = Q3 - Q1

    # Define lower and upper bounds
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    # Filter the dataset to remove rows with outliers in any column
    data_cleaned = data[~((data < lower_bound) | (data >
upper_bound)).any(axis=1)]

    return data_cleaned

cleaned_data = remove_outliers_iqr(data)

print("Original Data:")
print(len(data))
print("\nCleaned Data (without outliers):")
print(len(cleaned_data))

Original Data:
1382

Cleaned Data (without outliers):
1091
```

Using the Z-Score Method

```
data = df.copy() # Create a copy to avoid modifying the original dataframe

def remove_outliers_zscore(data, threshold=3):

    # Boolean DataFrame indicating if a value is an outlier
    outliers = pd.DataFrame(False, index=df.index, columns=df.columns)

    # Iterate through each numeric column
    for column in data.columns:
        if data[column].dtype in ['float64', 'int64']: # Only apply to numeric columns
            mean = data[column].mean()
            std_dev = data[column].std()
            z_scores = (data[column] - mean) / std_dev

            # Mark True in outliers DataFrame where abs(z_score) > threshold
            outliers[column] = abs(z_scores) > threshold

    # Filter the rows that have any True in the outliers DataFrame (indicating an outlier)
    rows_to_keep = ~outliers.any(axis=1)

    # Return the DataFrame with outlier rows removed
    return data[rows_to_keep]

# Example usage:
cleaned_data = remove_outliers_zscore(data, threshold=3)
print("Original Data:")
print(len(data))
print("\nCleaned Data (without outliers):")
print(len(cleaned_data))
```

```
Original Data:
1382
```

```
Cleaned Data (without outliers):
1257
```