

**POLITECHNIKA LUBELSKA**  
**WYDZIAŁ MATEMATYKI I INFORMATYKI**  
**TECHNICZNEJ**

**Kierunek: Inżynieria i Analiza Danych**



**Projekt Zaliczeniowy z Zakresu Eksploracji Danych**

***Praca wykonana przez:***

**Andrii Zapukhlyi, Nr albumu: s100935**

**Mateusz Drozd, Nr albumu: s100966**

# Zbiór Danych

Zbiór danych „Air Quality UCI” zawiera 9 358 instancji godzinnych uśrednionych odpowiedzi pięciu czujników półprzewodnikowych (metal-oxide) zainstalowanych w urządzeniu wieloczujnikowym do monitoringu jakości powietrza w terenie, wraz z jednoczesnymi pomiarami referencyjnymi stężeń zanieczyszczeń przez certyfikowany analizator.

Dane obejmują okres od marca 2004 do lutego 2005 (13 miesięcy).

Nr	Nazwa zmiennej	Opis
0	<b>Date</b>	Data pomiaru w formacie DD/MM/YYYY
1	<b>Time</b>	Godzina pomiaru w formacie HH.MM.SS
2	<b>True CO (mg/m<sup>3</sup>)</b>	Rzeczywiste stężenie tlenku węgla (CO) zmierzone przez analizator referencyjny
3	<b>PT08.S1 (tin oxide)</b>	Odpowiedź sensora tlenku cyny (nominalnie celowanego w CO)
4	<b>True NMHC (µg/m<sup>3</sup>)</b>	Rzeczywiste stężenie węglowodorów niemetanicznych (NMHC) przez analizator referencyjny
5	<b>True Benzene (µg/m<sup>3</sup>)</b>	Rzeczywiste stężenie benzenu przez analizator referencyjny
6	<b>PT08.S2 (titania)</b>	Odpowiedź sensora tlenku tytanu (nominalnie celowanego w NMHC)
7	<b>True NOx (ppb)</b>	Rzeczywiste stężenie tlenków azotu (NOx) w ppb przez analizator referencyjny
8	<b>PT08.S3 (tungsten oxide)</b>	Odpowiedź sensora tlenku wolframu (nominalnie celowanego w NOx)
9	<b>True NO<sub>2</sub> (µg/m<sup>3</sup>)</b>	Rzeczywiste stężenie dwutlenku azotu (NO <sub>2</sub> ) przez analizator referencyjny
10	<b>PT08.S4 (tungsten oxide)</b>	Odpowiedź sensora tlenku wolframu (nominalnie celowanego w NO <sub>2</sub> )
11	<b>PT08.S5 (indium oxide)</b>	Odpowiedź sensora tlenku indu (nominalnie celowanego w O <sub>3</sub> )
12	<b>Temperature (°C)</b>	Temperatura powietrza w stopniach Celsjusza
13	<b>Relative Humidity (%)</b>	Wilgotność względna powietrza w procentach
14	<b>Absolute Humidity</b>	Wilgotność bezwzględna (AH)

## Charakterystyka danych

- **Typ:** Wielozmienny, szereg czasowy.
- **Zadanie:** Regresja.
- **Typ cech:** wartości rzeczywiste (float).
- **Liczba instancji:** 9 358.
- **Liczba cech:** 15.

Brakujące wartości oznaczono wartością -200 w komórkach sensorów i analizatorów.

# Cel projektu

Celem projektu jest zbudowanie modelu regresyjnego, który na podstawie dostępnych pomiarów środowiskowych będzie w stanie jak najdokładniej przewidzieć godzinową wartość stężenia benenu ( C<sub>6</sub>H<sub>6</sub>(GT) ) w powietrzu. Benzen to toksyczny związek organiczny, uznawany za rakotwórczy i niebezpieczny dla zdrowia nawet w niskich stężeniach — dlatego jego skuteczne prognozowanie może mieć istotne znaczenie dla ochrony zdrowia publicznego, zarządzania jakością powietrza oraz systemów wczesnego ostrzegania.

Zmienna C<sub>6</sub>H<sub>6</sub>(GT) jest wartością ciągłą (wyrażoną w  $\mu\text{g}/\text{m}^3$ ), dlatego naturalnym podejściem analitycznym jest zastosowanie regresji nadzorowanej, której celem jest predykcja wartości liczbowej na podstawie zestawu cech wejściowych. Model może służyć m.in. do:

- przewidywania poziomu zanieczyszczenia w czasie rzeczywistym
- symulacji skutków zmian warunków atmosferycznych na jakość powietrza
- wspierania systemów ostrzegania w miastach i strefach przemysłowych

**W tym celu: wszystkie zmienne użyte do przewidywania ( C<sub>6</sub>H<sub>6</sub>(GT) ):**

- Odpowiedzi pięciu sensorów półprzewodnikowych ( PT08.S1(CO) , PT08.S2(NMHC) , PT08.S3(NOx) , PT08.S4(NO2) , PT08.S5(O3) )
- Pomiary referencyjne innych gazów ( CO(GT) , NMHC(GT) , NOx(GT) , NO2(GT) )
- Warunki meteorologiczne ( T , RH , AH )

In [2]:

```
import pandas as pd

df = pd.read_csv(
    'data\AirQualityUCI.csv',
    sep=';',
    decimal=',',
)
df.head().iloc[:,11:17]
```

Out[2]:

	PT08.S5(O3)	T	RH	AH	Unnamed: 15	Unnamed: 16
0	1268.0	13.6	48.9	0.7578	NaN	NaN
1	972.0	13.3	47.7	0.7255	NaN	NaN
2	1074.0	11.9	54.0	0.7502	NaN	NaN
3	1203.0	11.0	60.0	0.7867	NaN	NaN
4	1110.0	11.2	59.6	0.7888	NaN	NaN

Ponieważ kolumny "Unnamed: 15" i "Unnamed: 16" zawierają wyłącznie brakujące wartości (-200 lub NaN), usuwamy je przed dalszą analizą.

In [ ]:

```
import numpy as np
import pandas as pd
import scipy.stats as st
import seaborn as sns
df = df.drop(columns=["Unnamed: 15", "Unnamed: 16"])
df.replace(-200.0, np.nan, inplace=True)
df = df[:-114]
```

W ramach oczyszczania danych usunięto ostatnie 144 wiersze, które nie zawierały żadnych wartości (były w całości puste). Dzięki temu otrzymaliśmy spójniejszy zbiór, wolny od „pustych” rekordów, co ułatwia dalsze analizy statystyczne, wizualizacje oraz modelowanie.

In [ ]: df.info()

## 1. Podstawowe wymiary i typy danych

- **Liczba wierszy (entry):** 9 357
- **Liczba kolumn:** 15
- **Typy danych:**
  - 2 kolumny obiektowe ( object ): Date , Time
  - 13 kolumn numerycznych ( float64 ): pomiary gazów, sygnały czujników, T, RH, AH

## 2. Zestawienie braków danych

Kolumna	Non-Null	Braki (liczba)	Braki (%)
CO(GT)	7 674	1 683	18,0 %
PT08.S1(CO)	8 991	366	3,9 %
NMHC(GT)	914	8 443	90,2 %
C6H6(GT)	8 991	366	3,9 %
PT08.S2(NMHC)	8 991	366	3,9 %
NOx(GT)	7 718	1 639	17,5 %
PT08.S3(NOx)	8 991	366	3,9 %
NO2(GT)	7 715	1 642	17,6 %
PT08.S4(NO2)	8 991	366	3,9 %
PT08.S5(O3)	8 991	366	3,9 %
T	8 991	366	3,9 %
RH	8 991	366	3,9 %
AH	8 991	366	3,9 %

- **Najwięcej braków** w kolumnie NMHC(GT) (~90 %).
- **Pozostałe kolumny** poza kilkoma wartościami w CO(GT) , NOx(GT) , NO2(GT) i oczywiście w samych datach/czasie mają ok. 4 % braków.

In [ ]: df.shape

(9357, 15)

Nasz zbior ma 9357 kolumn i 15 wierszy.

# Tabela statystyk opisowych dla każdej zmiennej

Zmienna	Średnia	Min	Max	Odch. std.	Wariancja	Skośność	Kurtoza	Q1	Me
PT08.S3(NOx)	963.2975	461.00	1935.00	265.9142	70710.3448	0.8312	0.6530	769.0000	920
NO2(GT)	100.2600	19.00	196.00	31.4938	991.8609	0.0741	-0.1430	78.5000	99
C6H6(GT)	10.7711	0.50	39.20	7.4181	55.0287	0.9964	0.7950	4.8000	5
CO(GT)	2.3536	0.30	8.10	1.4095	1.9867	1.0643	1.0700	1.3000	2
NMHC(GT)	231.0254	7.00	1189.00	208.4619	43456.3686	1.4826	1.9560	77.0000	157
PT08.S2(NMHC)	966.1161	448.00	1754.00	266.4246	70982.0446	0.3504	-0.4220	754.0000	942
PT08.S1(CO)	1207.8791	753.00	2040.00	241.8170	58475.4599	0.5644	-0.1030	1017.0000	1172
NOx(GT)	143.5018	12.00	478.00	81.8297	6696.1026	0.8645	0.4560	81.0000	128
PT08.S4(NO2)	1600.6203	955.00	2679.00	302.2918	91380.3278	0.6900	0.2730	1369.5000	1556
PT08.S5(O3)	1045.8126	263.00	2359.00	400.1347	160107.7481	0.3372	-0.3510	760.0000	1009
AH	0.8319	0.4023	1.4852	0.1785	0.0319	0.7323	0.8370	0.7189	0
T	15.6015	6.3000	30.0000	4.8253	23.2836	0.6252	0.0700	11.9000	15
RH	49.0502	14.9000	83.2000	15.2667	233.0735	0.0081	-0.8650	36.7000	49

## Wnioski z tabeli statystyk opisowych:

### 1. Zróżnicowanie skali i rozrzutu

- Największe wartości średnie i rozrzut mają **czujniki PT08.S1–PT08.S5** (zakresy sygnałów od ~250 do ponad 2 600 jednostek ADC). Ich odchylenia standardowe sięgają kilkuset, a wariancje dziesiątek tysięcy, co sugeruje wysoką zmienność w funkcji warunków atmosferycznych i stężeń.
- W porównaniu do nich **stężenia gazów** (CO, NO<sub>2</sub>, NO<sub>x</sub>, C6H6, NMHC) mają znacznie mniejsze zakresy wartości oraz relativnie mniejsze odchylenia standardowe.

### 2. Asymetria i „ogonki” rozkładów

- Najsilniej prawoskośne są:
  - CO(GT)** (skośność ≈ 1,06)
  - C6H6(GT)** (≈ 0,996)
  - NMHC(GT)** (≈ 1,48) co oznacza, że od czasu do czasu pojawiają się bardzo wysokie pikowe wartości.
- Z kolei **wilgotność względna RH** ma prawie symetryczny rozkład (skośność ≈ 0,01), a **PT08.S2(NMHC)** i **PT08.S5(O<sub>3</sub>)** są nawet lekko lewoskośne (skośność ujemna).

### 3. „Spiczastość” rozkładów (kurtoza)

- Najwyższą kurtozę (tzw. „spiczastość”) ma **NMHC(GT)** ( $\approx 1,96$ ) – rozkład z ciężkimi ogonami i wyraźnym stożkowatym kształtem.
- Rozkład RH jest bardziej płaski niż normalny (kurtoza  $\approx -0,865$ ), co znaczy, że wartości skrajne trafiają się częściej niż w rozkładzie normalnym.

## 4. Kwantyle i mediany

- Dla większości zmiennych mediana (Q2) znajduje się znaczco poniżej średniej w przypadku prawoskośnych rozkładów, co wskazuje, że częściej spotykamy mniejsze wartości, a wysokie obserwacje podnoszą średnią.
- Zmienne bliskie symetrii (RH, NO2) mają medianę niemal równą średniej.

```
In [4]: import pandas as pd
```

```
df = pd.read_csv(  
    'data\AirQualityUCI.csv',  
    sep=';',  
    decimal=',',  
)  
df.head().iloc[:,11:17]  
df1=df.copy()
```

```
In [5]: import numpy as np
```

```
df = df.drop(columns=["Unnamed: 15", "Unnamed: 16", "Date", "Time"])  
df.replace(-200.0, np.nan, inplace=True)  
df = df[:-114]
```

```
In [6]: df=df.dropna()
```

# Zmienna PT08.S3(Nox)

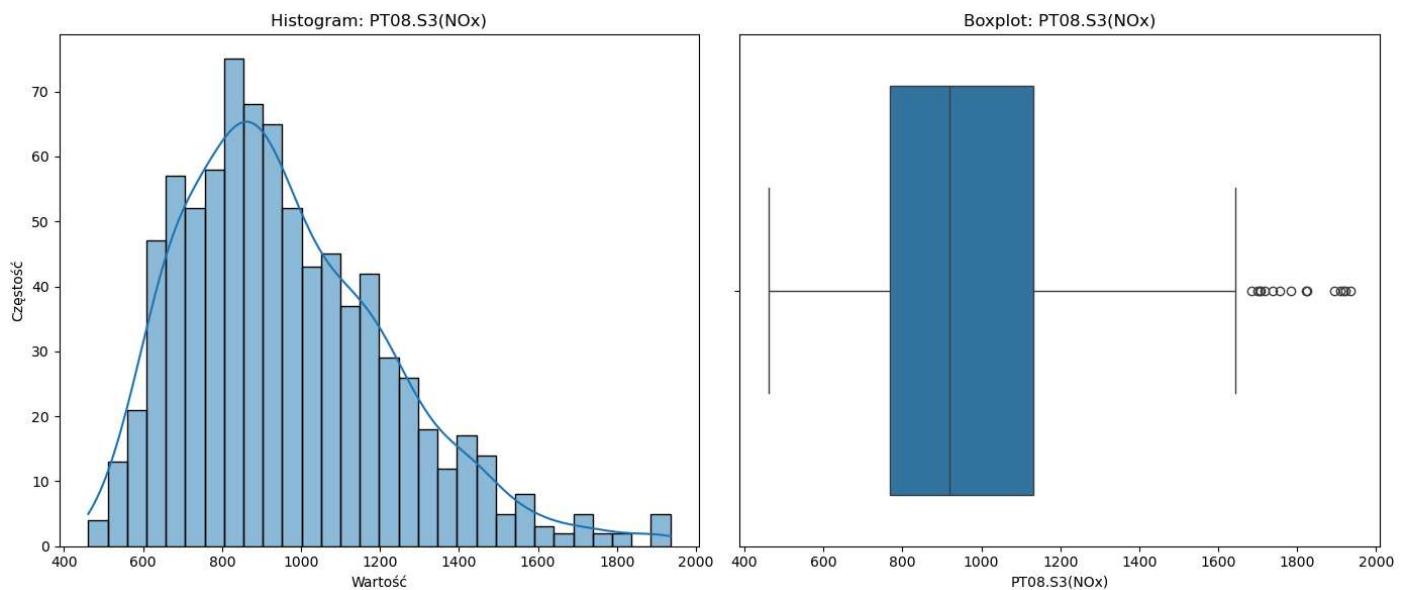
```
In [ ]: arr = df['PT08.S3(Nox)']
```

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

```
plt.figure(figsize=(14,6))
plt.subplot(1,2,1)
sns.histplot(arr, kde=True, bins=30)
plt.title("Histogram: PT08.S3(Nox)")
plt.xlabel("Wartość")
plt.ylabel("Częstość")

plt.subplot(1,2,2)
sns.boxplot(x=arr)
plt.title("Boxplot: PT08.S3(Nox)")
plt.xlabel("PT08.S3(Nox)")

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



## Histogram

- **Kształt rozkładu:** wyraźny szczyt (moda) w okolicach 800–1 000, a dalej gwałtowny spadek częstości — większość odczytów mieści się między ~500 a ~1 400.
- **Skośność:** rozkład przesunięty w prawo (pojedyncze sporadyczne, wysokie odczyty NOx powodują długi ogon).
- **Gęstość:** dolne wartości (< 500) i bardzo wysokie (> 1 500) zdarzają się rzadko.

## Boxplot

- **Mediana (linia wewnętrz pudełka):** wynosi 920 , co oznacza, że połowa wszystkich odczytów jest mniejsza niż ta wartość.
- **Wartości w przedziale między 25% (769) a 75% (1131)** stanowią większość danych (IQR – interquartile range).
- **Wartości odstające (outliers):** liczne punkty poza wąsami po prawej stronie – świadczą o sporadycznych skokowych wzrostach odczytu NOx, które mogą być spowodowane np. nagłą zmianą warunków atmosferycznych.

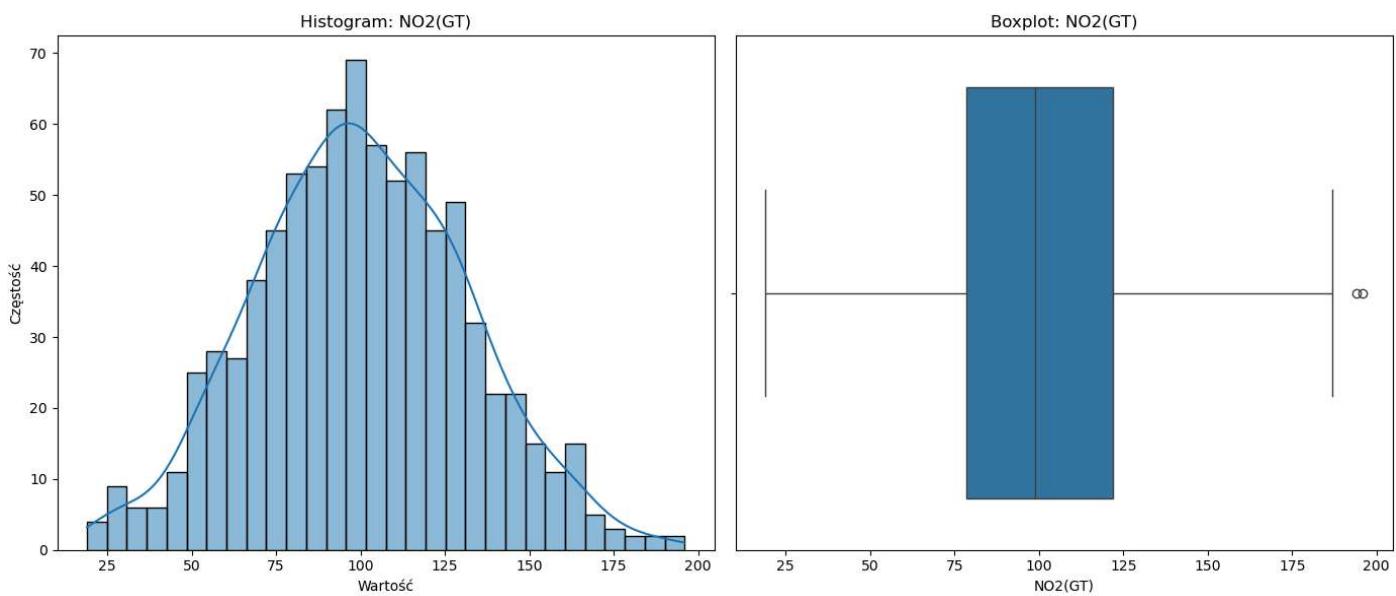
# Zmienna NO2(GT)

In [ ]:

```
arr1 = df['NO2(GT)']
plt.figure(figsize=(14,6))
plt.subplot(1,2,1)
sns.histplot(arr1, kde=True, bins=30)
plt.title("Histogram: NO2(GT)")
plt.xlabel("Wartość")
plt.ylabel("Częstość")

plt.subplot(1,2,2)
sns.boxplot(x=arr1)
plt.title("Boxplot: NO2(GT)")
plt.xlabel("NO2(GT)")

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



## Analiza wykresów dla zmiennej NO2(GT)

### Histogram: NO2(GT)

- **Kształt rozkładu**

Rozkład jest lekko prawoskony, z większością obserwacji skupionych pomiędzy 60 a 140.

- **Modalna wartość**

Najwięcej godzinnych odczytów wypada około 90–110.

- **Rozkładu**

Rozkład obserwacji przypomina rozkład normalny.

- **Gęstość i rozrzut**

Dolne wartości (< 30 ) oraz bardzo wysokie (> 200) są rzadkie.

## Boxplot: NO2(GT)

- **Mediana** wynosi 99, co oznacza, że połowa pomiarów ma stężenie benzenu poniżej tej wartości.
- **Wartości w przedziale między 25% (78) a 75% (122)** stanowią większość danych (IQR – interquartile range).
- **Wartości odstające** liczne punkty powyżej górnego wąsa świadczą o kilkunastu skokowych wzrostach stężenia NO<sub>2</sub>.

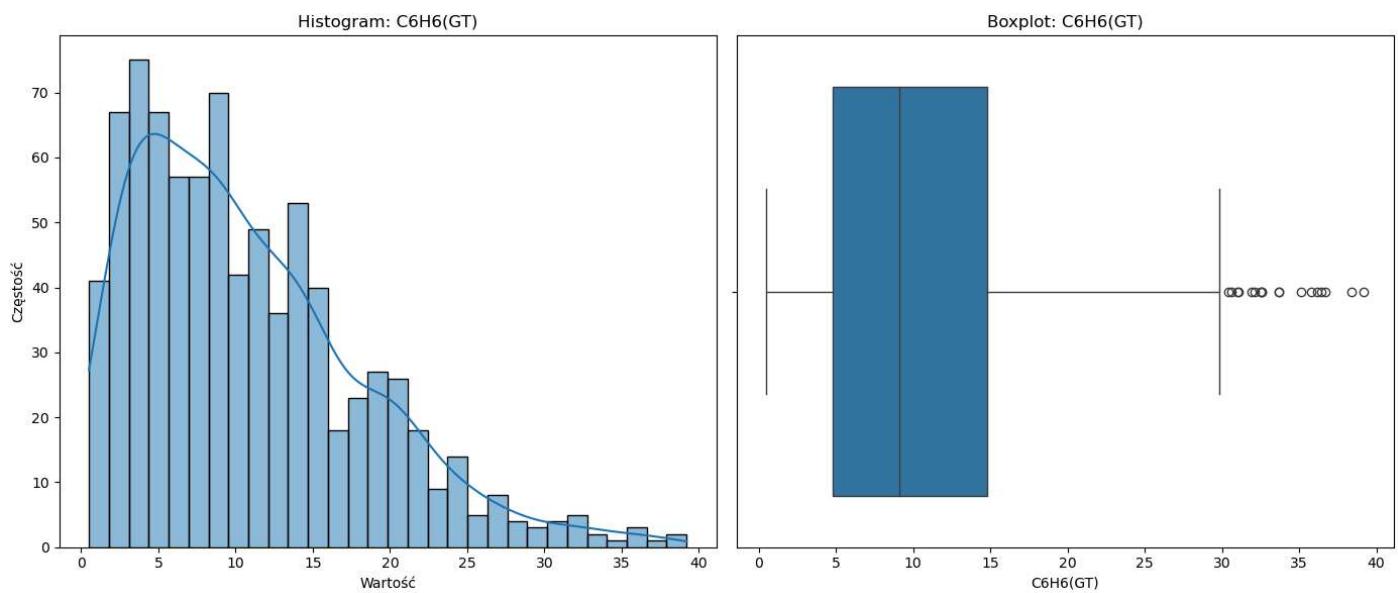
# Zmienna C6H6(GT)

In [ ]:

```
arr2 = df['C6H6(GT)']
plt.figure(figsize=(14,6))
plt.subplot(1,2,1)
sns.histplot(arr2, kde=True, bins=30)
plt.title("Histogram: C6H6(GT)")
plt.xlabel("Wartość")
plt.ylabel("Częstość")

plt.subplot(1,2,2)
sns.boxplot(x=arr2)
plt.title("Boxplot: C6H6(GT)")
plt.xlabel("C6H6(GT)")

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



## Analiza wykresów dla zmiennej C6H6(GT)

### Histogram: C6H6(GT)

- **Kształt rozkładu**

Wyraźnie prawoskony rozkład – większość wartości skupia się przy niskich stężeniach benzenu, natomiast sporadyczne obserwacje o wysokich wartościach tworzą długi prawy ogon.

- **Modalna wartość**

Najwięcej odczytów wypada w okolicach 4–8.

- **Ogon rozkładu**

Stężenia powyżej ~25 są rzadkie, ale sięgają nawet do 40.

## Boxplot: C6H6(GT)

- **Mediana** wynosi 9.1, co oznacza, że połowa pomiarów ma stężenie benzenu poniżej tej wartości.
- **Wartości w przedziale między 25% (4.8) a 75% (14.8)** stanowią większość danych (IQR – interquartile range).
- **Wartości odstające** Liczne punkty powyżej górnego wąsa wskazują na sporadyczne epizody podwyższonego stężenia benzenu, które mogą wynikać z lokalnych zdarzeń (np. wzmożony ruch samochodowy, prace remontowe, itp.).

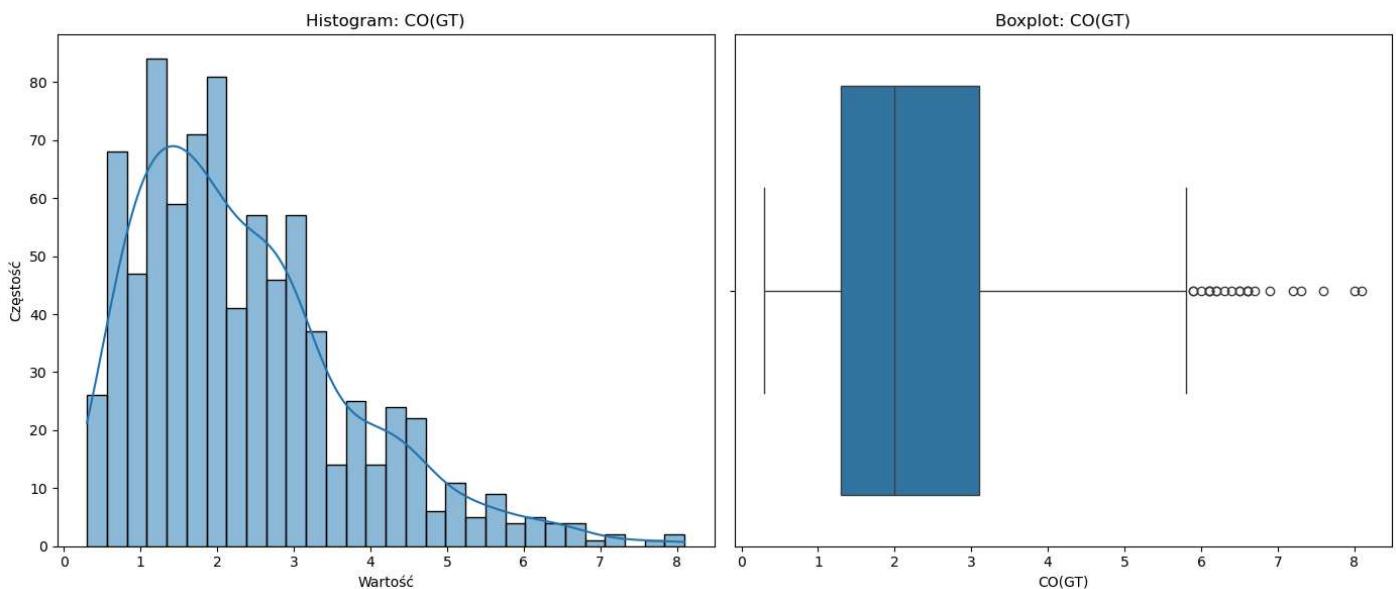
# Zmienna CO(GT)

```
In [ ]: arr1 = df['CO(GT)']
```

```
In [ ]: plt.figure(figsize=(14,6))
plt.subplot(1,2,1)
sns.histplot(arr1, kde=True, bins=30)
plt.title("Histogram: CO(GT)")
plt.xlabel("Wartość")
plt.ylabel("Częstość")

plt.subplot(1,2,2)
sns.boxplot(x=arr1)
plt.title("Boxplot: CO(GT)")
plt.xlabel("CO(GT)")

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



## Analiza wykresów dla zmiennej CO(GT)

### Histogram: CO(GT)

- **Kształt rozkładu**

Silnie **prawoskóry rozkład** – ponad 80% danych skupia się w przedziale **0–4**, z częstotliwością sięgającą **70** dla najniższych wartości (1–2).

**Ogon rozkładu:** Długi, rozciągający się do **8**, ale wartości powyżej **5** występują rzadko (częstotliwość spada do 10).

- **Dominujący zakres**

Wartości **1–3** dominują, co sugeruje, że większość pomiarów rejestruje **bardzo niskie stężenia CO**.

- **Implikacje**

Niskie średnie stężenie CO może wskazywać na ogólnie dobrą jakość powietrza, ale nie eliminuje ryzyka krótkotrwałych skoków.

## Boxplot: CO(GT)

- **Mediana**

Wynosi ok. **2** – połowa pomiarów nie przekracza tego poziomu.

- **Rozstęp międzykwartylowy (IQR)**

Przedział **25–75%** prawdopodobnie obejmuje wartości **1.3–3.1**, co potwierdza koncentrację danych przy dolnych wartościach.

- **Wartości odstające**

Punkty powyżej **6–8** wskazują na **incydentalne skoki stężenia CO**. Mogą wynikać np. z **Błędów pomiarowych** (wymagają weryfikacji).

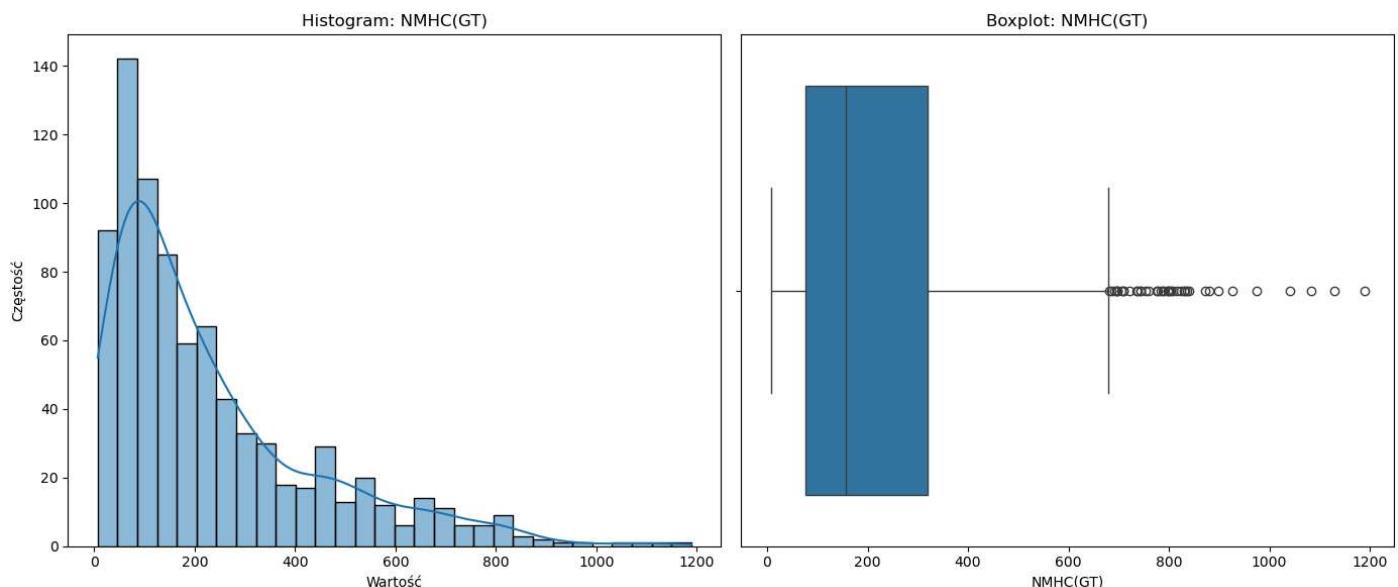
# Zmienna NMHC(GT)

```
In [ ]: arr1 = df['NMHC(GT)']
```

```
In [ ]: plt.figure(figsize=(14,6))
plt.subplot(1,2,1)
sns.histplot(arr1, kde=True, bins=30)
plt.title("Histogram: NMHC(GT)")
plt.xlabel("Wartość")
plt.ylabel("Częstość")

plt.subplot(1,2,2)
sns.boxplot(x=arr1)
plt.title("Boxplot: NMHC(GT)")
plt.xlabel("NMHC(GT)")

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



## Analiza wykresów dla zmiennej NMHC(GT) (węglowodory niemetanowe)

### Histogram: NMHC(GT)

- **Kształt rozkładu:**
  - **Prawoskoność** – większość danych skupia się w niższych zakresach (0–200), z długim ogonem sięgającym **1200**.
  - **Maksymalna częstotliwość**: Wartości w przedziale **50–150** osiągają najwyższą częstotliwość, co wskazuje na dominację umiarkowanych stężeń.
  - **Rzadkie wysokie wartości**: Powyżej **800** częstotliwość gwałtownie spada, ale ekstremalne pomiary (do **1200**) sugerują incydentalne emisje.

## Boxplot: NMHC(GT)

- **Statystyki opisowe:**

- **Medianą:** Wynosi **157** – połowa pomiarów jest niższa od tej wartości.
- **Rozstęp międzykwartylowy (IQR):** Przedział **77–318.5** obejmuje 50% danych, co wskazuje na umiarkowaną zmienność.
- **Wartości odstające:** Punkty powyżej **700** (sięgające **1200**) – mogą być związanymi z awariami przemysłowymi, intensywnym ruchem drogowym lub spalaniem odpadów.

- **Implikacje środowiskowe:**

- NMHC są prekursorami ozonu przygruntowego – nawet średnie stężenia wymagają monitorowania.
- Skrajne wartości ( $>1000$ ) mogą wskazywać na lokalne źródła zanieczyszczeń, np. rafinerie lub magazyny paliw.

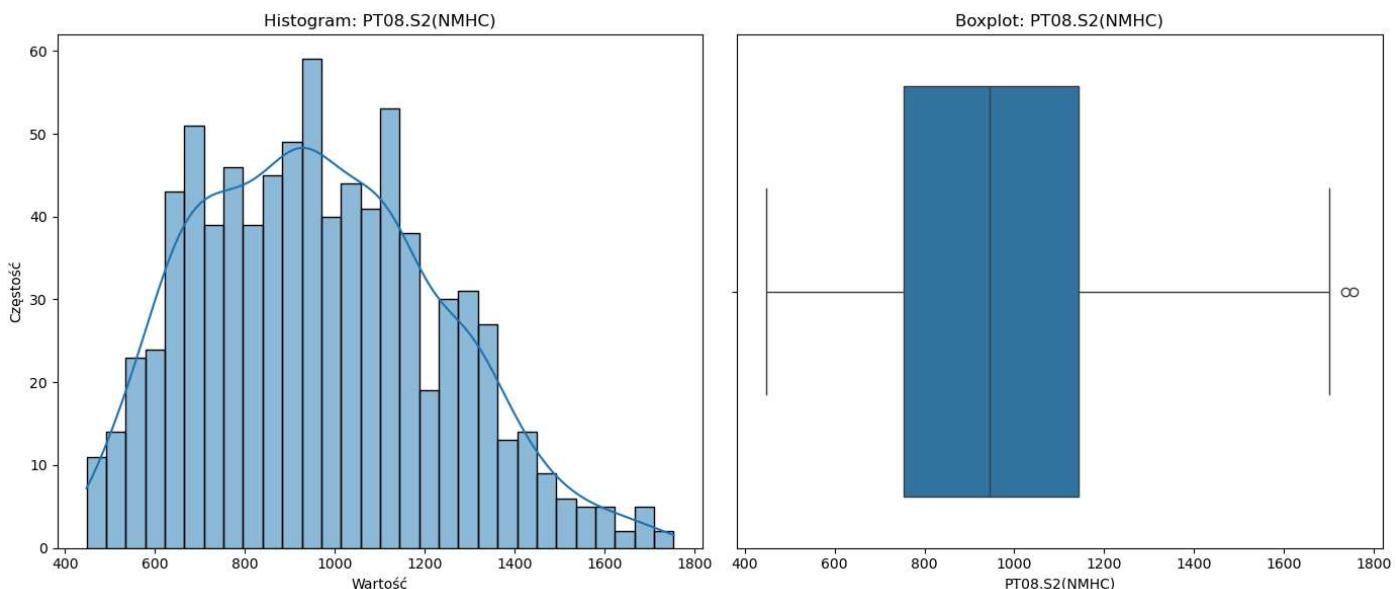
# Zmienna PT08.S2(NMHC)

```
In [ ]: arr1 = df['PT08.S2(NMHC)']
```

```
In [ ]: plt.figure(figsize=(14,6))
plt.subplot(1,2,1)
sns.histplot(arr1, kde=True, bins=30)
plt.title("Histogram: PT08.S2(NMHC)")
plt.xlabel("Wartość")
plt.ylabel("Częstość")

plt.subplot(1,2,2)
sns.boxplot(x=arr1)
plt.title("Boxplot: PT08.S2(NMHC)")
plt.xlabel("PT08.S2(NMHC)")

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



## Analiza wykresów dla zmiennej PT08.S2(NMHC) (prawdopodobnie pomiar czujnika NMHC)

### Histogram: PT08.S2(NMHC)

- **Kształt rozkładu:**
  - **Silna prawoskośność** – większość danych skupia się w przedziale **800–1100**, z maksymalną częstotliwością ok. **50** dla wartości ok. **950**.
  - **Ogon rozkładu**: Długi, rozciągający się do **1700**, ale powyżej **1600** częstotliwość spada prawie do zera.
- **Dominujący zakres:**
  - Ponad 70% pomiarów mieści się w przedziale **500–1400**, co sugeruje stabilne warunki pomiarowe dla większości czasu.
- **Anomalia:**
  - Brak danych poniżej **448** – może wynikać z zakresu czułości czujnika lub braku emisji w niższych wartościach.

## Boxplot: PT08.S2(NMHC)

- **Statystyki opisowe:**

- **Medianą:** Szacowana na **944** – połowa pomiarów jest niższa od tej wartości.
- **Rozstęp międzykwartylowy (IQR):** Przedział **754–1142.5** obejmuje 50% centralnych danych, co wskazuje na umiarkowaną zmienność.
- **Wartości odstające:** Punkty powyżej **1700** – mogą wynikać z **Chwilowych zakłóceń** (np. błędy czujnika, kurz).

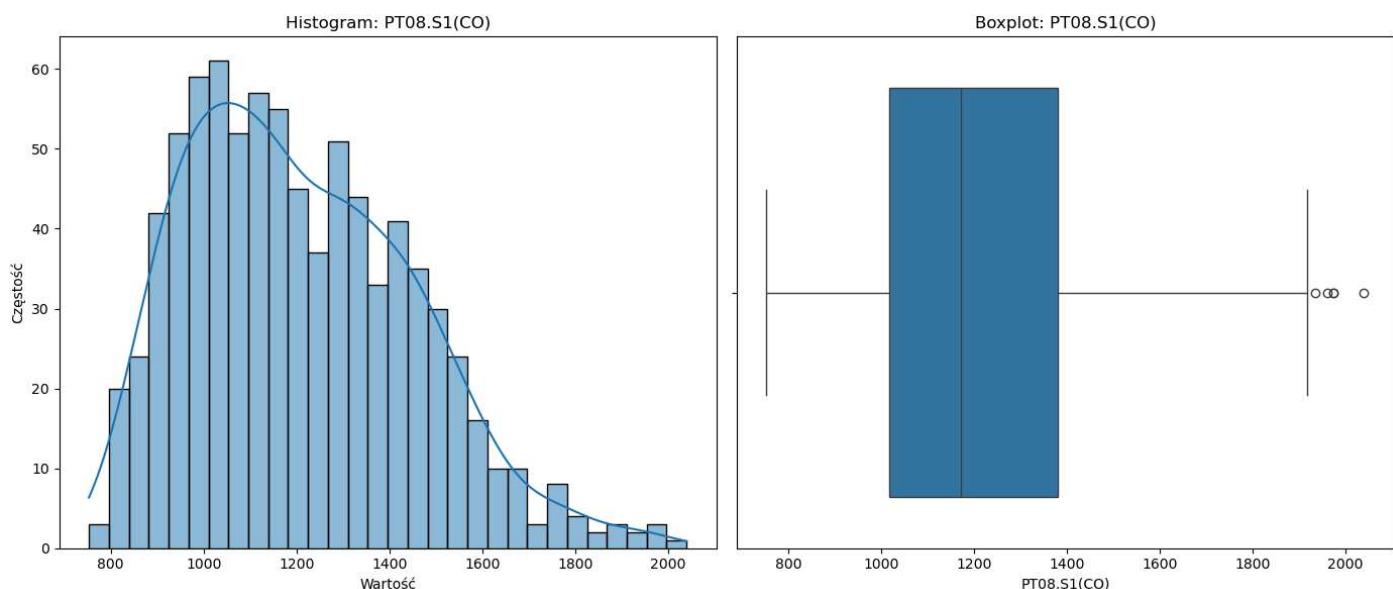
# Zmienna PT08.S1(CO)

```
In [ ]: arr1 = df['PT08.S1(CO)']
```

```
In [ ]: plt.figure(figsize=(14,6))
plt.subplot(1,2,1)
sns.histplot(arr1, kde=True, bins=30)
plt.title("Histogram: PT08.S1(CO)")
plt.xlabel("Wartość")
plt.ylabel("Częstość")

plt.subplot(1,2,2)
sns.boxplot(x=arr1)
plt.title("Boxplot: PT08.S1(CO)")
plt.xlabel("PT08.S1(CO)")

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



## Analiza wykresów dla zmiennej PT08.S1(CO) (pomiar czujnika CO)

### Histogram: PT08.S1(CO)

- **Kształt rozkładu:**
  - **Lewoskośność** – najwyższa częstotliwość (**55**) występuje dla wartości (ok. **1100**), a następnie maleje wraz ze wzrostem stężenia CO.
  - **Brak danych powyżej 2040 oraz poniżej 753** – zakres pomiarowy czujnika lub rzeczywisty brak stężeń.
- **Dominujący zakres:**
  - Ponad 70% danych skupia się w przedziale **900–1100**, co wskazuje na przewagę niskich i umiarkowanych stężeń CO.
- **Anomalie:**
  - Etykieta "Crystoid" (prawdopodobnie błąd dla "Częstość") sugeruje problem z formatowaniem osi Y.

## Boxplot: PT08.S1(CO)

- **Statystyki opisowe:**

- **Medianą:** Wynosi **1172** – połowa pomiarów jest niższa od tej wartości.
- **Rozstęp międzykwartylowy (IQR):** Przedział **1017–1380** obejmuje 50% centralnych danych, co wskazuje na umiarkowaną zmienność.
- **Wartości odstające:** Punkty powyżej **1900** – mogą wynikać z **Krótkotrwałych emisji** (np. korki drogowe, awarie systemów wentylacyjnych).

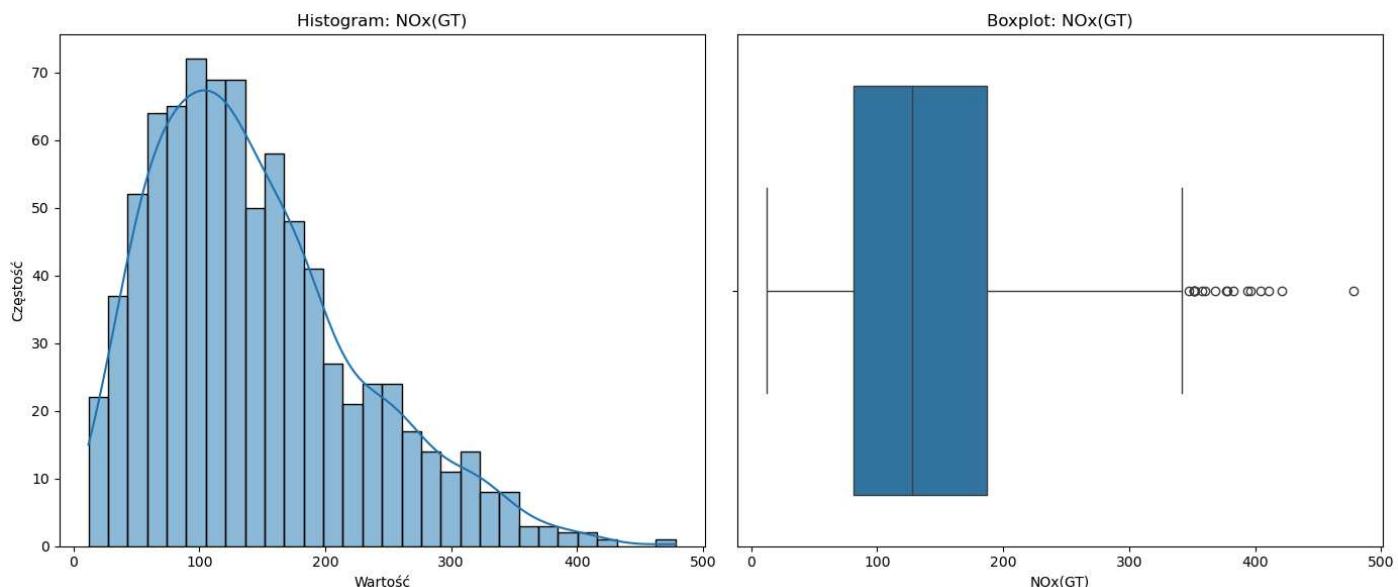
# Zmienna NOx(GT)

```
In [ ]: arr1 = df['NOx(GT)']
```

```
In [ ]: plt.figure(figsize=(14,6))
plt.subplot(1,2,1)
sns.histplot(arr1, kde=True, bins=30)
plt.title("Histogram: NOx(GT)")
plt.xlabel("Wartość")
plt.ylabel("Częstość")

plt.subplot(1,2,2)
sns.boxplot(x=arr1)
plt.title("Boxplot: NOx(GT)")
plt.xlabel("NOx(GT)")

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



## Analiza wykresów dla zmiennej NOx(GT) (tlenki azotu)

### Histogram: NOx(GT)

- **Kształt rozkładu:**
  - **Silna prawoskośność** – większość danych koncentruje się w przedziale **0–400**, z maksymalną częstotliwością ok. **67** dla wartości ok. **110**.
  - **Ogon rozkładu:** Długi, rozciągający się do **500**, ale wartości powyżej **300** występują rzadko (częstotliwość spada prawie do zera).
- **Dominujący zakres:**
  - Ponad **80% pomiarów** mieści się w przedziale **0–300**, co wskazuje na przewagę niskich i umiarkowanych stężeń NOx.
- **Anomalie:**
  - Brak danych powyżej **478** – może wynikać z ograniczeń czujnika lub braku ekstremalnych emisji.

## Boxplot: NOx(GT)

- **Statystyki opisowe:**

- **Medianą:** Wynosi **128** – połowa pomiarów jest niższa od tej wartości.
- **Rozstęp międzykwartylowy (IQR):** Przedział **81–187** obejmuje 50% centralnych danych, co sugeruje umiarkowaną zmienność.
- **Wartości odstające:** Punkty powyżej **350** (sięgające **500**) – mogą być związane z **Intensywnym ruchem drogowym** (szczególnie w godzinach szczytu).

- **Implikacje środowiskowe:**

- NOx są kluczowe dla powstawania smogu i kwaśnych deszczów – nawet średnie stężenia wymagają monitorowania.

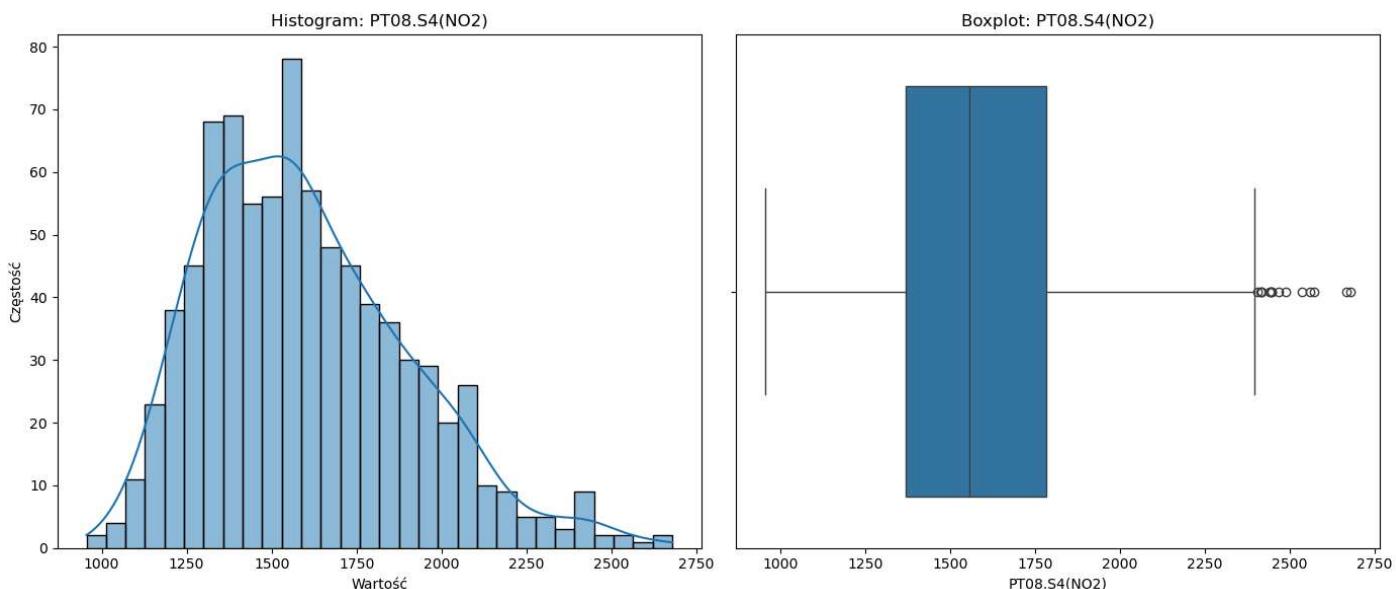
# Zmienna PT08.S4(NO2)

```
In [ ]: arr1 = df['PT08.S4(NO2)']
```

```
In [ ]: plt.figure(figsize=(14,6))
plt.subplot(1,2,1)
sns.histplot(arr1, kde=True, bins=30)
plt.title("Histogram: PT08.S4(NO2)")
plt.xlabel("Wartość")
plt.ylabel("Częstość")

plt.subplot(1,2,2)
sns.boxplot(x=arr1)
plt.title("Boxplot: PT08.S4(NO2)")
plt.xlabel("PT08.S4(NO2)")

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



## Analiza wykresów dla zmiennej PT08.S4(NO2) (pomiar czujnika NO2)

### Histogram: PT08.S4(NO2)

- **Kształt rozkładu:**
  - **Silna prawoskośność** – najwyższa częstotliwość ok. (62) występuje dla wartości **1250–1650**, po czym gwałtownie spada w kierunku wyższych stężeń.
  - **Ogon rozkładu:** Cienki, rozciągający się do **2700**, ale wartości powyżej **2100** są rzadkie (częstotliwość spada do poniżej 10).
- **Dominujący zakres:**
  - Ponad **70% danych** skupia się w przedziale **110–2100**, co wskazuje na przewagę **średnich stężeń NO2** w badanym okresie.
- **Anomalie:**
  - Brak danych poniżej **955** – może wynikać z dolnego progu czułości czujnika lub braku emisji w niższych zakresach.

## Boxplot: PT08.S4(NO2)

- **Statystyki opisowe:**
  - **Medianą:** Wynosi **1556** – połowa pomiarów jest niższa od tej wartości.
  - **Rozstęp międzykwartylowy (IQR):** Przedział **1369–1785** obejmuje 50% centralnych danych, co sugeruje **znaczną zmienność**.
  - **Wartości odstające:** Punkty powyżej **2400** (sięgające **2700**) – mogą być związane z **Emisjami przemysłowymi** (np. spalanie paliw stałych).
- **Implikacje środowiskowe:**
  - NO2 jest kluczowy dla powstawania **smogu** i negatywnego wpływu na drogi oddechowe – skrajne wartości ( $>2400$ ) wymagają natychmiastowej interwencji.

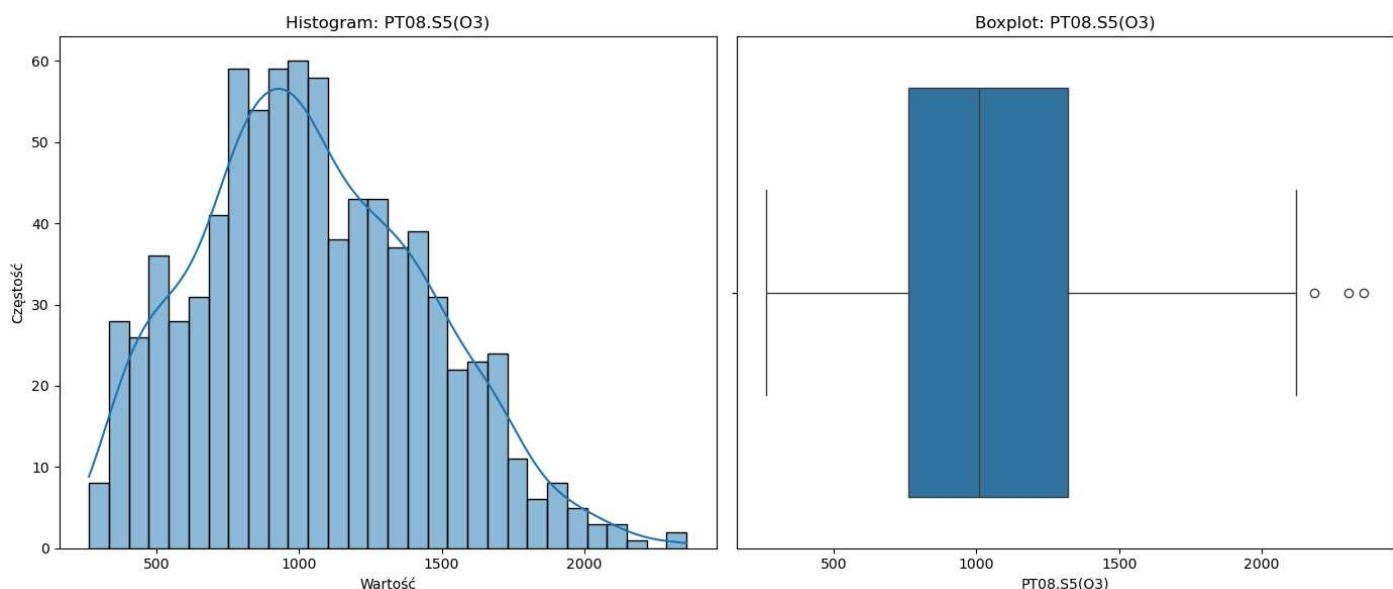
# Zmienna PT08.S5(O3)

```
In [ ]: arr1 = df['PT08.S5(O3)']
```

```
In [ ]: plt.figure(figsize=(14,6))
plt.subplot(1,2,1)
sns.histplot(arr1, kde=True, bins=30)
plt.title("Histogram: PT08.S5(O3)")
plt.xlabel("Wartość")
plt.ylabel("Częstość")

plt.subplot(1,2,2)
sns.boxplot(x=arr1)
plt.title("Boxplot: PT08.S5(O3)")
plt.xlabel("PT08.S5(O3)")

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



## Analiza wykresów dla zmiennej PT08.S5(O3) (pomiar ozonu)

### Histogram: PT08.S5(O3)

- **Kształt rozkładu:**
  - **Silna prawoskośność** – większość danych skupia się w przedziale **500–1000**, z maksymalną częstotliwością dla wartości ok. **60**.
  - **Ogon rozkładu:** Długi, rozciągający się do **2359**, ale wartości powyżej **1900** występują rzadko.
- **Dominujący zakres:**
  - Ponad **60%** pomiarów mieści się w przedziale **500–1500**, co sugeruje przewagę **nisko-średnich stężeń ozonu**.
- **Anomalie:**
  - Brak danych poniżej **263** – może wynikać z dolnego progu czułości czujnika lub rzeczywistego braku emisji.

## Boxplot: PT08.55(O3)

- **Statystyki opisowe:**

- **Medianą:** Szacowana na ~1009 – połowa pomiarów jest niższa od tej wartości.
- **Rozstęp międzykwartylowy (IQR):** Przedział 760–1320 obejmuje 50% danych, co wskazuje na znaczną zmienność.
- **Wartości odstające:** Punkty powyżej 2100 – mogą być związane z Epizodami fotochemicznymi (np. upalne dni z wysokim nasłonecznieniem).

### Implikacje środowiskowe:

- Wysokie stężenia O<sub>3</sub> są szkodliwe dla zdrowia i ekosystemów – wymagają alertów jakości powietrza.

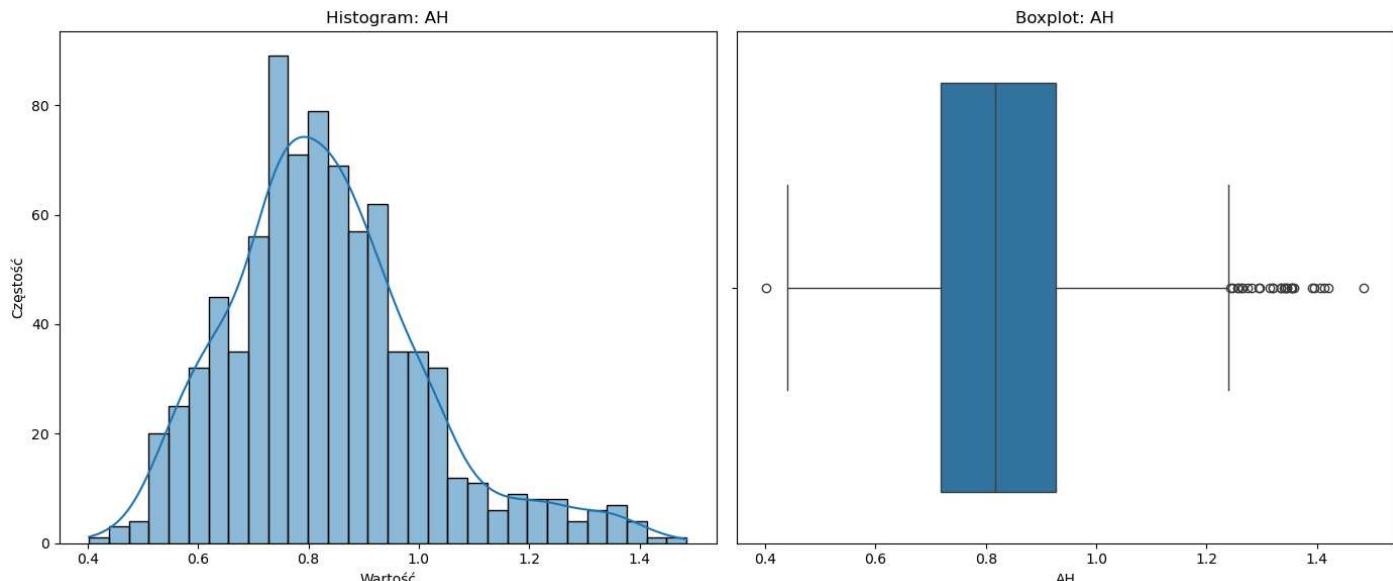
# Zmienna AH

```
In [ ]: arr1 = df['AH']
```

```
In [ ]: plt.figure(figsize=(14,6))
plt.subplot(1,2,1)
sns.histplot(arr1, kde=True, bins=30)
plt.title("Histogram: AH")
plt.xlabel("Wartość")
plt.ylabel("Częstość")

plt.subplot(1,2,2)
sns.boxplot(x=arr1)
plt.title("Boxplot: AH")
plt.xlabel("AH")

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



## Analiza wykresów dla zmiennej AH (wilgotność bezwzględna)

### Histogram: AH

- **Kształt rozkładu:**
  - **Silna prawoskość** – większość danych skupia się w przedziale **0.75–0.9**, z maksymalną częstotliwością ok. **70** dla wartości **0.8**.
  - **Ogon rozkładu:** Krótki, rozciągający się do ok. **1.5**, ale wartości powyżej **1.1** występują bardzo rzadko (częstotliwość spada prawie do 0).
- **Dominujący zakres:**
  - Ponad **90% pomiarów** mieści się w przedziale **0.7–1.1**, co wskazuje na przewagę **niskiej i umiarkowanej wilgotności**.
- **Anomalie:**
  - Brak danych poniżej **0.4** – może wynikać z warunków środowiskowych lub ograniczeń czujnika.

## Boxplot: AH

- **Statystyki opisowe:**

- **Medianą:** Wynosi **~0.82** – połowa pomiarów jest niższa od tej wartości.
- **Rozstęp międzykwartylowy (IQR):** Przedział **0.72–0.93** obejmuje 50% danych, co potwierdza koncentrację pomiarów w dolnych zakresach.
- **Wartości odstające:** Punkty powyżej **1.2** – mogą być związane z **Epizodami pogodowymi** (np. deszczowe dni).

- **Implikacje:**

- Niska wilgotność bezwzględna dominuje w danych, co może wynikać z suchych warunków pomiarowych lub sezonowości (np. pomiary zimą).

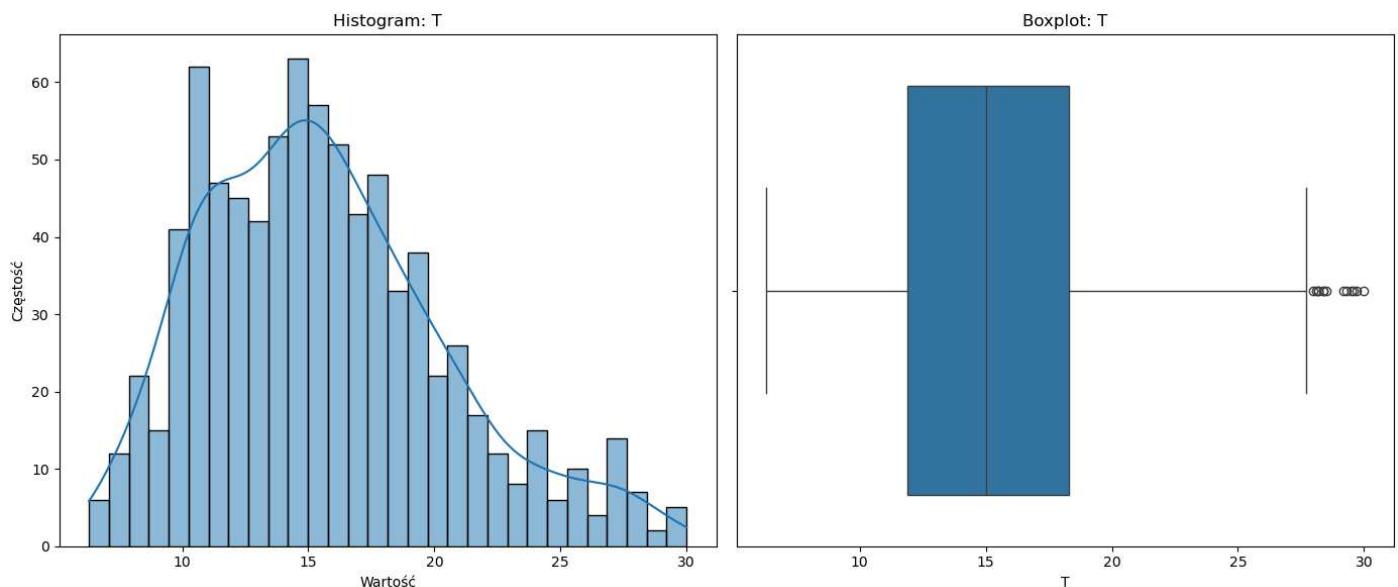
# Zmienna T

```
In [ ]: arr1 = df['T']
```

```
In [ ]: plt.figure(figsize=(14,6))
plt.subplot(1,2,1)
sns.histplot(arr1, kde=True, bins=30)
plt.title("Histogram: T")
plt.xlabel("Wartość")
plt.ylabel("Częstość")

plt.subplot(1,2,2)
sns.boxplot(x=arr1)
plt.title("Boxplot: T")
plt.xlabel("T")

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



## Analiza wykresów dla zmiennej T (temperatura)

### Histogram: T

- **Kształt rozkładu:**
  - **Prawoskośność** – większość danych skupia się w przedziale **0–30**, z maksymalną częstotliwością ok. **55** dla najniższych wartości (ok. **15**).
  - **Ogon rozkładu:** Rozciąga się do **30**, ale wartości powyżej **25** występują rzadko (częstotliwość spada do **10**).
- **Dominujący zakres:**
  - Ponad **80% pomiarów** mieści się w przedziale **7–20**, co sugeruje przewagę **nisko-średnich temperatur** w badanym okresie.

## Boxplot: T

- **Statystyki opisowe:**

- **Medianą:** Wynosi **15** – połowa pomiarów jest niższa od tej wartości.
- **Rozstęp międzykwartylowy (IQR):** Przedział **11.9–18.3** obejmuje 50% danych, co wskazuje na **umiarkowaną zmienność**.
- **Wartości odstające:** Punkty powyżej **27** – mogą być związane z **Epizodami pogodowymi** (np. upalne dni).

- **Implikacje:**

- Dominacja niskich temperatur może wynikać z sezonowości (np. pomiary zimowe) lub lokalnych warunków klimatycznych.

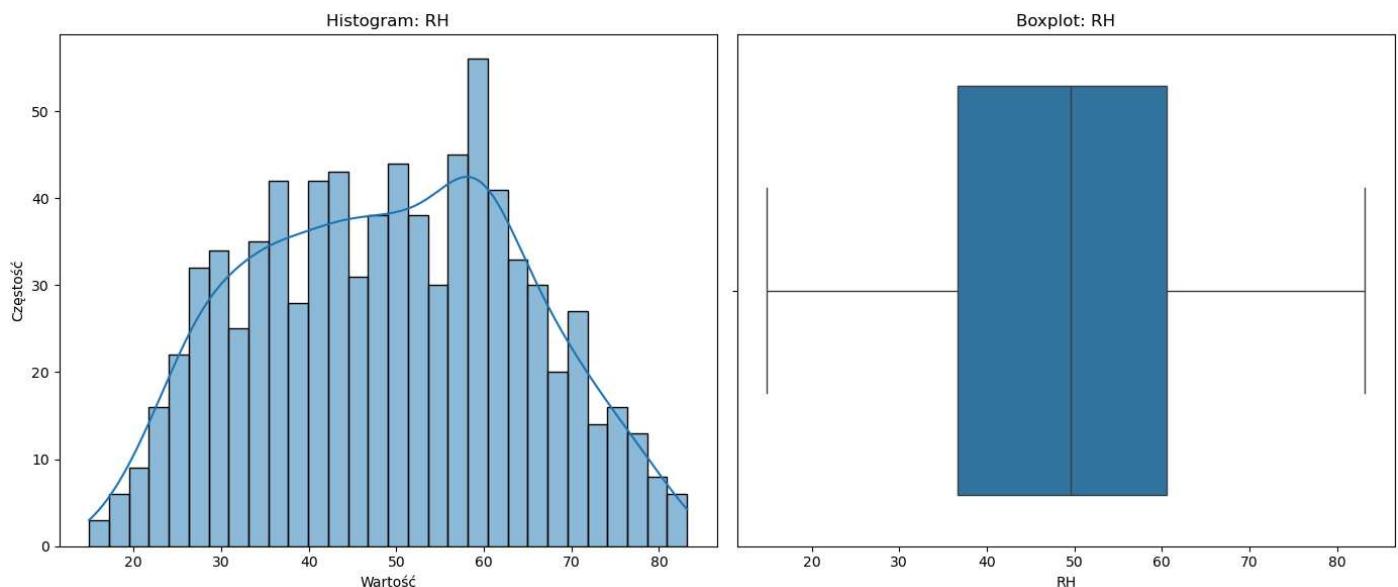
# Zmienna RH

```
In [ ]: arr1 = df['RH']
```

```
In [ ]: plt.figure(figsize=(14,6))
plt.subplot(1,2,1)
sns.histplot(arr1, kde=True, bins=30)
plt.title("Histogram: RH")
plt.xlabel("Wartość")
plt.ylabel("Częstość")

plt.subplot(1,2,2)
sns.boxplot(x=arr1)
plt.title("Boxplot: RH")
plt.xlabel("RH")

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



## Analiza wykresów dla zmiennej RH (wilgotność względna)

### Histogram: RH

- **Kształt rozkładu:**
  - **Rozkład prawdopodobnie równomierny** – dane rozłożone w zakresie **15–83**, bez wyraźnej dominacji jednego przedziału.
  - **Brak skrajnych wartości** – wszystkie pomiary mieszczą się w zakresie **14.9–83.**, co sugeruje stabilne warunki pomiarowe.
- **Dominujący zakres:**
  - Najczęściej występują wartości w okolicy **~60**.

## Boxplot: RH

- **Statystyki opisowe:**

- **Medianą:** Szacowana na ~50 – połowa pomiarów jest niższa od tej wartości.
- **Rozstęp międzykwartylowy (IQR):** Przedział **36.7–60.5** obejmuje 50% danych, co wskazuje na umiarkowaną zmienność.
- **Wartości odstające:** Brak wyraźnych outlierów – dane skupione w głównym zakresie.

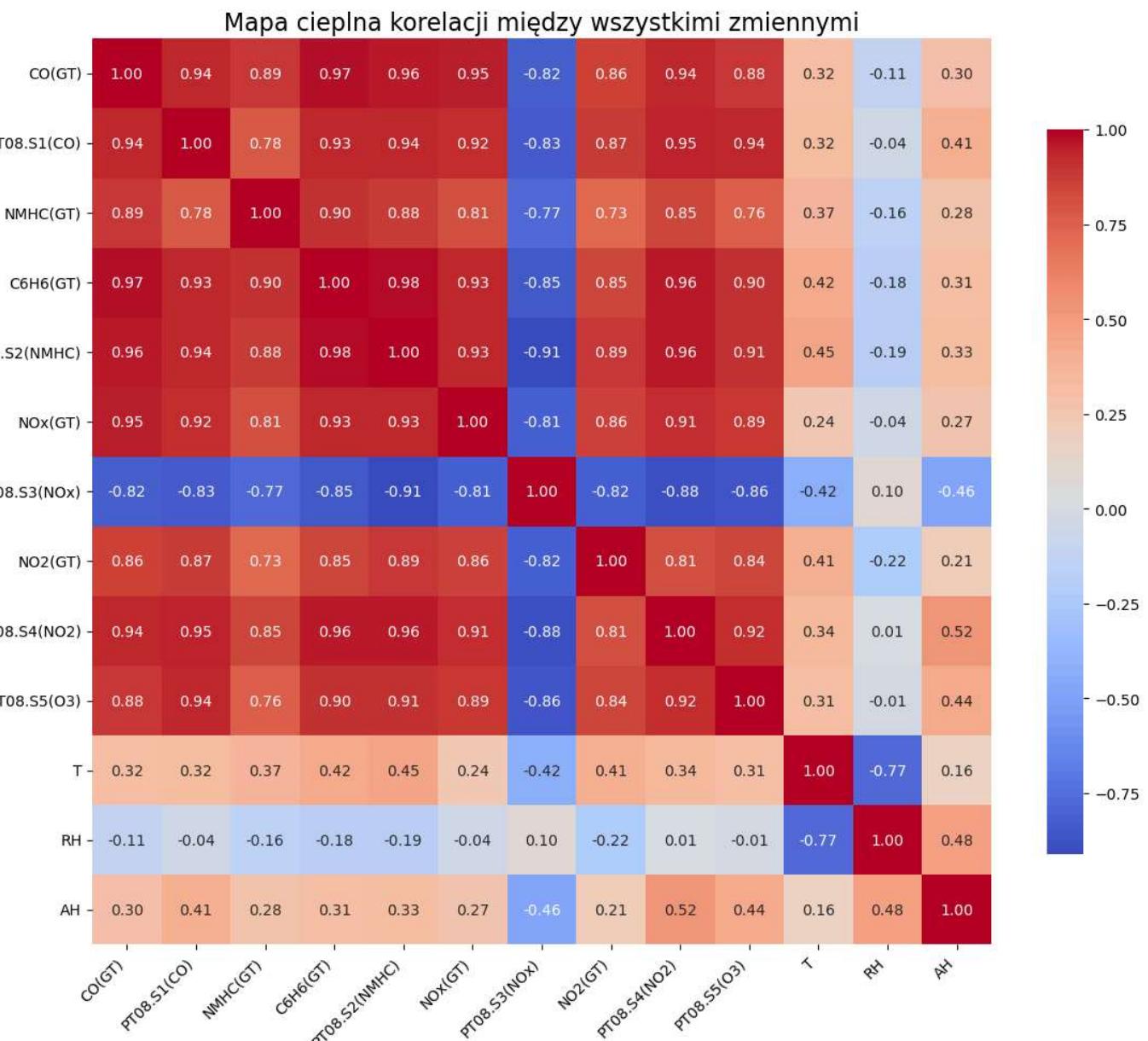
- **Implikacje:**

- Wilgotność względna utrzymuje się w typowym zakresie, co może odpowiadać warunkom mieszanym (np. naprzemienne okresy suche i wilgotne).

# Mapa cieplna (heatmap)

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

```
corr = df.corr()  
  
plt.figure(figsize=(12,10))  
sns.heatmap(corr,  
            annot=True,  
            fmt=".2f",  
            cmap="coolwarm",  
            cbar_kws={"shrink": .8})  
plt.title("Mapa cieplna korelacji między wszystkimi zmiennymi", fontsize=16)  
plt.xticks(rotation=45, ha='right')  
plt.yticks(rotation=0)  
plt.tight_layout()  
plt.show()
```



# Mapa cieplna przedstawiająca współczynniki korelacji Pearsona pomiędzy wszystkimi zmiennymi w zbiorze danych.

## Kluczowe obserwacje:

### 1. Bardzo silne dodatnie korelacje ( $> 0,9$ ) między czujnikami a odpowiadającymi im gazami oraz między samymi czujnikami

- **CO(GT) vs PT08.S1(CO):**  $r \approx 0,94$
- **NMHC(GT) vs PT08.S2(NMHC):**  $r \approx 0,98$
- **NO<sub>2</sub>(GT) vs PT08.S4(NO<sub>2</sub>):**  $r \approx 0,96$
- **O<sub>3</sub> (PT08.S5)** vs inne sensory: wiele korelacji  $r \geq 0,90$
- Czujniki między sobą (np. PT08.S1–PT08.S2, PT08.S2–PT08.S4, itp.) również  $r \approx 0,9–0,96$

### 2. Silna ujemna korelacja czujnika NO<sub>x</sub> (PT08.S3) z większością pozostałych zmiennych

- PT08.S3(NO<sub>x</sub>) vs PT08.S2(NMHC):  $r \approx -0,91$
- PT08.S3 vs CO(GT), C6H6(GT), NO<sub>2</sub>(GT), PT08.S5(O<sub>3</sub>):  $r$  w granicach  $-0,82$  do  $-0,88$
- Wyjątek: **PT08.S3 vs RH** – lekko dodatnia ( $r \approx 0,10$ )

### 3. Temperatura (T)

- Umiarkowanie dodatnio koreluje z większością gazów i sensorów ( $r \approx 0,24–0,45$ ), np. T vs PT08.S2(NMHC)  $r \approx 0,45$ .
- Oznacza to, że wzrost T wiąże się z wyższymi odczytami poszczególnych sensorów (możliwe zmiany w chemii detekcji lub sezonowe źródła emisji).

### 4. Wilgotność względna (RH)

- Silnie ujemnie koreluje z temperaturą:  $r \approx -0,77$  (cieplejsze powietrze  $\rightarrow$  niższa RH).
- Z sensorami i gazami korelacje słabe ( $r \approx -0,04$  do  $-0,22$ ), co sugeruje niewielki bezpośredni wpływ RH na odczyty.

### 5. Wilgotność absolutna (AH)

- Umiarkowanie dodatnio koreluje z temperaturą ( $r \approx 0,16$ ) i czujnikami ( $r \approx 0,27–0,52$ ), np. AH vs PT08.S4(NO<sub>2</sub>)  $r \approx 0,52$ .
- Wskazuje to, że większa ilość pary wodnej w powietrzu nieco zwiększa sygnały sensorów.

---

## Podsumowując:

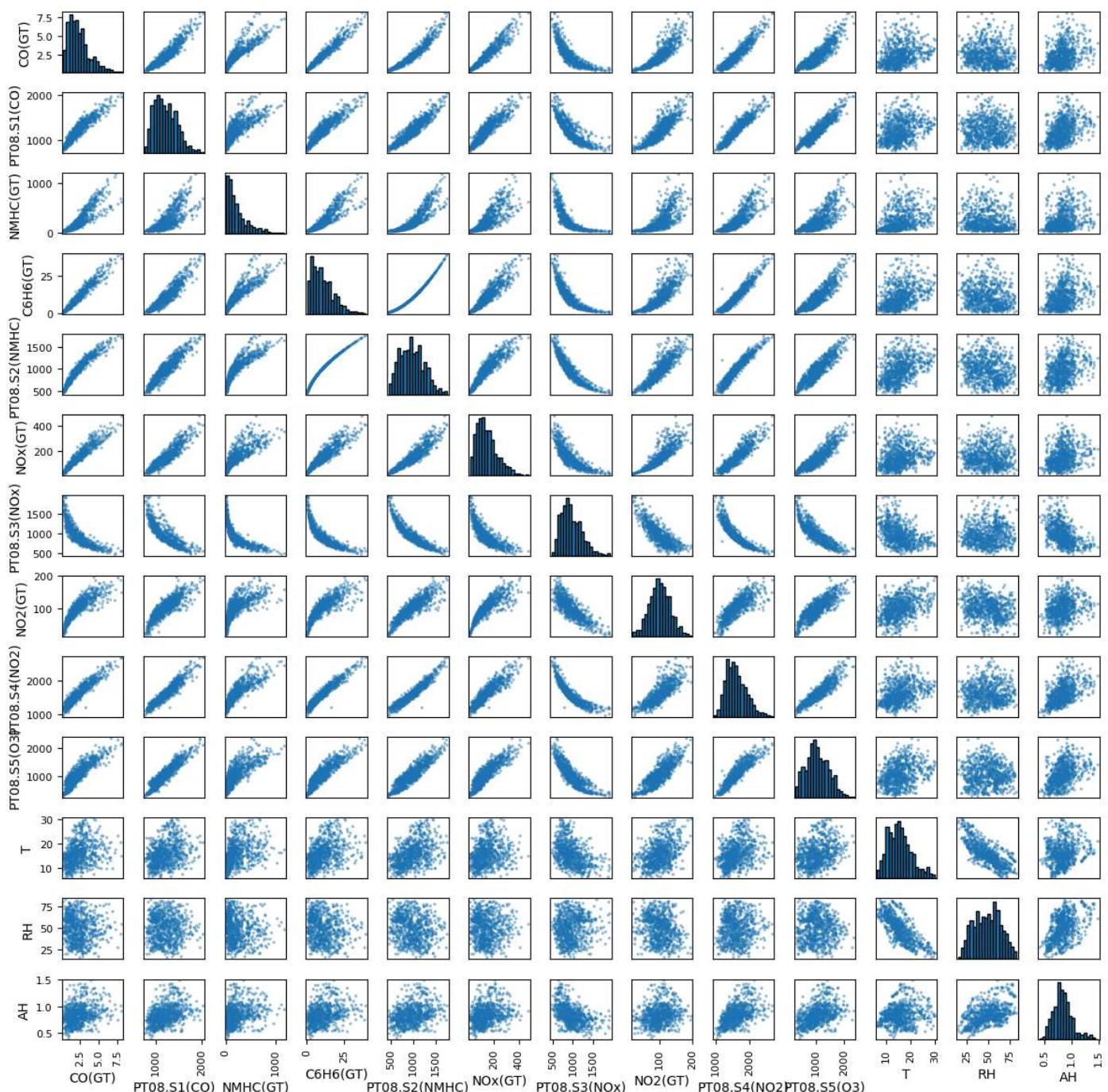
- Dane wykazują niemal idealne sprzężenie czujników z odpowiadającymi im gazami oraz ze sobą nawzajem ( $r \sim 0,9–1,0$ ).
- Czujnik PT08.S3 (NO<sub>x</sub>) zachowuje się odwrotnie – jego odczyty silnie ujemnie korelują z większością innych zmiennych.
- Meteorologia: temperatura sprzyja wyższym odczytom sensorów, wilgotność względna wręcz przeciwnie, a absolutna ma efekt umiarkowanie dodatni.

# Macierz wykresów punktowych (scatter matrix)

In [ ]:

```
from pandas.plotting import scatter_matrix
import matplotlib.pyplot as plt

scatter_matrix(df,
               alpha=0.5,
               diagonal='hist',
               figsize=(12,12),
               s=20,
               hist_kwds={'bins': 20, 'edgecolor':'k'})
plt.suptitle("", y=1.02, fontsize=16)
plt.tight_layout()
plt.show()
```



# Macierz rozrzutu łączy w sobie w jednym wykresie dwie informacje:

## 1. Przekątne (histogramy)

- **Gazowe zmienne** (CO(GT), NMHC(GT), C6H6(GT), NOx(GT), NO2(GT)): wszystkie mają prawoskośne rozkłady – większość pomiarów jest niska, a nieliczne wartości mocno odstające tworzą długą „ogon” w prawo.
- **Sygnały czujników** (PT08.S1...PT08.S5): podobnie – silne prawoskośne rozkłady z wieloma niskimi odczytami i kilkoma bardzo wysokimi.
- **Temperatura (T)**: bardziej symetryczna, skupiona ok. 0–30 °C.
- **Wilgotność względna (RH)**: rozkład w miarę równomierny między ~20 % a ~80 %.
- **Wilgotność absolutna (AH)**: wartości od ~0,5 do ~1,5 g/m<sup>3</sup>, również prawoskośnie rozłożone.

## 2. Zależności między czujnikami i gazami

### 1. Bardzo silne, niemal liniowe relacje

- **CO(GT) vs PT08.S1(CO)**: wąski, prostoliniowy obłok.
- **NMHC(GT) vs PT08.S2(NMHC), NO<sub>2</sub>(GT) vs PT08.S4(NO<sub>2</sub>), NOx(GT) vs PT08.S3(NO<sub>x</sub>)** – wszystkie pokazują bardzo wyraźne, wąskie „kłęby” punktów liniowo rosnące.

### 2. Wysokie korelacje między samymi czujnikami

- Punkty PT08.S1 vs PT08.S2, PT08.S2 vs PT08.S4 itd. układają się w niemal prostą linię.

## 3. Wpływ temperatury

- **Czujniki vs T** (np. PT08.S3(NO<sub>x</sub>) vs T, PT08.S2 vs T): widzimy charakterystyczny spadek sygnału wraz ze wzrostem temperatury – kształt hiperboliczny/exponential decay. Oznacza to, że przy wyższej T odczyty czujników (zwłaszcza PT08.S3) maleją szybciej, potem stabilizują się.

## 4. Wilgotność

- **RH vs większość zmiennych**: punkty są mocno rozproszone, bez wyraźnego trendu, co potwierdza niskie korelacje.
- **AH vs czujniki i gazy**: umiarkowany trend rosnący – przy wyższej wilgotności absolutnej sygnały czujników nieznacznie rosną (obszary punktów lekko pochylone w górę).

## 5. Ogólne wnioski

- **Czujniki chemiczne** Większość czujników ściśle reaguje na stężenia konkretnych zanieczyszczeń, co widać po bardzo wąskich, liniowych „obłokach” punktów (wysokie korelacje).
- **Temperatura** ma silny, ale nieliniowy wpływ na odczyty sensorów (spadek wartości wraz z ociepleniem).
- **Wilgotność względna** ma niewielki, praktycznie losowy związek z odczytami, podczas gdy **wilgotność absolutna** wykazuje umiarkowany dodatni trend.

# Wykres czasowy danych pomiarowych

```
In [10]: df1= df1.drop(columns=["Unnamed: 15", "Unnamed: 16"])
df1.replace(-200.0, np.nan, inplace=True)
df1= df1[:-114]
```

```
In [11]: import pandas as pd
import matplotlib.pyplot as plt

df1['Time_clean'] = df1['Time'].str.replace('.', ':', regex=False)

df1['Datetime'] = pd.to_datetime(
    df1['Date'] + ' ' + df1['Time_clean'],
    format='%d/%m/%Y %H:%M:%S'
)
df1 = df1.set_index('Datetime')

df1 = df1.drop(columns=['Date', 'Time', 'Time_clean'])
```

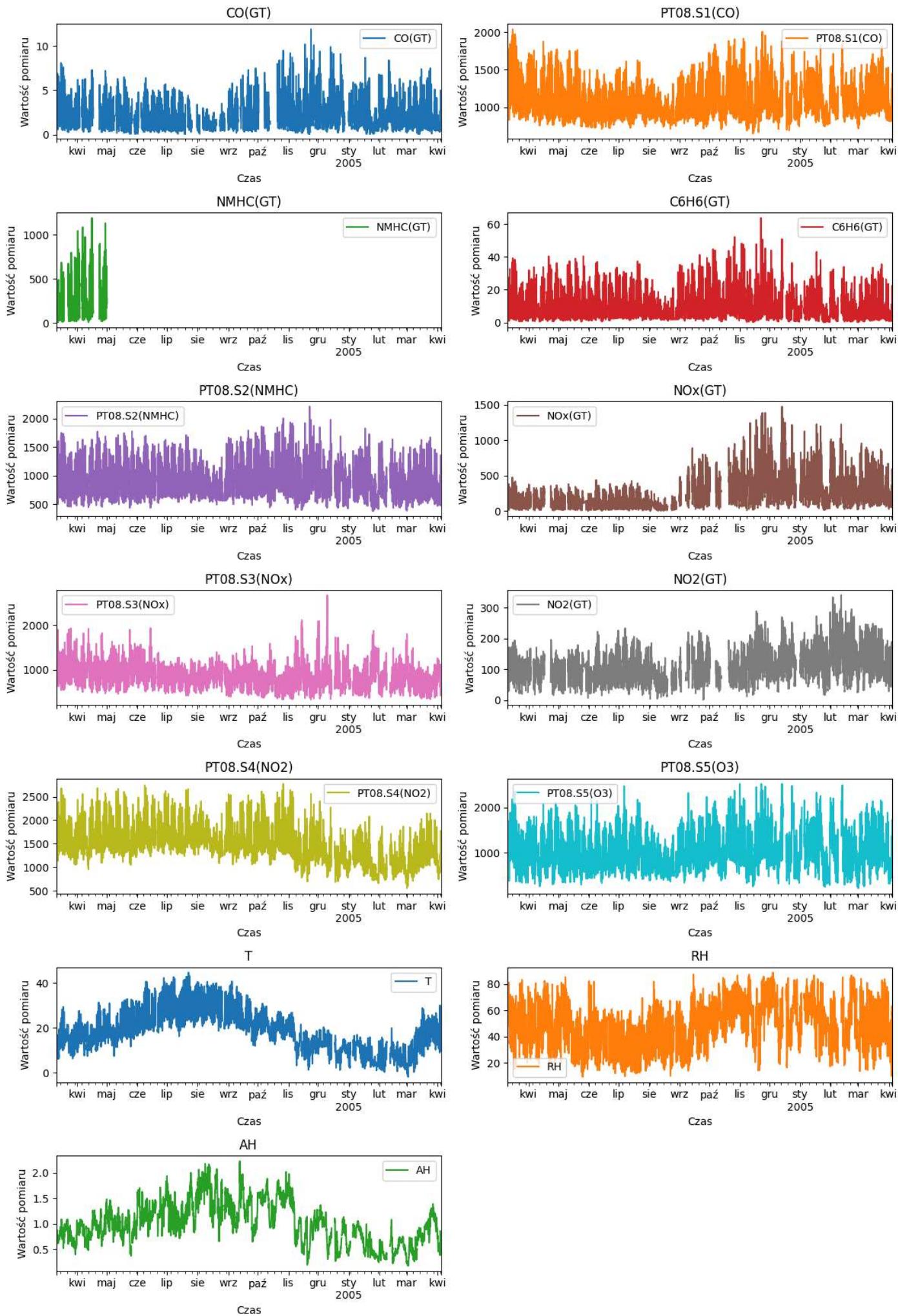
```
In [13]: import matplotlib.pyplot as plt
cols = df1.columns.tolist()
n = len(cols)
ncols = 2
nrows = (n + ncols - 1) // ncols

axes = df1[cols].plot(
    subplots=True,
    layout=(nrows, ncols),
    figsize=(12, nrows * 2.5),
    sharex=False
)

for ax, col in zip(axes.flatten(), cols):
    ax.set_xlabel("Czas")
    ax.set_ylabel("Wartość pomiaru")
    ax.set_title(col)

if nrows * ncols > n:
    for ax in axes.flatten()[n:]:
        fig = ax.get_figure()
        fig.delaxes(ax)

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



# Analiza każdej zmiennej z wykresu:

## 1. CO(GT)

- **Jednostka:** mg/m<sup>3</sup>
- **Trend:** wiosną (kwiecień–maj) niskie stężenia (~1–4), latem lekkie podwyższenie (~2–6), a od jesieni do zimy (październik–luty) wyraźny wzrost do 6–12 mg/m<sup>3</sup>. Wiosną następuje spadek z powrotem.
- **Sezonowość:** wyższe stężenia CO zimą (sprzyja inwersja, więcej spalania).

## 2. PT08.S1(CO)

- **Sygnał czujnika CO (jednostki ADC)**
- **Trend:** wzrost od wiosny (~1000–1500) do jesieni/zimy (~1250–2000), potem spadek wiosną kolejnego roku.
- **Zależność od T:** przy wyższej temperaturze (lato) odczyty są niższe, a przy chłodzie (zima) rosną.

## 3. NMHC(GT)

- **Węglowodory niezamierzone (µg/m<sup>3</sup>)**
- **Braki danych:** widoczny tylko „czysty” okres wiosenny, ponieważ nasze dane dla tej zmiennej były zbierane tylko do maja.

## 4. C6H6(GT)

- **Benzen (µg/m<sup>3</sup>)**
- **Trend:** względnie stabilne 5–25 wiosną/latem, skoki do 40–60 jesienią/zimą, spadek wiosną.

## 5. PT08.S2(NMHC)

- **Sygnał czujnika NMHC**
- **Trend:** podobny kształt jak NMHC(GT): najniższe wartości latem (~700–1200), wyższe zimą (~1000–2000).

## 6. NOx(GT)

- **Tlenki azotu (ppb)**
- **Trend:** do września umiarkowane ~0–400, następnie gwałtowny wzrost do 400–1400 w październiku–lutym, potem powrót do niższych poziomów.

## 7. PT08.S3(NOx)

- **Sygnał czujnika NO<sub>x</sub>**
- **Trend odwrotny do NOx(GT):** latem wysoki poziom sygnału (ok. 1500–2800), zimą spada (ok. 1200–2000). Wynika to z ujemnej korelacji sensora z temperaturą.

## 8. NO2(GT)

- **Dwutlenek azotu (µg/m<sup>3</sup>)**
- **Trend:** łagodny wzrost od wiosny (~0–100) do zimy (~100–300), z widocznymi skokami w miesiącach chłodnych.

## 9. PT08.S4(NO<sub>2</sub>)

- **Sygnał czujnika NO<sub>2</sub>**
- **Trend:** latem wyższe sygnały (~1500–2600), zimą nieco niższe (~1000–2000) – odwrotnie niż mierzona wartość NO<sub>2</sub>.

## 10. PT08.S5(O<sub>3</sub>)

- **Sygnał czujnika O<sub>3</sub>**
- **Trend:** względnie stały, ale z lekkim obniżeniem zimą (przy niższym nasłonecznieniu) i podwyższeniem latem (~1000–2500).

## 11. T (Temperatura, °C)

- **Trend sinusoidalny:** od ~5 °C w kwietniu do ~35 °C w lipcu–sierpniu, spadek do ~0 °C w styczniu i ponowny wzrost wiosną.

## 12. RH (Wilgotność względna, %)

- **Trend odwrotny do T:** wyższa RH zimą (~40–80 %), niższa latem (~20–60 %) – chłodne powietrze utrzymuje więcej wilgoci względnie.

## 13. AH (Wilgotność absolutna, g/m<sup>3</sup>)

- **Trend:** rośnie latem (~1–2,2 g/m<sup>3</sup>), spada zimą (~0,5–1,2 g/m<sup>3</sup>), co odzwierciedla fizyczny wzrost pojemności pary przy wyższej T.

---

### Kluczowe wnioski:

- **Sezonowość** odczytów (gazy i czujniki) silnie związana z temperaturą: zimą stężenia zanieczyszczeń (CO, NOx, benzen) rosną, latem spadają.
- **Czujniki PT08.S1–PT08.S5** reagują odwrotnie lub zgodnie z mierzonymi gazami (np. PT08.S3 odwrotnie do NOx).
- **Wilgotność absolutna (AH)** rośnie/ maleje zgodnie z temperaturą, natomiast **wilgotność względna (RH)** wykazuje przeciwną do T tendencję.
- **Zauważalne duże skoki** stężeń w okresie jesienno-zimowym (możliwe źródła komunikacyjne + ogrzewanie).

# Inżynieria danych

Ta część projektu zawiera w sobie głębszą analizę oraz inżynierię zbioru danych. Dotyczy ona następujących etapów przetwarzania danych:

- Sprawdzanie poprawności typów danych
- Usuwanie zduplikowanych wierszy
- Podział na zbiory treningowy / walidacyjny / testowy (unkanie data leakage)
- Praca z wartościami pustymi
- Praca z wartościami odstającymi
- Selekcja cech
- Eksport oczyszczonych danych

## Wczytanie danych

In [ ]:

```
import pandas as pd
import numpy as np

df = pd.read_csv("data/AirQualityUCI.csv", sep=";", decimal=",")
```

	Date	Time	CO(GT)	PT08.S1(CO)	NMHC(GT)	C6H6(GT)	PT08.S2(NMHC)	NOx(GT)	PT08.
0	10/03/2004	18.00.00	2.6	1360.0	150.0	11.9		1046.0	166.0
1	10/03/2004	19.00.00	2.0	1292.0	112.0	9.4		955.0	103.0
2	10/03/2004	20.00.00	2.2	1402.0	88.0	9.0		939.0	131.0
3	10/03/2004	21.00.00	2.2	1376.0	80.0	9.2		948.0	172.0
4	10/03/2004	22.00.00	1.6	1272.0	51.0	6.5		836.0	131.0
...	...	...	...	...	...	...	...	...	...
9466	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
9467	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
9468	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
9469	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
9470	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

9471 rows × 17 columns

# Usuwanie niepotrzebnych danych

Na podstawie wniosków z pliku `initial_analysis.ipynb` usuwam zbędne wiersze i kolumny (o kolumnach `Time` i `Date` wspomniano w rozdziale *Selekcja cech*). Wartość -200 została przekonwertowana na NaN, zgodnie z opisem danych.

```
In [ ]: df = df[:-114]
df = df.drop(columns=["Time", "Date", "Unnamed: 15", "Unnamed: 16"])
df.replace(-200.0, np.nan, inplace=True)
print("Data shape:", df.shape)
```

Data shape: (9357, 13)

Wstępnie dane mają 13 kolumn i 9357 wierszy

## Sprawdzanie typów danych

```
In [ ]: df.info()
```

#	Column	Non-Null Count	Dtype	
0	CO(GT)	7674	non-null	float64
1	PT08.S1(CO)	8991	non-null	float64
2	NMHC(GT)	914	non-null	float64
3	C6H6(GT)	8991	non-null	float64
4	PT08.S2(NMHC)	8991	non-null	float64
5	NOx(GT)	7718	non-null	float64
6	PT08.S3(NOx)	8991	non-null	float64
7	NO2(GT)	7715	non-null	float64
8	PT08.S4(NO2)	8991	non-null	float64
9	PT08.S5(O3)	8991	non-null	float64
10	T	8991	non-null	float64
11	RH	8991	non-null	float64
12	AH	8991	non-null	float64

dtypes: float64(13)  
memory usage: 950.4 KB

Po wczytaniu danych, wszystkie kolumny mają typ danych float. W repozytorium z oryginalnymi danymi jest podana informacja o typach danych dla każdej kolumny i są zdefiniowane 3 takie typy: Integer (int), Categorical (object) i Continuous (float). Jednak typ Categorical dla kolumn `PT08.S1(CO)`, `PT08.S2(NMHC)`, `PT08.S3(NOx)`, `PT08.S4(NO2)` i `PT08.S5(O3)` jest błędny ponieważ wartości w tych kolumnach są odpowiedziami sensoru a więc powinny być enkodowane jako liczby. Można by było przekonwertować odpowiednie kolumny z float64 do int, ale ponieważ zbiór jest mały, nie ma w tym dużego sensu, dlatego zostawiam float64 dla wszystkich kolumn, bo nie wpływa to na dalszą analizę.

## Usuwanie zduplikowanych wierszy

Nie ma sensu od dwóch jednakowych wierszy, bo nie wnosi to nowej informacji do modelu, więc sprawdzam czy są takie w zbiorze i usuwam

```
In [ ]: print("Liczba zduplikowanych wierszy:", df.duplicated().sum())
df.drop_duplicates(inplace=True)
```

Liczba zduplikowanych wierszy: 31

Są 31 zduplikowanych, wierszy

## Usuwanie wartości brakujących

```
In [ ]: df.isna().sum()
```

CO(GT)	1653
PT08.S1(CO)	335
NMHC(GT)	8412
C6H6(GT)	335
PT08.S2(NMHC)	335
NOx(GT)	1608
PT08.S3(NOx)	335
NO2(GT)	1611
PT08.S4(NO2)	335
PT08.S5(O3)	335
T	335
RH	335
AH	335

dtype: int64

Pierwsze co jest zauważalne, to duża liczba wartości brakujących w kolumnie NMHC(GT) (8412 / 9326).

Ponieważ wartości pustych jest kilka razy więcej niż rzeczywistych, nie ma sensu coś robić z tą kolumną bo będzie ona sztucznie utworzona. Dlatego usuwam ją

```
In [ ]: df = df.drop("NMHC(GT)", axis=1)
```

Kilka kolumn (PT08.S1(CO), C6H6(GT), PT08.S2(NMHC) itp.) mają taką samą liczbę wartości pustych (335), sprawdzam czy to nie są te same wiersze jak w przypadku ostatnich 144 w zbiorze początkowym.

```
In [ ]: df[df["PT08.S1(CO)"].isna()].drop(["CO(GT)", "NOx(GT)", "NO2(GT)"], axis = 1)
```

	PT08.S1(CO)	C6H6(GT)	PT08.S2(NMHC)	PT08.S3(NOx)	PT08.S4(NO2)	PT08.S5(O3)	T	RH	A
524	NaN	NaN		NaN	NaN	NaN	NaN	NaN	NaN
525	NaN	NaN		NaN	NaN	NaN	NaN	NaN	NaN
526	NaN	NaN		NaN	NaN	NaN	NaN	NaN	NaN
701	NaN	NaN		NaN	NaN	NaN	NaN	NaN	NaN
702	NaN	NaN		NaN	NaN	NaN	NaN	NaN	NaN
...	...	...	...	...	...	...	...	...	...
8111	NaN	NaN		NaN	NaN	NaN	NaN	NaN	NaN
8112	NaN	NaN		NaN	NaN	NaN	NaN	NaN	NaN
8113	NaN	NaN		NaN	NaN	NaN	NaN	NaN	NaN
8114	NaN	NaN		NaN	NaN	NaN	NaN	NaN	NaN
8777	NaN	NaN		NaN	NaN	NaN	NaN	NaN	NaN

335 rows × 9 columns



Faktycznie, tak jest. Usuwam te wiersze, bo wszystkie wartości w nich są puste, a ich liczba jest mała w porównaniu do całego zbioru

```
In [ ]: df = df.drop(df[df["PT08.S1(CO)"].isna()].drop(["CO(GT)", "NOx(GT)", "NO2(GT)"], axis = 1).index)
          .reset_index(drop=True)
```

Sytuacja z kolumnami CO(GT), NOx(GT), NO2(GT) jest niejednoznaczna. Z jednej strony, liczba wierszy jest zauważa aby usuwać ich ze zbioru (około 20%). Z drugiej, nie możemy usunąć 3 kolumn ze zbioru, ponieważ mają w sobie wartości, które będą wykorzystane do modelowania. Dlatego, odpowiednim sposobem będzie raczej imputacja. Istnieje kilka metod imputacji od prostych, do bardziej złożonych:

- Imputacja wartością stałą
- Imputacja średnią/medianą/percentylami
- Interpolacja
- Imputacja za pomocą metod ML (KNN, Random Forest, Decision Trees)

Ponieważ wypełniania stałą wartością (nie ma różnicę czy średnią czy zerem) zniekształca rozkład zmiennej, robiąc imputowaną wartość dominującą, wybieram KNN imputację, czyli na podstawie podobnych obserwacji ze zbioru.

# Data Leakage

Data Leakage (wyciek danych) – jeden z najczęstszych problemów w projektach ML/DS. Występuje wtedy, gdy informacje ze zbioru testowego zostaną nieświadomie wykorzystane podczas trenowania modelu, np. poprzez wcześniejsze przetwarzanie danych (skalowanie, imputację braków, itp.) na całym zbiorze danych przed podziałem na zbiór treningowy i testowy. To prowadzi do zawyżonych wyników na etapie walidacji i sprawia, że model gorzej generalizuje do nowych, rzeczywistych danych.

## Train / Val / Test split

Aby uniknąć wycieku danych podzielę zbiór danych do jakichkolwiek przekształceń. W celu uzyskanie bardziej obiektywnej oceny w ewaluacji modelu, tworzę 3 zbiory zamiast 2 standardowych (drugi najczęstszy problem w projektach), ponieważ zbudowanie dobrego modelu uczenia maszynowego to pętla, która ma w sobie kilkadziesiąt / kilkaset iteracji. Pod czas tych iteracji, coś zmieniamy w modelu lub danych (np. metodę przetwarzania lub zmianę hiperparametrów modelu) i w ten sposób możemy przeuczyć model na obserwacjach ze zbioru testowego, a z tego wynika zła generalizacja i zła ocena modelu. Dlatego dzielę zbiór na trzy podzbiory:

1. Training set (do trenowania)(80%) - zbiór wykorzystywany do uczenia modelu (wyznaczania parametrów, pozyskiwania patternów)
2. Validation set (do ewaluacji)(10%) - zbiór wykorzystywany do wyboru najlepszego modelu (tuningu hiperparametrów, porównania modeli)
3. Test set (do końcowej ewaluacji)(10%) - zbiór, który jest odkładany do ostatniego etapu ewaluacji wybranym najlepszym modelem, żadnych zmian w danych/modelu po ewaluacji nie wykonuje się

```
In [ ]: from sklearn.model_selection import train_test_split

X = df.drop("C6H6(GT)", axis=1)
y = df["C6H6(GT)"]

X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.2, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)

X_train.shape, X_val.shape, X_test.shape

((7192, 11), (899, 11), (900, 11))
```

Po podzieleniu mamy 7192 obserwacje w zbiorze treningowym i po 900 w zbiorach walidacyjnym i testowym

# KNN Imputacja

```
In [ ]: from sklearn.impute import KNNImputer  
imputer = KNNImputer()
```

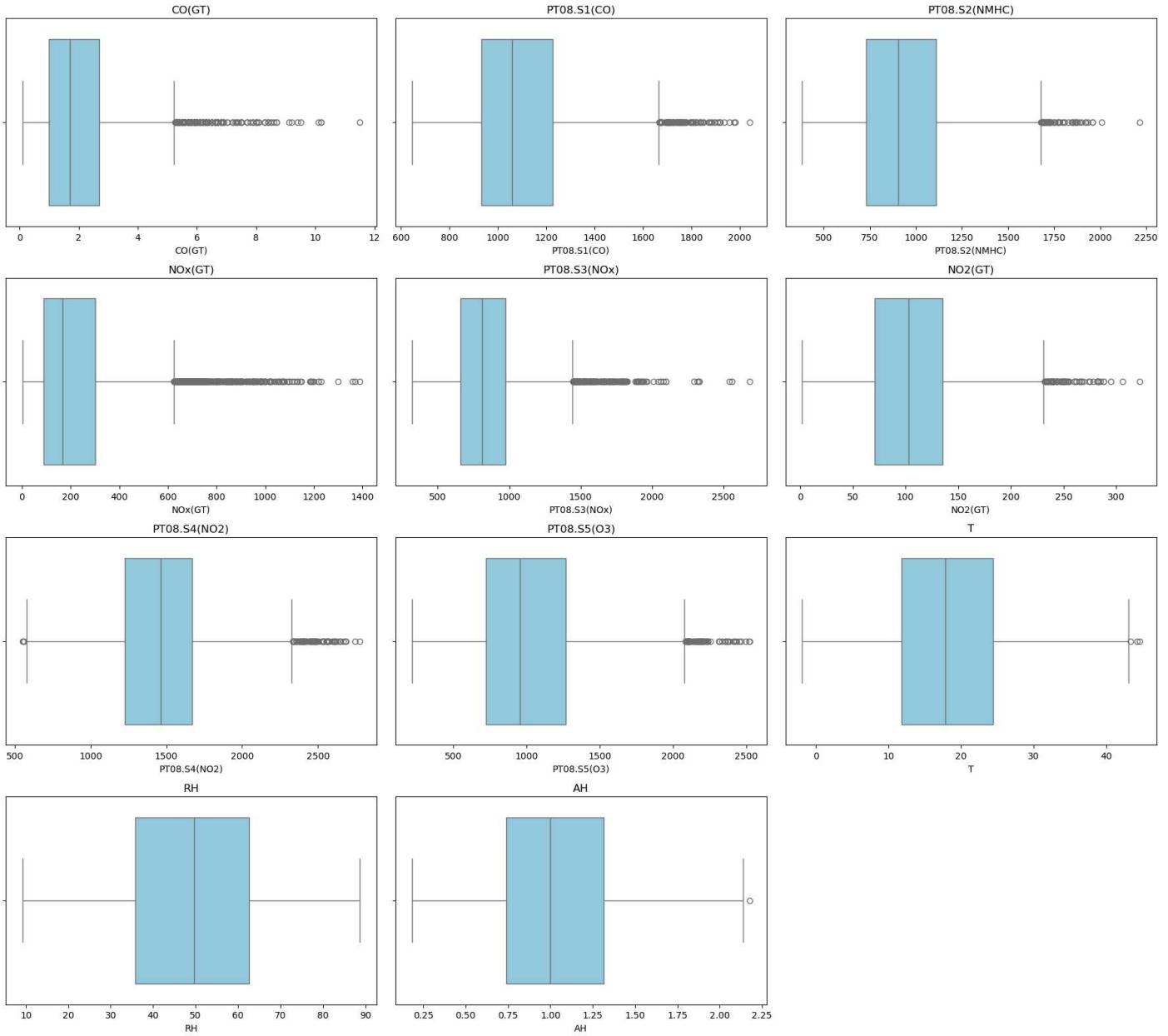
```
X_train_imputed = pd.DataFrame(imputer.fit_transform(X_train), columns=X_train.columns, index=X_train.index)  
X_val_imputed = pd.DataFrame(imputer.transform(X_val), columns=X_val.columns, index=X_val.index)  
X_test_imputed = pd.DataFrame(imputer.transform(X_test), columns=X_test.columns, index=X_test.index)
```

```
In [ ]: X_train_imputed.isna().sum().sum(), X_val_imputed.isna().sum().sum(), X_test_imputed.isna().sum().sum()  
(0, 0, 0)
```

## Wartości odstające

Identyfikuję wartości odstające za pomocą wykresów pudełkowych

```
In [ ]: import math  
import matplotlib.pyplot as plt  
import seaborn as sns  
  
cols = X_train_imputed.columns  
n = len(cols)  
cols_per_row = 3  
rows = math.ceil(n / cols_per_row)  
  
fig, axes = plt.subplots(rows, cols_per_row, figsize=(18, rows * 4))  
axes = axes.flatten()  
  
for i, col in enumerate(cols):  
    sns.boxplot(x=X_train_imputed[col], ax=axes[i], color="skyblue")  
    axes[i].set_title(col)  
  
for j in range(i + 1, len(axes)):  
    axes[j].set_visible(False)  
  
plt.tight_layout()  
plt.show()
```



Boxploty pokazują, że są wartości odstające we wszystkich kolumnach oprócz RH, ale nie widać dokładnie ile ich jest

## Metoda IQR

Do wyznaczania wartości odstających korzystam z metody IQR (interquartile range) czyli rozstępu międzykwartylowego. Metoda ta nie jest zależna od rozkładu cechy (jak np. z-score method), a więc bez sprawdzania histogramów obliczam różnicę między trzecim a pierwszym kwartylami i obliczam outlier'y poza dolną i górną granicami. Dolna i górna granice są zdefiniowane jako  $Q1 - 1.5 * IQR$  i  $Q3 + 1.5 * IQR$  odpowiednio.

```
In [ ]: numeric_cols = X_train_imputed.select_dtypes(include=[np.number])

Q1 = numeric_cols.quantile(0.25)
Q3 = numeric_cols.quantile(0.75)
IQR = Q3 - Q1

is_outlier = (numeric_cols < (Q1 - 1.5 * IQR)) | (numeric_cols > (Q3 + 1.5 * IQR))

outliers_per_column = is_outlier.sum()
print("Liczba outlierów w zbiorze treningowym:\n\n", outliers_per_column)

total_outliers = is_outlier.sum().sum()
rows_w_outliers = len(X_train_imputed[is_outlier.any(axis=1)])
```

```
print(f"\nŁącznie {total_outliers} wartości odstających w {rows_w_outliers} wierszach.")
```

Liczba outlierów w zbiorze treningowym:

```
CO(GT)           248  
PT08.S1(CO)     100  
PT08.S2(NMHC)   58  
NOx(GT)          399  
PT08.S3(NOx)    195  
NO2(GT)          75  
PT08.S4(NO2)    83  
PT08.S5(O3)     69  
T                3  
RH               0  
AH               1  
dtype: int64
```

Łącznie 1231 wartości odstających w 735 wierszach.

Sytuacja z wartościami odstającymi jest różna dla każdej kolumny i niejednoznaczna dla całego zbioru. Mianowicie, w kolumnach CO(GT), NOx(GT) oraz PT08.S3(NOx) jest duża liczba outlierów, natomiast w kolumnach T, RH, AH - mała, w pozostałych - średnio. Łącznie wartości odstające są w 735 wierszach zbioru treningowego, czyli około 10% całego.

Metoda obróbki wartości odstających w większości zależy od charakteru danych i celów projektu, najbardziej popularnymi metodami są:

- Nie robić żadnych przekształceń
- Usuwanie wierszów z outlierami
- Obcinanie wartości do dolnej / górnej granicy

Ponieważ końcowym celem jest zbudowanie modelu regresyjnego, a są modele odporne na wartości odstające (XGBoost, DecisionTree, LightGBM itp.) to w celach tego projektu nie będę ani usuwać, ani obcinać wartości odstających. Outliery to nie zawsze jest coś złego i na pewno nie to same co wartości brakujące. Nie mamy dostępnej informacji czy są to błędy czujników, czy realne wartości. Ewentualnie sprawdzam czy wpływają one na model

```
In [ ]: X_train_clipped = X_train_imputed.copy()  
  
for col in numeric_cols.columns:  
    Q1 = X_train_imputed[col].quantile(0.25)  
    Q3 = X_train_imputed[col].quantile(0.75)  
    IQR = Q3 - Q1  
    lower_bound = Q1 - 1.5 * IQR  
    upper_bound = Q3 + 1.5 * IQR  
  
    X_train_clipped[col] = np.clip(X_train_imputed[col], lower_bound, upper_bound)  
  
outliers = ~is_outlier.any(axis=1)  
X_train_removed_outliers = X_train_imputed[outliers]  
y_train_removed_outliers = y_train[outliers]
```

```
In [ ]: def build_and_evaluate(X_train, y_train, X_val, y_val):  
  
    from xgboost import XGBRegressor  
    from sklearn.linear_model import LinearRegression  
    from sklearn.metrics import mean_squared_error  
  
    model = XGBRegressor()  
    model.fit(X_train, y_train)
```

```

y_pred = model.predict(X_val)
rmse_xgb = np.sqrt(mean_squared_error(y_val, y_pred))

model = LinearRegression()
model.fit(X_train, y_train)
y_pred = model.predict(X_val)
rmse_lr = np.sqrt(mean_squared_error(y_val, y_pred))

return rmse_xgb, rmse_lr

rmse_imputed, rmse_imputed_lr = build_and_evaluate(X_train_imputed, y_train, X_val_imputed, y_val)
rmse_clipped, rmse_clipped_lr = build_and_evaluate(X_train_clipped, y_train, X_val_imputed, y_val)
rmse_removed_outliers, rmse_removed_outliers_lr = build_and_evaluate(X_train_removed_outliers, y_train, X_val_imputed, y_val)

print("XGBoost RMSE (odporny na outliers):")
print(f"RMSE for data with outliers: {rmse_imputed:.2f}")
print(f"RMSE for clipped outliers: {rmse_clipped:.2f}")
print(f"RMSE for removed outliers: {rmse_removed_outliers:.2f}")

print("\nLinear Regression RMSE (wrażliwy na outliers):")
print(f"RMSE for data with outliers: {rmse_imputed_lr:.2f}")
print(f"RMSE for clipped outliers: {rmse_clipped_lr:.2f}")
print(f"RMSE for removed outliers: {rmse_removed_outliers_lr:.2f}")

```

XGBoost RMSE (odporny na outliers):

RMSE for data with outliers: 0.35

RMSE for clipped outliers: 0.45

RMSE for removed outliers: 1.36

Linear Regression RMSE (wrażliwy na outliers):

RMSE for data with outliers: 1.18

RMSE for clipped outliers: 1.17

RMSE for removed outliers: 1.30

Zarówno dla modelu odpornego (XGBoost) na wartości odstające i wrażliwego (Linear Regression) RMSE w przypadku danych z wartościami odstającymi jest niski w porównaniu do innych metod. To potwierdza wybraną wcześniej metodę obróbki wartości odstających.

```
In [ ]: X_train = X_train_imputed.copy()
X_val = X_val_imputed.copy()
X_test = X_test_imputed.copy()
```

## Selekcja Cech

Im więcej cech tym złożoniejszy jest model. Zaleca się trenować modele najbardziej proste, aby zminimalizować koszt trenowania / predykcji, można było łatwiej interpretować cechy ważność cech (Feature Importance) np. w XAI i tp. W tym celu, przeanalizuję kolumny zbioru aby wybrać najbardziej odpowiednie.

## Kolumny nie potrzebne do analizy

Oryginalny zbiór zawiera kolumny `Date` i `Time`, ale nie potrzebujemy ich w celu modelowania, więc usunąłem ich wraz po wczytaniu danych.

## Kolumny o zerowej wariancji

Kolumny o zerowej wariancji - kolumny w zbiorze, w których wszystkie wartości są identyczne. Jeżeli wartości są jednakowe dla całej kolumny to nie ta kolumna nie wnoszi nowej informacji przy trenowaniu

modeli. Sprawdzam czy są takie w zbiorze

```
In [ ]: zero_var_cols = [col for col in X_train.columns if X_train[col].nunique() == 1]
print("Zero variance columns:", zero_var_cols)
```

Zero variance columns: []

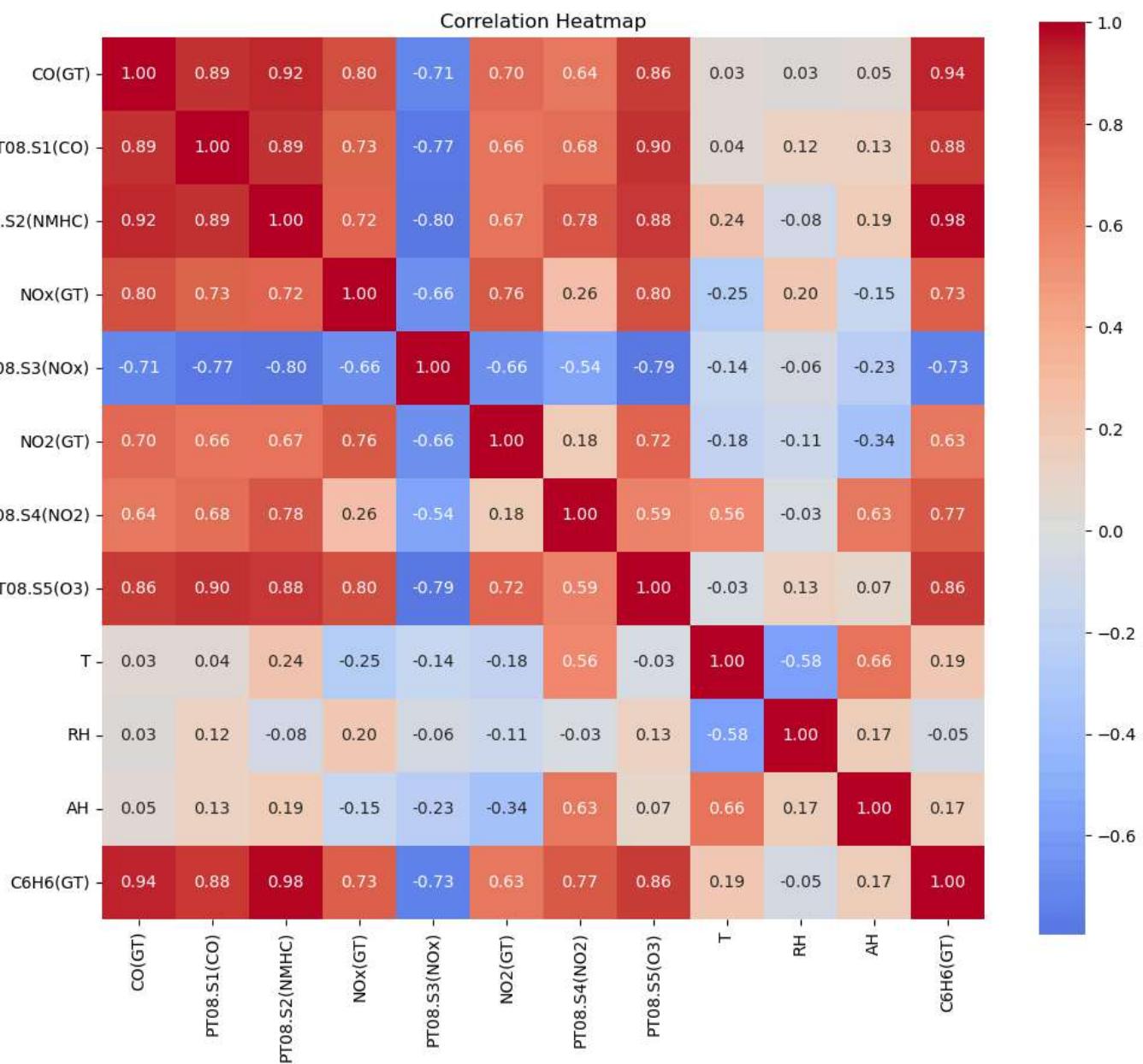
Nie ma kolumn o zerowej wariancji

## Redundancy

Redundancja, czyli nadmiarowość, wynika w przypadku, gdy predyktory są mocno skorelowane między sobą, zawierając podobną информацию. Choć takie cechy nie są błędnymi, jednak mają negatywny wpływ na złożoność modelu, interpretację wyników, ryzyko overfittingu. Dlatego, na podstawie macierzy korelacji pomiędzy cechami oraz korelacji cech i targetu wybieram te, który mają niską korelację pomiędzy sobą a najwyższą możliwą ze zmienną objaśnianą.

```
In [ ]: df_corr = pd.concat([X_train, y_train], axis=1)
corr = df_corr.corr()

plt.figure(figsize=(12, 10))
sns.heatmap(corr, annot=True, fmt=".2f", cmap="coolwarm", square=True, center = 0)
plt.title("Correlation Heatmap")
plt.show()
```



W tym zbiorze, jak w podobnych, które zawierają dane o jakości powietrza jest problem redundancji, ponieważ większość cech jest silnie skorelowana pomiędzy sobą. I można w takim przypadku zrobić na 4 sposoby:

1. Spróbować wybrać cechy ręcznie
2. Wykonać PCA, czyli redukcję wymiarowości
3. Wykorzystać gotową metodę do selekcji cech (np. SelectKBest)
4. Zostawić jak jest

Ponieważ końcowym celem jest zbudowanie jaknajlepszego modelu predykcyjnego (czyli z najmniejszym błędem), podobnie jak z wartościami odstającymi, testuję wszystkie opcje, trenując najprostszy model.

C6H6(GT) jest zmienną objasianą, więc skupiam się na korelacji predyktorów z nią. PT08.S2(NMHC) ma największy wsp. korelacji z nią więc wybieram tą kolumnę jako zmienną objaśniającą i usuwam te cechy, które silnie ( $>0.7$ ) są skorelowane

```
In [ ]: columns_to_drop_manual = ["CO(GT)", "PT08.S1(CO)", "NOx(GT)", "PT08.S3(NOx)", 'PT08.S4(NO2)', 'P  
X_train_manual = X_train.drop(columns=columns_to_drop_manual)  
X_val_manual = X_val.drop(columns=columns_to_drop_manual)  
  
print("Liczba kolumn po usunięciu manualnym:", X_train_manual.shape[1])  
Liczba kolumn po usunięciu manualnym: 5
```

```
In [ ]: from sklearn.decomposition import PCA  
  
pca = PCA(n_components=0.95)  
X_train_pca = pca.fit_transform(X_train)  
X_val_pca = pca.transform(X_val)  
  
print("Liczba kolumn po PCA:", X_train_pca.shape[1])  
Liczba kolumn po PCA: 3
```

Wadą PCA jest utrata interpretowalności cech (bo są to kombinacje liniowe oryginalnych), co może utrudniać wyjaśnianie wyników modelu.

```
In [ ]: from sklearn.feature_selection import SelectKBest, f_regression  
  
selector = SelectKBest(score_func=f_regression, k=5)  
X_train_kbest = selector.fit_transform(X_train, y_train)  
X_val_kbest = selector.transform(X_val)  
print("Liczba kolumn po SelectKBest:", X_train_kbest.shape[1])  
Liczba kolumn po SelectKBest: 5
```

```
In [ ]: rmse_orig, rmse_orig_lr = build_and_evaluate(X_train, y_train, X_val, y_val)  
rmse_manual, rmse_manual_lr = build_and_evaluate(X_train_manual, y_train, X_val_manual, y_val)  
rmse_pca, rmse_pca_lr = build_and_evaluate(X_train_pca, y_train, X_val_pca, y_val)  
rmse_kbest, rmse_kbest_lr = build_and_evaluate(X_train_kbest, y_train, X_val_kbest, y_val)  
  
print("XGBoost RMSE (oryginalne dane):", rmse_orig)  
print("XGBoost RMSE (usunięte ręcznie):", rmse_manual)  
print("XGBoost RMSE (PCA):", rmse_pca)  
print("XGBoost RMSE (SelectKBest):", rmse_kbest)  
  
print("\nLinear Regression RMSE (oryginalne dane):", rmse_orig_lr)  
print("Linear Regression RMSE (usunięte ręcznie):", rmse_manual_lr)  
print("Linear Regression RMSE (PCA):", rmse_pca_lr)  
print("Linear Regression RMSE (SelectKBest):", rmse_kbest_lr)
```

```
XGBoost RMSE (oryginalne dane): 0.3460492680167805
XGBoost RMSE (usunięte ręcznie): 0.5507019851134495
XGBoost RMSE (PCA): 1.9760168198703294
XGBoost RMSE (SelectKBest): 0.5104304904139009
```

```
Linear Regression RMSE (oryginalne dane): 1.1758562616031005
Linear Regression RMSE (usunięte ręcznie): 1.3627617601667226
Linear Regression RMSE (PCA): 2.3743754204362375
Linear Regression RMSE (SelectKBest): 1.322532397359445
```

Sytuacja jest podobna do outlierów. Zarówno odporny na redundancję XGBoost jak i wrażliwy Linear Regeression model pokazują najmniejszy błąd dla danych bez usuwania kolumn. Po PCA jest najgorzys rezultat, oprócz faktu, że tracimy interpretację cech. Ponieważ przyszłe modele nie są skoplikowane, wielkość datasetu nie jest duża a trenowanie modelu zajmuje mniej niż sekundę, to nie będę usuwać żadnych kolumn. Choć i so skorelowane między sobą i jedna wyjaśnia drugą, ale razem wnoszą dodatkową informację, która jest ważna do predykacji.

```
In [ ]: train_cleaned = pd.concat([X_train, y_train], axis=1)
train_cleaned.to_csv("data/train_cleaned.csv", index=False)

val_cleaned = pd.concat([X_val, y_val], axis=1)
val_cleaned.to_csv("data/val_cleaned.csv", index=False)

test_cleaned = pd.concat([X_test, y_test], axis=1)
test_cleaned.to_csv("data/test_cleaned.csv", index=False)
```

# Klastrowanie

W tej części przeprowadzone będzie rozwiązywanie problemu predykcyjnego nienadzorowanego za pomocą algorytmów klastrowania. Ideą jest sprawdzić czy możliwy jest podział danych na klasy (np. dobra / średnia / zła jakość powierza).

In [ ]:

```
import pandas as pd

train = pd.read_csv('data/train_cleaned.csv')
val = pd.read_csv('data/val_cleaned.csv')
test = pd.read_csv('data/test_cleaned.csv')

df = pd.concat([train, val, test], axis=0)
df.head(10)
```

	CO(GT)	PT08.S1(CO)	PT08.S2(NMHC)	NOx(GT)	PT08.S3(NOx)	NO2(GT)	PT08.S4(NO2)	PT08.S5(O3)
0	0.64	807.0	658.0	33.0	1113.0	45.0	1328.0	428.0
1	2.50	998.0	1119.0	207.0	832.0	120.0	1780.0	1057.0
2	1.10	968.0	669.0	67.0	1261.0	81.0	1246.0	538.0
3	0.98	925.0	749.0	49.4	826.0	48.4	1482.0	739.0
4	1.90	1100.0	1022.0	492.0	752.0	193.0	1101.0	1367.0
5	2.80	1171.0	1064.0	160.0	881.0	110.0	1639.0	1097.0
6	1.40	1097.0	721.0	183.0	746.0	96.0	1266.0	807.0
7	0.70	844.0	538.0	158.0	1164.0	109.0	726.0	479.0
8	4.00	1305.0	1188.0	403.6	546.0	124.8	1843.0	1524.0
9	1.60	1235.0	828.0	118.0	1055.0	83.0	1527.0	1093.0

## Standaryzacja

Przed wykonaniem klastrowania obowiązkowa jest standaryzacja cech, ponieważ zmienne mają różne skale. Wykonuje się ona w celu uniknięcia dominowania cech o większych wartościach liczbowych nad cechami z mniejszymi.

In [ ]:

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
df = pd.DataFrame(scaler.fit_transform(df), columns=df.columns)
```

## K-means clustering

Ponieważ klastrowanie nie jest głównym celem projektu a raczej dodatkowym, używamy tylko jednego algorytmu do klastrowania - K-means. Metoda ta działa dane na K (z góry ustalone) klastrów tak, by punkty w każdym klastrze były najbardziej do siebie podobne (zwykle wg. odległości euklidesowej).

Istnieje kilka metod do wyznaczania liczby K, ponieważ jest to podstawowym założeniem tego algorytmu. Najlepszy wybór oczywiście opiera się znajomości charakteru danych i końcowego celu. My chcemy

zaklasyfikować dane do trzech grup w celu oceniania jakości powietrza:

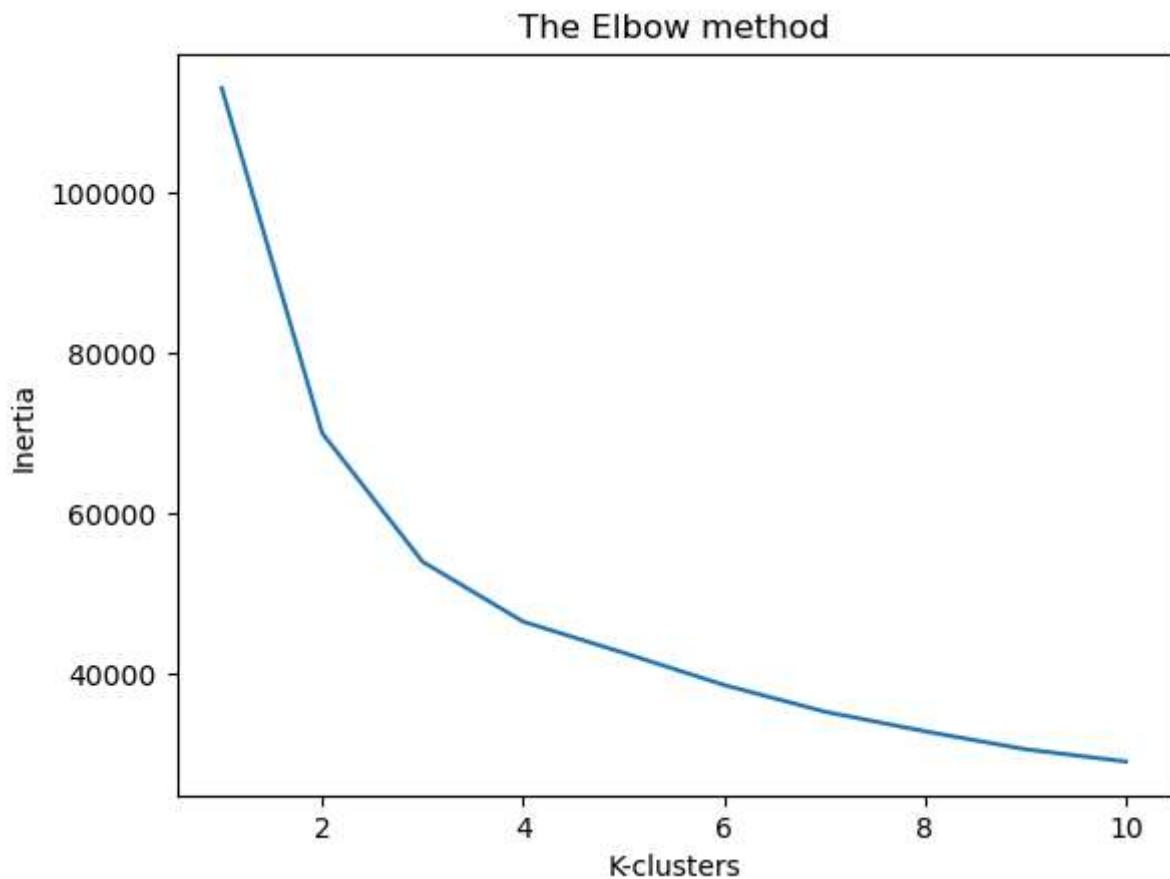
- Czyste / dobra jakość powietrza
- Trochę zabrudnione lub toksyczne / średnia jakość powietrza
- Zabrudnione lub toksyczne / zła jakość powietrza

W przypadku, gdy nie możemy ustalić liczbę klastrów sami, można skorzystać z gotowych metod, jednak są one bardziej subiektywne niż obiektywne. Jedną z takich metod jest metod łokcia lub Elbow Method

```
In [ ]: from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

sse = []
for i in range(1,11):
    km = KMeans(n_clusters = i, random_state=42)
    km.fit(df)
    sse.append(km.inertia_)

plt.plot(range(1,11), sse)
plt.xlabel("K-clusters")
plt.ylabel("Inertia")
plt.title("The Elbow method")
plt.show()
```



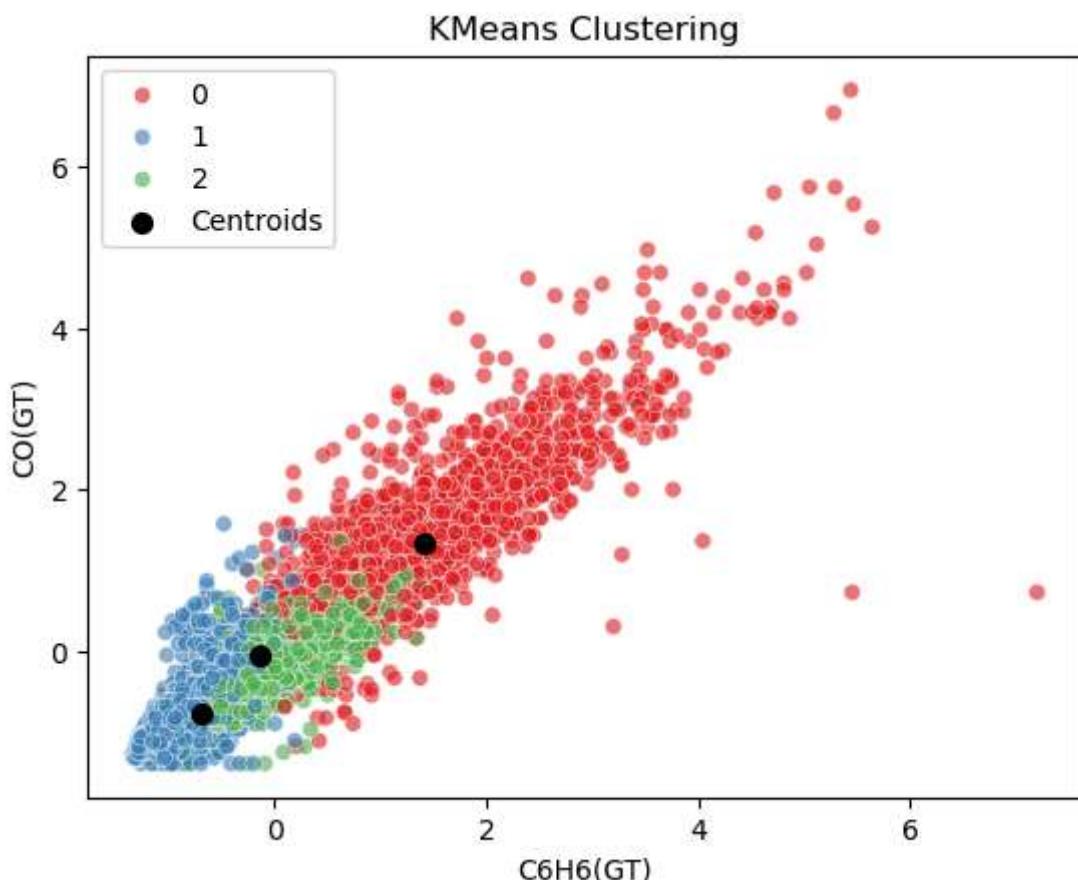
W naszym przypadku właśnie jest sytuacja kiedy tego zgięcia nie widać dobrze. Na pewno odpowiednia liczba wg. tej metody jest w przedziale od 2 do 5, choć ktoś mógłby i wybrać 6. Jednak, wybieramy 3 klasy, bo taki jest końcowy cel do tej części projektu.

```
In [ ]: kmeans = KMeans(n_clusters=3, random_state=42)
labels = kmeans.fit_predict(df)
df['Cluster'] = labels
```

# Wyniki klastrowania

```
In [ ]: import seaborn as sns

sns.scatterplot(data = df, x = "C6H6(GT)", y = "CO(GT)", hue = "Cluster", palette = "Set1", alpha=0.5)
plt.title('KMeans Clustering')
plt.xlabel('C6H6(GT)')
plt.ylabel('CO(GT)')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s=50, c='black', label='Centroids')
plt.legend()
plt.show()
```



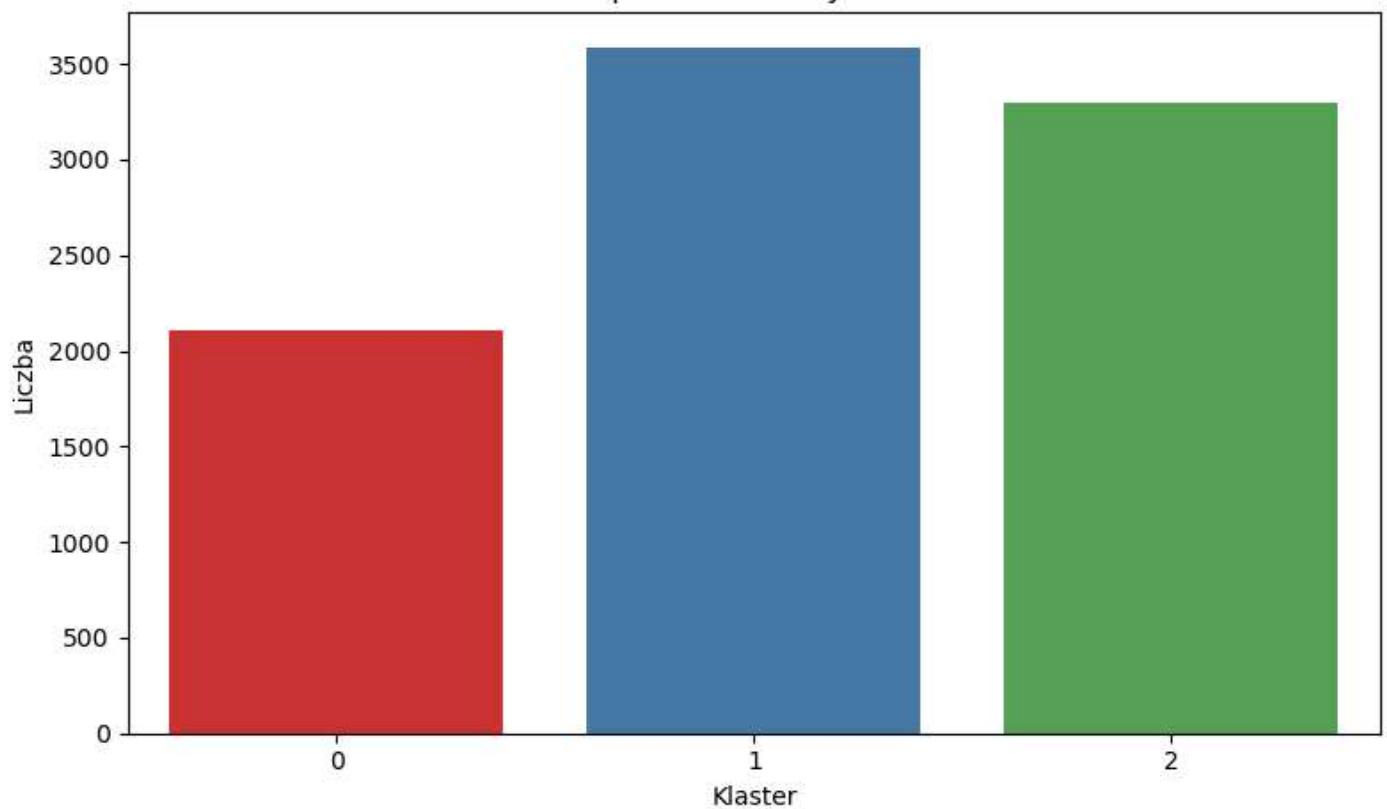
Na powyższym wykresie przedstawiono zależność CO(GT) od C6H6(GT), czyli stężenia tlenku węgla od stężenia benenu. Im wyższa koncentracja tych elemntów w powietrzu tym gorsza jest jakość tego powietrza. Mimo tego, benzen podwyższa toksyczność. Widać, że udało się podzielić dane na 3 klasy, gdy:

- Klasa 0 (czerwony kolor) - zła jakość powietrza (wartości CO i C6H6 są duże)
- Klasa 1 (niebieski kolor) - dobra jakość powietrza (wartości CO i C6H6 są małe)
- Klasa 2 (zielony kolor) - środkowa klasa, czyli niedoskonale czyste powietrze (średnie wartości)

```
In [ ]: cluster_counts = df["Cluster"].value_counts().sort_index()
cluster_df = cluster_counts.reset_index()
cluster_df.columns = ['Cluster', 'Count']

plt.figure(figsize=(8, 5))
sns.barplot(data=cluster_df, x='Cluster', y='Count', hue = "Cluster", palette="Set1", legend=False)
plt.title('Liczba próbek w każdym klastrze')
plt.xlabel('Klauster')
plt.ylabel('Liczba')
plt.tight_layout()
plt.show()
```

Liczba próbek w każdym klastrze



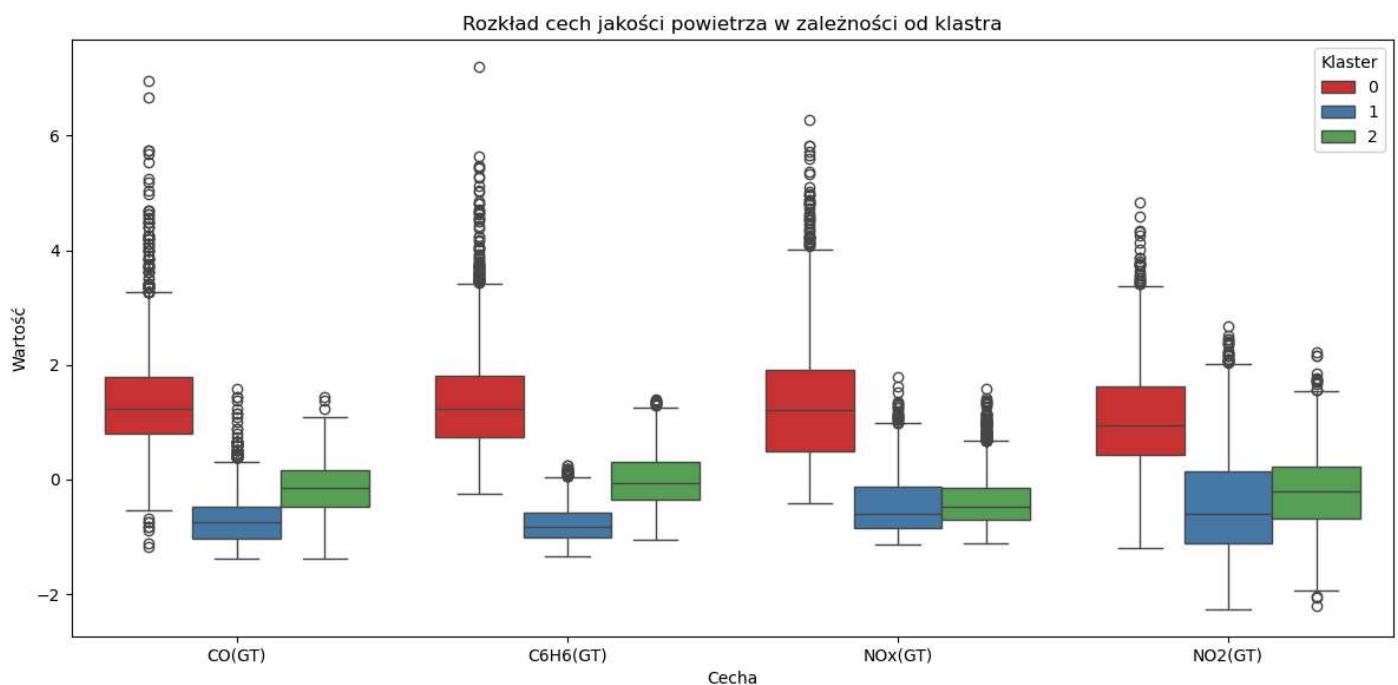
Według liczby elementów Klasa 1 jest największa (ponad 3500 elementów), zatem idzie klasa 2 (około 3200) i klasa 0 (2100 elementów).

```
In [ ]: import seaborn as sns
```

```
features = ['CO(GT)', 'C6H6(GT)', 'NOx(GT)', 'NO2(GT)']

df_melted = df.melt(id_vars='Cluster', value_vars=features,
                     var_name='Cecha', value_name='Wartość')

plt.figure(figsize=(12, 6))
sns.boxplot(data=df_melted, x='Cecha', y='Wartość', hue='Cluster', palette='Set1')
plt.title('Rozkład cech jakości powietrza w zależności od klastra')
plt.legend(title='Klaster')
plt.tight_layout()
plt.show()
```



Rozkład cech, za pomocą których można ocenić jakość powietrza (im większe wartości tym gorsza jakość) pokazuje, że dla wszystkich zmiennych klaster 0, czyli zła jakość ma największe wartości. Zakres wartości i mediana jest dużo wyżej w porównaniu do 1 i 2 klastru. Dobra i średnia jakość natomiast znajdują się blisko siebie, a w niektórych przypadkach (NOx i NO2) nawet są bardzo podobne. Wykres ten potwierdza, że dało się podzielić dane na 3 określone grupy, rozkład analizowanych cech w podziale o klasy to pokazuje.

# Modelowanie

```
In [ ]: def train_and_evaluate_model(model, model_name):
    if model_name == "LightGBM":
        model = model.train(params, lgb_train, valid_sets=[lgb_train, lgb_val])
    elif model_name == "CatBoost":
        model.fit(X_train_scaled, y_train,
                   eval_set=(X_val_scaled, y_val),
                   use_best_model=True,
                   logging_level='Silent')
    else:
        model.fit(X_train_scaled, y_train)

    y_train_pred = model.predict(X_train_scaled)
    y_val_pred = model.predict(X_val_scaled)

    def compute_metrics(y_true, y_pred):
        rmse = np.sqrt(mean_squared_error(y_true, y_pred))
        mae = mean_absolute_error(y_true, y_pred)
        r2 = r2_score(y_true, y_pred)
        return rmse, mae, r2

    rmse_train, mae_train, r2_train = compute_metrics(y_train, y_train_pred)
    rmse_val, mae_val, r2_val = compute_metrics(y_val, y_val_pred)

    plt.figure(figsize=(8, 6))
    plt.scatter(y_val, y_val_pred, alpha=0.5)
    plt.plot([y_val.min(), y_val.max()], [y_val.min(), y_val.max()], 'r--')
    plt.xlabel('Rzeczywiste wartości')
    plt.ylabel('Przewidywane wartości')
    plt.title(f'{model_name} - Predicted vs. True Values')
    plt.grid(True)
    plt.show()

    print(f"--- {model_name} ---")
    print(f"TRAIN → RMSE: {rmse_train:.4f} | MAE: {mae_train:.4f} | R2: {r2_train:.4f}")
    print(f"VALID → RMSE: {rmse_val:.4f} | MAE: {mae_val:.4f} | R2: {r2_val:.4f}")

    return {
        "rmse_train": rmse_train,
        "mae_train": mae_train,
        "r2_train": r2_train,
        "rmse_val": rmse_val,
        "mae_val": mae_val,
        "r2_val": r2_val
    }
```

```
In [ ]: import pandas as pd
import numpy as np
from xgboost import XGBRegressor
from lightgbm import LGBMRegressor, Dataset as LGBDataset
from catboost import CatBoostRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.metrics import mean_absolute_error
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.metrics import PredictionErrorDisplay
from sklearn.preprocessing import PolynomialFeatures
```

```
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
```

```
In [ ]: train_cleaned = pd.read_csv('data/train_cleaned.csv')
val_cleaned = pd.read_csv('data/val_cleaned.csv')
```

```
In [ ]: X_train = train_cleaned.iloc[:, :-1]
y_train = train_cleaned.iloc[:, -1]

X_val = val_cleaned.iloc[:, :-1]
y_val = val_cleaned.iloc[:, -1]
```

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

scaler = StandardScaler()

X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)
```

## Standaryzacja (Normalizacja zmiennych)

**Standaryzujemy (normalizujemy) zmienne**, ponieważ występują one w **różnych jednostkach i mają bardzo różne zakresy wartości** – bez tego dana o dużej skali mogłaby **zdominować obliczenia kosztem cech o mniejszej skali**.

Dzięki standaryzacji, czyli przekształceniu każdej cechy do rozkładu o **średniej 0 i odchyleniu standardowym 1**, zapewniamy, że algorytmy uczące się (zwłaszcza oparte na odległościach lub wykorzystujące regularyzację) traktują **wszystkie cechy równorzędnie**.

Ponadto wiele metod optymalizacji, np. algorytm gradientu prostego, zbiega szybciej i stabilniej na danych o **ujednoliconej skali**, co przekłada się na **efektywniejsze trenowanie modelu**.

## Metryki używane w ocenie modeli:

### RMSE (Root Mean Squared Error)

Miara błędu, która uwzględnia kwadrat różnicy między wartościami rzeczywistymi a przewidywanymi.

Wzór:

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}$$

#### Objaśnienie:

- $(y_i)$ : wartość rzeczywista dla  $(i)$ -tej obserwacji,
- $(\hat{y}_i)$ : wartość przewidywana przez model dla  $(i)$ -tej obserwacji,
- $(n)$ : liczba obserwacji.

## MAE (Mean Absolute Error)

Średnia wartość bezwzględna różnicy między wartościami rzeczywistymi a przewidywanymi.

Wzór:

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$$

### Objaśnienie:

- $(|y_i - \hat{y}_i|)$ : bezwzględna różnica między wartością rzeczywistą a przewidywaną.

---

## R<sup>2</sup> (R-squared)

Wskaźnik dopasowania modelu; wartość bliska 1 oznacza bardzo dobre dopasowanie.

Wzór:

$$R^2 = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2}$$

### Objaśnienie:

- $(\bar{y})$ : średnia wartość zmiennej docelowej ( $y$ ),
- Licznik: suma kwadratów reszt (różnic między wartościami rzeczywistymi a przewidywanymi),
- Mianownik: całkowita suma kwadratów (różnic między wartościami rzeczywistymi a ich średnią).

---

## Podsumowanie:

- **RMSE** karze większe błędy (bo kwadratuje różnice), co jest przydatne, gdy duże błędy są szczególnie niepożądane.
- **MAE** jest bardziej odporny na wartości odstające, ponieważ używa wartości bezwzględnych.
- **R<sup>2</sup>** interpretuje się jako **procent wyjaśnionej wariancji** przez model. Im bliżej 1, tym lepiej model pasuje do danych.

# Model Random Forest Regressor

**Random Forest Regressor** (Las losowy dla regresji) to **zespołowy algorytm uczenia nadzorowanego**, który buduje wiele drzew decyzyjnych i łączy ich przewidywania, aby uzyskać bardziej stabilne i dokładne wyniki. Jest wykorzystywany do przewidywania **wartości ciągłych**.

## Jak działa?

### 1. Tworzenie wielu drzew decyzyjnych:

- Każde drzewo jest trenowane na **losowej podpróbce danych** (tzw. **bootstrapping** — próbkowanie ze zwracaniem).
- W każdym węźle drzewa wybierany jest **losowy podzbiór cech** do podziału (np. 30% wszystkich cech).

### 2. Predykcja:

- Każde drzewo w lesie generuje własną prognozę.
- **Końcowy wynik** to **średnia arytmetyczna** wszystkich przewidywań poszczególnych drzew.

## Kluczowe elementy:

### • Bootstrapping:

Każde drzewo uczy się na innej losowej próbce danych (mogą wystąpić powtórzenia obserwacji).

### • Losowy wybór cech:

W każdym węźle drzewa algorytm wybiera podzbiór cech, co redukuje korelację między drzewami i zapobiega przetrenowaniu.

### • Aggregacja (bagging):

Wyniki drzew są łączone, aby zmniejszyć wariancję modelu.

## Zalety:

1. **Odporność na przetrenowanie:** Dzięki losowości i agregacji, model jest mniej podatny na overfitting niż pojedyncze drzewo.
2. **Obsługa danych nieliniowych:** Skutecznie modeluje złożone zależności między cechami a zmienną docelową.
3. **Automatyczna selekcja cech:** Pomija nieistotne cechy w procesie losowego wyboru.
4. **Interpretowalność:** Możliwość oceny ważności cech (tzw. **feature importance**).

## Wady:

1. **Wolniejsze działanie:** Budowa wielu drzew zwiększa czas obliczeń.
2. **Mniejsza interpretowalność niż pojedyncze drzewo:** Trudniej prześledzić logikę całego lasu.
3. **Wrażliwość na szum:** Jeśli dane zawierają dużo losowego szumu, model może być mniej dokładny.

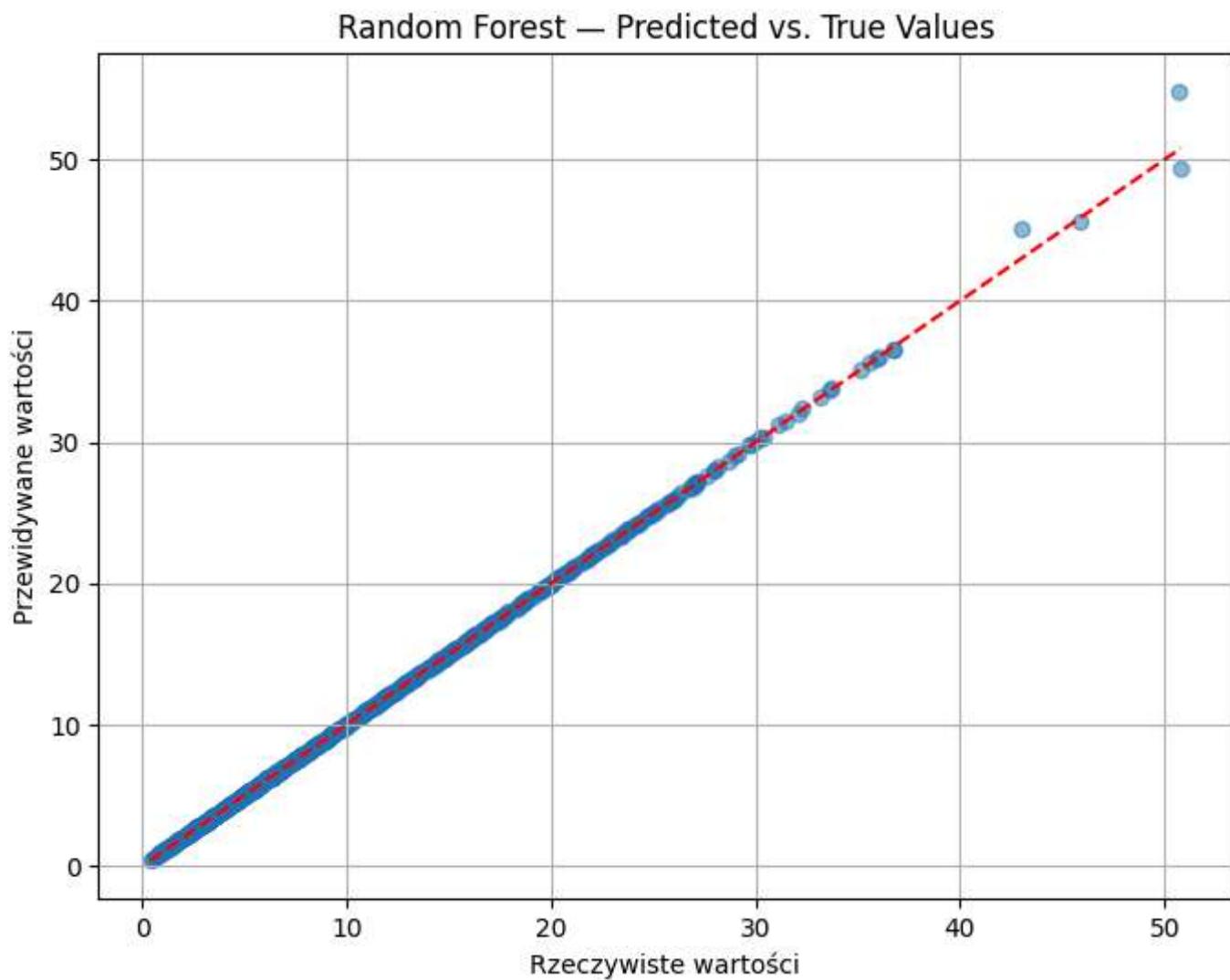
## Hiperparametry do strojenia:

- `n_estimators` : Liczba drzew w lesie (np. 100, 200).
  - `max_depth` : Maksymalna głębokość pojedynczego drzewa.
  - `min_samples_split` : Minimalna liczba próbek wymagana do podziału węzła.
  - `max_features` : Liczba/losowy podzbior cech branych pod uwagę w każdym węźle.
- 

## Podsumowanie:

Random Forest Regressor to **potężne narzędzie do regresji**, które łączy prostotę drzew decyzyjnych z siłą ensemble learningu. Sprawdza się zwłaszcza w problemach z dużą liczbą cech lub złożonymi zależnościami w danych.

```
In [ ]: rf = RandomForestRegressor(random_state=42)
rmse_rf = train_and_evaluate_model(rf, "Random Forest")
```



```
--- Random Forest ---
TRAIN → RMSE: 0.0649 | MAE: 0.0057 | R2: 0.9999
VALID → RMSE: 0.1628 | MAE: 0.0201 | R2: 0.9995
```

## Interpretacja wykresu

- Punkty są bardzo blisko linii  $y = x$ , co wskazuje na **wyjątkowo wysoką dokładność modelu**.
- Niewielkie odchylenia od linii idealnej są widoczne tylko w powiększeniu (ze względu na małe wartości błędów).

## Podsumowanie:

- **Model jest bardzo dokładny:**
  - Zarówno na zbiorze treningowym, jak i walidacyjnym, błędy (RMSE, MAE) są **bliskie zeru**, a  $R^2$  **bliskie 1**.
  - Świadczy to o tym, że model doskonale uchwycił zależności w danych.
- **Brak oznak przetrenowania:**
  - Różnica między wynikami TRAIN i VALID jest minimalna, co sugeruje, że model nie wykazuje oznak **przetrenowania**.

# Model XGBoostRegressor

**XGBRegressor** (eXtreme Gradient Boosting Regressor) to zaawansowana implementacja **gradient boostingu**, zaprojektowana do przewidywania wartości ciągłych. Jest częścią biblioteki **XGBoost**, która słynie z wydajności i skuteczności w konkursach data science (np. Kaggle).

---

## Jak działa?

### 1. Gradient Boosting:

- Algorytm buduje **sekwencję słabych modeli** (zwykle drzew decyzyjnych), gdzie każdy kolejny model koryguje błędy poprzedniego.
- W każdej iteracji obliczane są **reszty** (różnice między przewidywaniami a wartościami rzeczywistymi), a nowe drzewo uczy się je minimalizować.

### 2. Funkcja straty i optymalizacja:

- XGBRegressor minimalizuje **funkcję straty** (np. MSE – błąd średniokwadratowy) za pomocą gradientu (pochodnej funkcji straty).
- Stosuje **optymalizację drugiego rzędu** (uwzględnia również drugą pochodną), co przyspiesza zbieżność.

### 3. Regularyzacja:

- Dodaje kary za zbyt skomplikowane modele (np. za dużą liczbę liści w drzewach), aby zapobiec przetrenowaniu.
  - Parametry `lambda` (L2) i `alpha` (L1) kontrolują siłę regularyzacji.
- 

## Kluczowe cechy XGBRegressor:

### 1. Wydajność:

- Optymalizacja pod kątem szybkości (wykorzystanie równoległych obliczeń, cache'owania danych).

### 2. Obsługa brakujących wartości:

- Automatycznie radzi sobie z brakującymi danymi, wybierając optymalne kierunki podziału w drzewach.

### 3. Elastyczność:

- Może używać różnych funkcji straty i metryk ewaluacji.

### 4. Ważenie próbek:

- Umożliwia przypisanie większej wagi wybranym obserwacjom (np. ważnym outlierom).
- 

## Hiperparametry:

- `learning_rate` ( $\eta$ ): Tempo uczenia (domyślnie 0.3) – im mniejsza wartość, tym więcej drzew potrzeba.
  - `n_estimators`: Liczba drzew w sekwencji (domyślnie 100).
  - `max_depth`: Maksymalna głębokość pojedynczego drzewa (kontroluje złożoność modelu).
  - `subsample`: Proporcja próbek używanych do trenowania każdego drzewa (zapobiega overfittingowi).
  - `colsample_bytree`: Proporcja cech używanych do budowy każdego drzewa.
-

## Zalety:

1. **Wysoka dokładność:** Radzi sobie z złożonymi zależnościami i dużymi zbiorami danych.
  2. **Regularizacja:** Wbudowane mechanizmy przeciwko przetrenowaniu.
  3. **Interpretowalność:** Możliwość oceny ważności cech ( `.feature_importances_` ).
- 

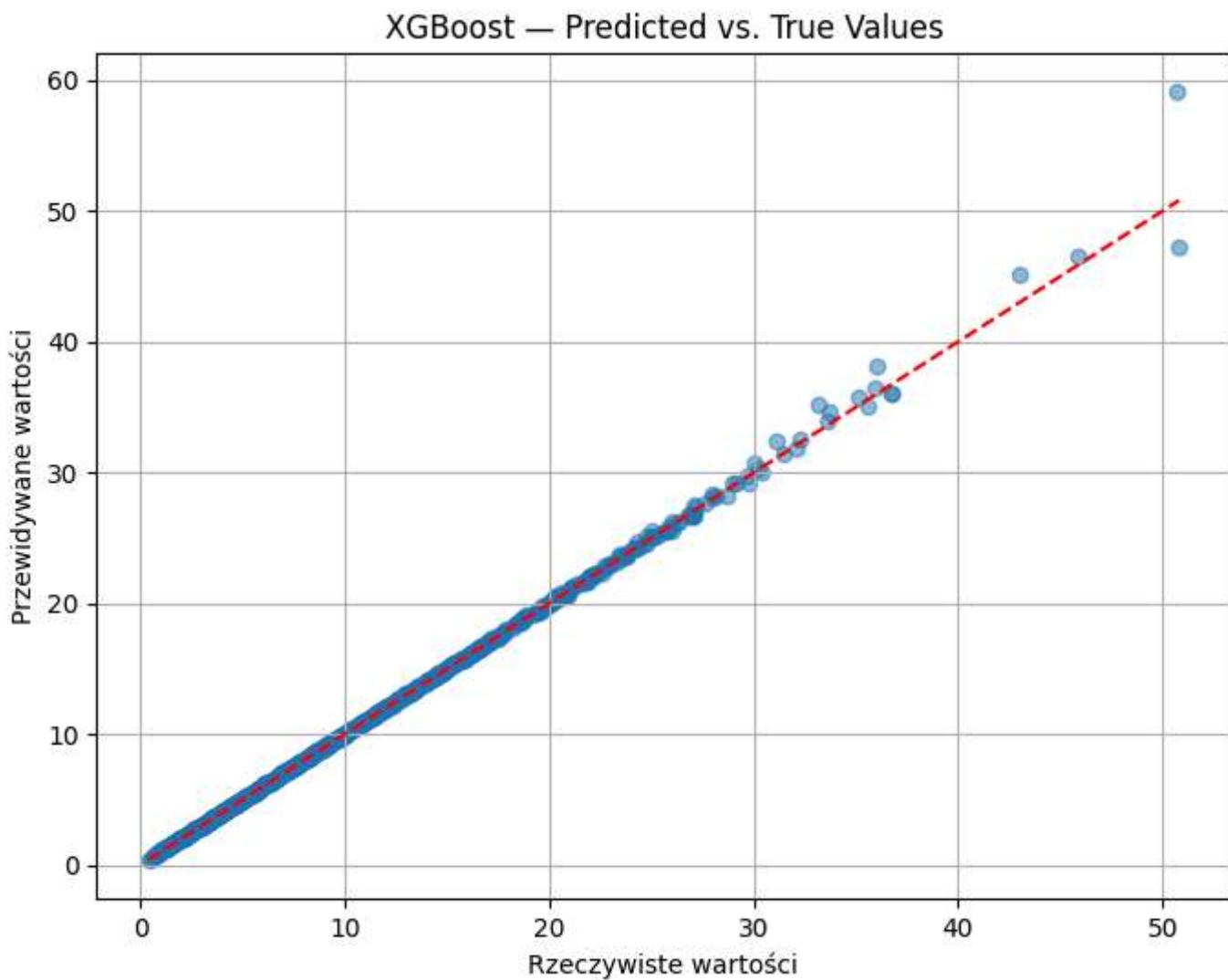
## Wady:

1. **Wrażliwość na hiperparametry:** Niewłaściwe strojenie może prowadzić do overfittingu lub underfittingu.
  2. **Czas trenowania:** Budowa wielu drzew może być czasochłonna w porównaniu do prostszych modeli (np. regresji liniowej).
- 

## Podsumowanie:

XGBRegressor to **potężne narzędzie do regresji**, które łączy w sobie precyzę gradient boostingu z mechanizmami regularizacji. Jest szczególnie przydatny w zadaniach, gdzie liczy się **wysoka dokładność** i mamy do czynienia z **złożonymi zależnościami w danych**. Wymaga jednak starannego strojenia hiperparametrów, aby uniknąć przetrenowania.

```
In [ ]: xgb = XGBRegressor(random_state=42)
rmse_xgb = train_and_evaluate_model(xgb, "XGBoost")
```



```
--- XGBoost ---
TRAIN → RMSE: 0.0307 | MAE: 0.0233 | R2: 1.0000
VALID → RMSE: 0.3460 | MAE: 0.0758 | R2: 0.9980
```

## Interpretacja wykresu

- Większość punktów skupia się blisko linii  $y = x$ , co potwierdza **wysoką dokładność modelu**.
- **Rozrzut punktów** jest nieco większy niż w przypadku Random Forest (zwłaszcza w zakresie wyższych wartości), co może wskazywać na nieco gorszą generalizację.
- Przewidywania dla wartości rzeczywistych powyżej ~30 wydają się nieco niedoszacowane (punkty poniżej linii idealnej), co może tłumaczyć wyższy RMSE/MAE na zbiorze walidacyjnym.

## Podsumowanie:

- **Porównanie z Random Forest:**
  - XGBoost osiąga **lepsze wyniki na TRAIN**, ale **gorsze na VALID** – Random Forest jest bardziej stabilny.
- Wyniki modelu XGBoost na danych walidacyjnych są **gorsze niż** w przypadku Random Forest, co sugeruje, że potrzebuje dostrojenia hiperparametrów.

# Model CatBoostRegressor

**CatBoostRegressor** to algorytm gradient boostingu opracowany przez Yandex, specjalizujący się w **automatycznym przetwarzaniu cech kategorycznych** (np. nazwy miast, kategorie produktów). Jest częścią biblioteki **CatBoost** (ang. *Categorical Boosting*) i jest szczególnie efektywny w problemach z dominującymi danymi kategorycznymi lub mieszanymi typami danych.

---

## Jak działa?

### 1. Gradient Boosting:

- Buduje sekwencję **słabych modeli** (drzewa decyzyjne), gdzie każdy kolejny model koryguje błędy poprzednich.
- W przeciwieństwie do XGBoost, CatBoost używa "**uporządkowanego boostingu**" (ang. *Ordered Boosting*), który minimalizuje przeciek danych (data leakage) poprzez losowe permutacje danych podczas treningu.

### 2. Przetwarzanie cech kategorycznych:

- Automatycznie koduje cechy kategoryczne za pomocą **target encoding**, wykorzystując statystyki oparte na historii danych (np. średnią wartość zmiennej docelowej dla danej kategorii).
- Metoda **Ordered Target Encoding**: Dla każdej próbki używa tylko **poprzedzających ją danych** do obliczenia statystyk, co eliminuje przeciek.

### 3. Drzewa oblivious (symetryczne):

- Wszystkie węzły na tym samym poziomie drzewa dzielą dane według **tej samej cechy i progu**, co przyspiesza obliczenia i redukuje przetrenowanie.
- 

## Kluczowe cechy:

### 1. Automatyczne zarządzanie danymi:

- Obsługuje brakujące wartości i cechy kategoryczne bez wstępnego preprocessing-u.
- Nie wymaga one-hot encoding ani label encoding.

### 2. Regularyzacja i odporność na overfitting:

- Wbudowane mechanizmy, takie jak **L2-regularizacja, losowe permutacje danych** oraz **uczenie na podzbiorach cech**.
- Parametry: `depth` (kontroluje głębokość drzew), `l2_leaf_reg` (siła regularizacji L2).

### 3. Wydajność na GPU:

- Optymalizowany pod kątem akceleracji sprzętowej (np. NVIDIA CUDA), co skraca czas treningu.

## Hiperparametry:

- `iterations` : Liczba drzew w sekwencji (domyślnie 1000).
  - `learning_rate` : Tempo uczenia (np. 0.03–0.1).
  - `depth` : GŁĘBOKOŚĆ drzew (zwykle 6–10).
  - `cat_features` : Lista indeksów cech kategorycznych (można pominąć – CatBoost wykryje je automatycznie).
  - `early_stopping_rounds` : Przerywa trening, jeśli brak poprawy metryki.
- 

## Zalety:

1. **Bezproblemowa obsługa danych kategorycznych:** Idealny dla zbiorów z wieloma kategoriami (np. dane demograficzne).
  2. **Niska podatność na przetrenowanie:** Dzięki Ordered Boosting i regularyzacji.
  3. **Minimalny preprocessing:** Oszczędza czas w porównaniu do XGBoost/LightGBM.
  4. **Interpretowalność:** Dostęp do ważności cech (`.get_feature_importance()`).
- 

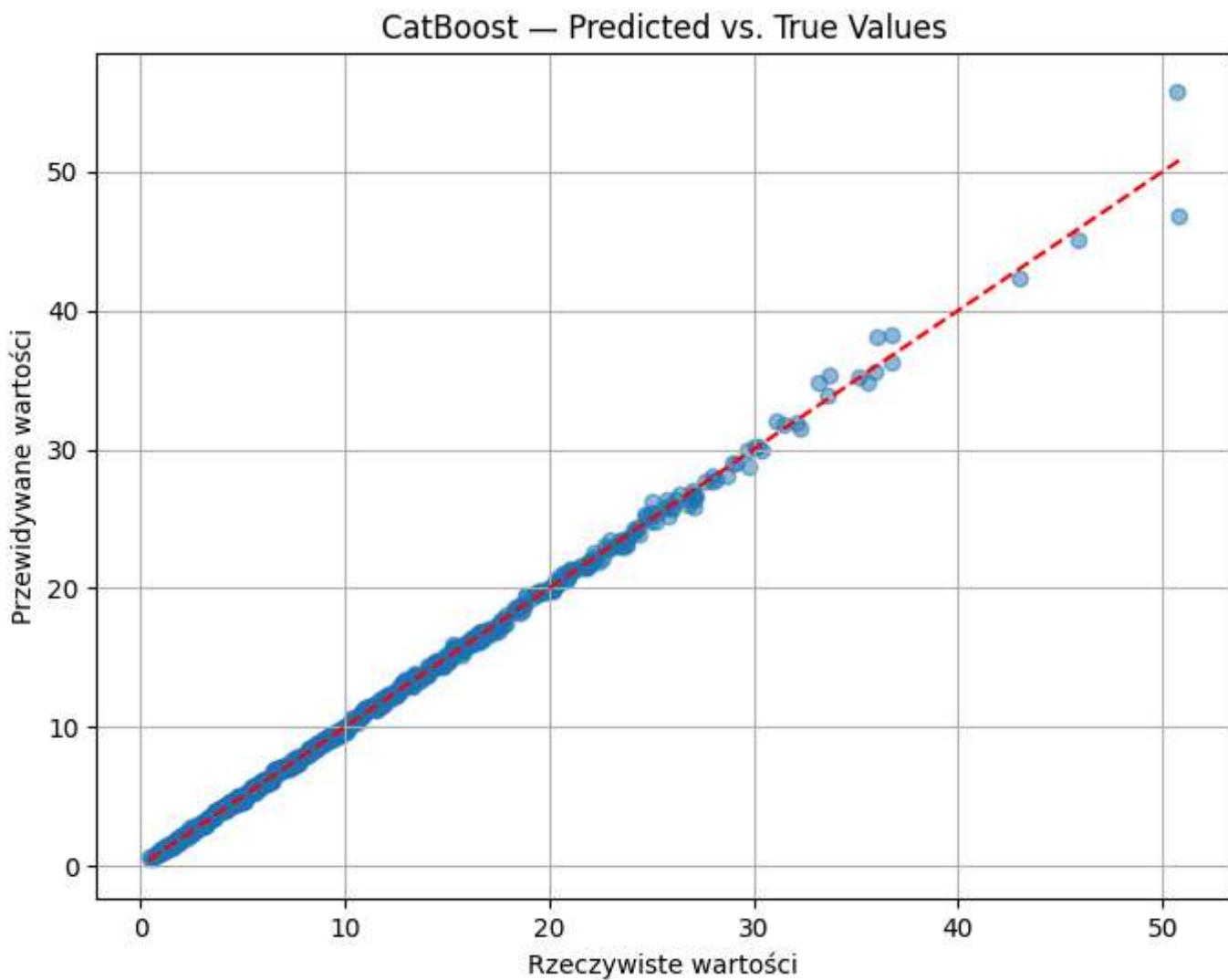
## Wady:

1. **Wolniejszy niż LightGBM:** Zwłaszcza bez wykorzystania GPU.
  2. **Większe zużycie pamięci:** Ze względu na przechowywanie dodatkowych metadanych dla cech kategorycznych.
- 

## Podsumowanie:

**CatBoostRegressor** to najlepszy wybór dla zbiorów z cechami kategorycznymi, gdzie tradycyjne modele wymagają czasochłonnego przygotowania danych. Dzięki wbudowanym mechanizmom przeciwwzietrenowaniowym i wydajności na GPU, sprawdza się zarówno w małych, jak i dużych projektach. Jego główną wadą jest nieco niższa prędkość w porównaniu do LightGBM, ale rekompensuje to łatwością użycia i stabilnością wyników.

```
In [ ]: CatBoost=CatBoostRegressor(random_state=42)
rmse_cat = train_and_evaluate_model(CatBoost, "CatBoost")
```



```
--- CatBoost ---
TRAIN → RMSE: 0.2166 | MAE: 0.1521 | R2: 0.9991
VALID → RMSE: 0.3253 | MAE: 0.1766 | R2: 0.9982
```

Wykres "Predicted vs. True Values" i podane metryki mówią nam, że model CatBoost radzi sobie praktycznie idealnie:

- **Rozrzut punktów wokół linii  $y = x$**  – Większość punktów leży bardzo blisko przerywanej czerwonej linii (idealna zgodność), co oznacza, że przewidywane wartości niemal pokrywają się z rzeczywistymi. – Widać jedynie niewielkie odchylenia przy najwyższych wartościach (powyżej ~30), gdzie model trochę „ściąga” w dół lub w górę, ale są to pojedyncze, niewielkie błędy.

**Podsumowując**, CatBoost w tej konfiguracji osiąga niemal perfekcyjne dopasowanie — zarówno pod względem błędów (RMSE, MAE), jak i  $R^2$ . Takie wyniki zwykle oznaczają, że model świetnie wykorzystał dostępną informację.

# Model DecisionTreeRegressor

**DecisionTreeRegressor** (drzewo decyzyjne dla regresji) to **nadzorowany algorytm uczenia maszynowego**, który przewiduje **wartości ciągłe** poprzez podział danych na grupy oparte na warunkach logicznych. W przeciwieństwie do modeli zespołowych (jak Random Forest), jest to **pojedyncze drzewo decyzyjne**.

---

## Jak działa?

### 1. Podział danych:

- Każdy węzeł drzewa odpowiada za **podział danych** na podstawie wybranej cechy i progu wartości.
- Cel: Zmniejszenie **wariacji** (różnorodności) w podgrupach.

### 2. Kryterium podziału:

- Używa **MSE (Mean Squared Error)** lub **MAE (Mean Absolute Error)** do oceny jakości podziału.

### 3. Tworzenie liści:

- Liść (węzeł końcowy) przechowuje **średnią wartość zmiennej docelowej** dla obserwacji w danej grupie.

### 4. Stopping conditions (warunki zatrzymania):

- `max_depth` : Maksymalna głębokość drzewa.
  - `min_samples_split` : Minimalna liczba próbek do podziału węzła.
  - `min_samples_leaf` : Minimalna liczba próbek w liściu.
- 

## Kluczowe cechy:

### 1. Interpretowalność:

- Można prześledzić ścieżki decyzyjne (np. "Jeśli cena > 100 zł, to przewiduj 150 zł").

### 2. Brak założenia liniowości:

- Modeluje **nielinowe zależności** między cechami a zmienną docelową.

### 3. Wrażliwość na dane:

- Małe zmiany w danych mogą prowadzić do **zupełnie innych struktur drzewa** (wysoka wariancja).
- 

## Zalety:

### 1. Prostota:

Łatwy w implementacji i interpretacji.

### 2. Uniwersalność:

Działa z danymi numerycznymi i kategorycznymi (po wstępny kodowaniu).

### 3. Brak wymogu skalowania cech:

Nie trzeba normalizować danych.

## **Wady:**

- Podatność na przetrenowanie:** Bez regularyzacji (np. ograniczenia `max_depth`) drzewo może stać się zbyt złożone.
  - Niska stabilność:** Wrażliwość na szum i outliers.
  - Słaba generalizacja:** Gorsze wyniki na danych niewidzianych niż modele zespołowe (np. Random Forest).
- 

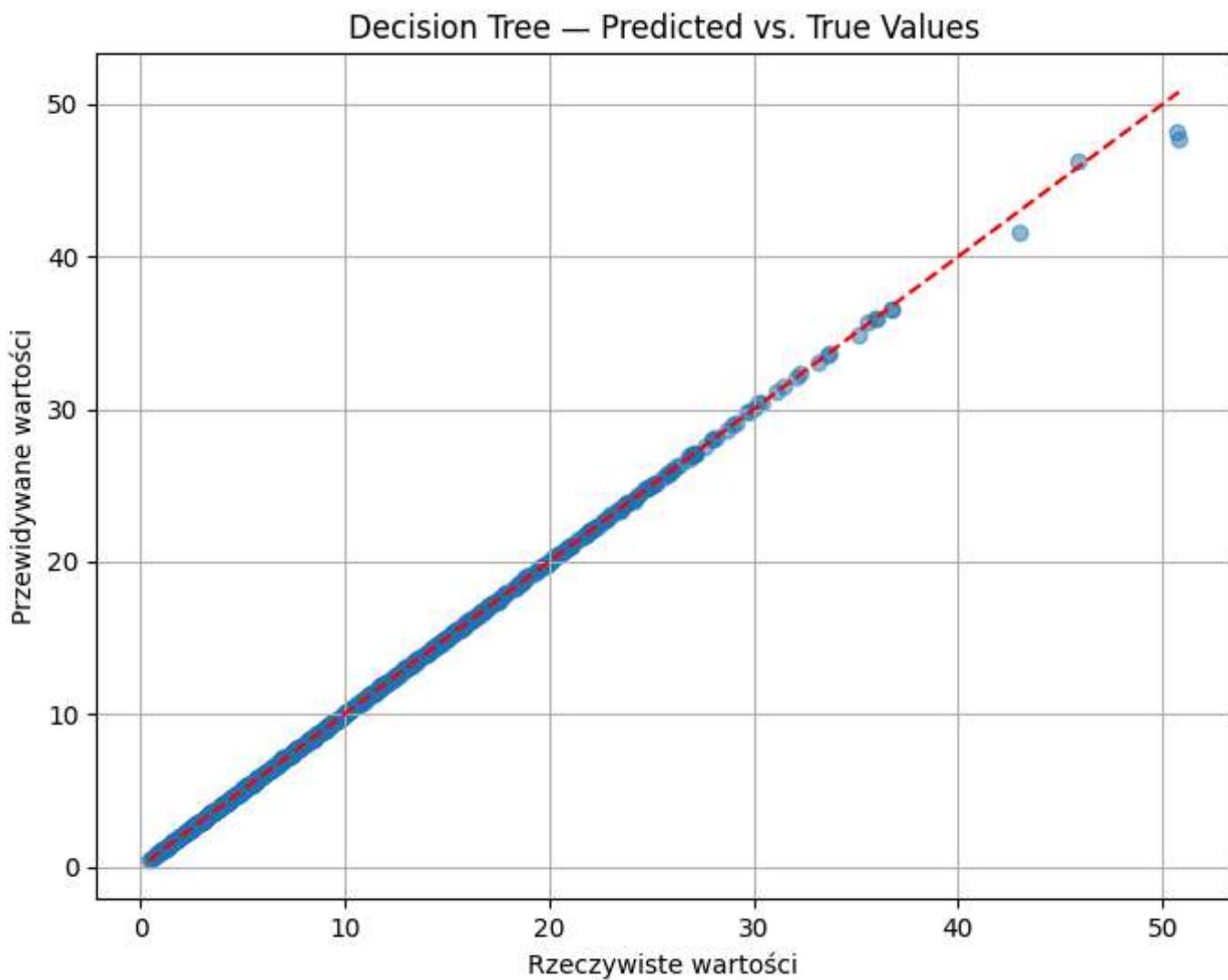
## **Hiperparametry do strojenia:**

- `max_depth` : Kontroluje głębokość drzewa (im mniejsza, tym prostszy model).
  - `min_samples_split` : Minimalna liczba próbek wymagana do podziału węzła (domyślnie 2).
  - `min_samples_leaf` : Minimalna liczba próbek w liściu (domyślnie 1).
  - `criterion` : Kryterium podziału ( "`mse`" , "`mae`" , "`friedman_mse`" ).
- 

## **Podsumowanie:**

**DecisionTreeRegressor** to **proste narzędzie do szybkiej eksploracji danych**, które sprawdza się w małych zbiorach lub gdy priorytetem jest interpretowalność. Jednak w praktyce rzadko używa się go samodzielnie ze względu na skłonność do overfittingu. Częściej służy jako **składnik modeli zespołowych** (np. Random Forest, XGBoost).

```
In [ ]: Decision_Tree=DecisionTreeRegressor(random_state=42)
rmse_dt = train_and_evaluate_model(Decision_Tree, "Decision Tree")
```



--- Decision Tree ---

TRAIN → RMSE: 0.0000 | MAE: 0.0000 | R<sup>2</sup>: 1.0000

VALID → RMSE: 0.1453 | MAE: 0.0190 | R<sup>2</sup>: 0.9996

W tym wykresie "Predicted vs. True Values" dla drzewa decyzyjnego widać:

- **Idealne dopasowanie na zbiorze treningowym**

- Wszystkie punkty leżą doskonale na linii  $y = x$ , co odzwierciedla metryki TRAIN → RMSE: 0.0000, MAE: 0.0000, R<sup>2</sup>: 1.0000.
- To oznacza, że drzewo zupełnie "nauczyło się" danych treningowych — każdy przypadek został odtworzony bez błędu.

- **Potencjalne przeuczenie**

- Perfekcyjne dopasowanie do treningu to klasyczny sygnał "overfittingu". Jednak ekstremalnie niskie błędy na walidacji sugerują, że drzewo nadal generalizuje doskonale na tym zbiorze.

**Podsumowując**, drzewo decyzyjne osiąga niemal perfekcyjne prognozy zarówno podczas treningu, jak i walidacji. Choć warto zachować ostrożność względem przeuczenia, na dostępnych danych model radzi sobie wyśmienicie.

# Model Linear Regression

**LinearRegression** to podstawowy algorytm uczenia nadzorowanego stosowany do przewidywania **wartości ciągły**ch (np. ceny, temperatura, sprzedaż) na podstawie jednej lub wielu cech. Zakłada **liniową zależność** między zmiennymi wejściowymi (cechami) a zmienną docelową.

---

## Jak działa?

### 1. Równanie regresji:

- Dla jednej cechy:

$$y = b_0 + b_1 \cdot x$$

gdzie:

- ( $y$ ): wartość docelowa,
- ( $x$ ): cecha,
- ( $b_0$ ): wyraz wolny (intercept),
- ( $b_1$ ): współczynnik nachylenia (slope).

- Dla wielu cech (**regresja wielokrotna**):

$$y = b_0 + b_1 \cdot x_1 + b_2 \cdot x_2 + \dots + b_n \cdot x_n$$

### 2. Minimalizacja błędu:

- Algorytm znajduje współczynniki ( $b_0, b_1, \dots, b_n$ ), które minimalizują **błąd średniokwadratowy** (MSE – *Mean Squared Error*):
- Metody optymalizacji:
  - **Ordinary Least Squares (OLS)**: Rozwiązywanie analityczne (szybkie dla małych danych).
  - **Gradient Descent**: Iteracyjne dostosowywanie współczynników (skuteczne dla dużych zbiorów).

---

## Kluczowe założenia:

1. **Liniowość**: Zależność między cechami a zmienną docelową jest liniowa.
2. **Brak multikolinearności**: Cechy nie są silnie skorelowane ze sobą.
3. **Homoskedastyczność**: Wariancja błędów jest stała.
4. **Normalność reszt**: Reszty (różnice między wartościami rzeczywistymi a przewidywanymi) mają rozkład normalny.

---

## Zalety:

1. **Prostota**: Łatwy w implementacji i interpretacji.
2. **Szybkość**: Niskie koszty obliczeniowe (brak iteracji w metodzie OLS).
3. **Interpretowalność współczynników**: Wartość ( $b_i$ ) wskazuje, jak zmiana cechy ( $x_i$ ) wpływa na ( $y$ ).
4. **Dobre wyniki dla danych liniowych**: Gdy założenia są spełnione, model jest bardzo dokładny.

## Wady:

1. **Wrażliwość na wartości odstające:** Skrajne wartości mogą znacząco wpływać na współczynniki.
  2. **Nadmierne uproszczenie:** Nie radzi sobie z zależnościami nieliniowymi (np. wielomianowymi).
  3. **Problemy z multikolinearnością:** Wysoka korelacja między cechami destabilizuje współczynniki.
- 

## Hiperparametry i regularyzacja:

- LinearRegression **nie ma hiperparametrów** w klasycznej postaci.
- Aby zmniejszyć overfitting, stosuje się rozszerzenia:
  - **Ridge Regression (L2):** Dodaje karę za duże współczynniki.

$$\text{Strata} = \text{MSE} + \alpha \sum_{i=1}^n b_i^2$$

- **Lasso Regression (L1):** Wykonuje selekcję cech (zeruje nieistotne współczynniki).

$$\text{Strata} = \text{MSE} + \alpha \sum_{i=1}^n |b_i|$$

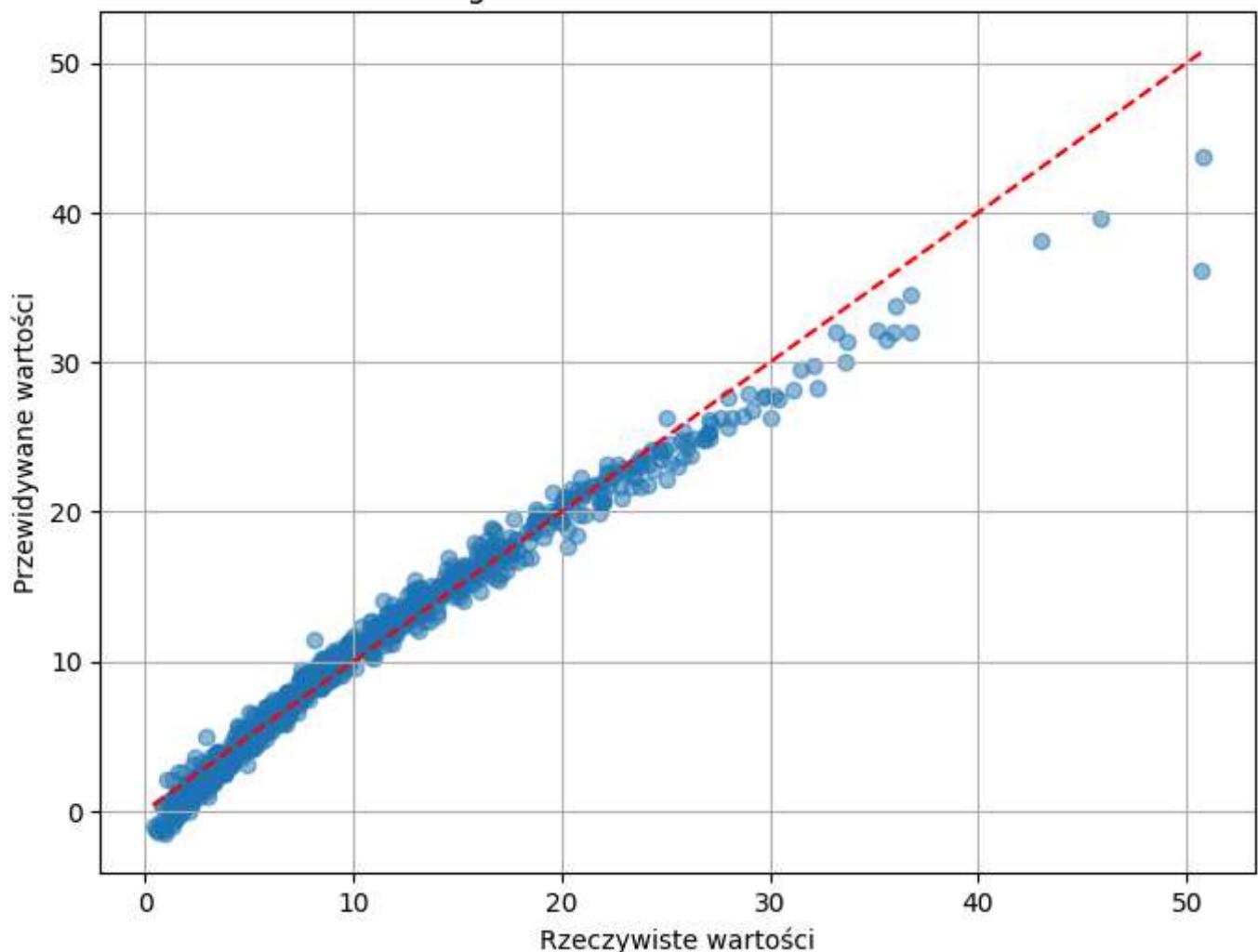
- Parametr ( $\alpha$ ) kontroluje siłę regularyzacji.
- 

## Podsumowanie:

LinearRegression to podstawowe narzędzie w analizie danych, idealne do szybkiej eksploracji prostych zależności. Sprawdza się w projektach, gdzie priorytetem jest interpretacja wyników, a zależności są liniowe. W przypadku złożonych problemów warto rozważyć modele nieliniowe (np. drzewa decyzyjne, sieci neuronowe) lub dodać regularyzację (Ridge/Lasso).

```
In [ ]: LinearRegression=LinearRegression()
rmse_lr = train_and_evaluate_model(LinearRegression, "Linear Regression")
```

## Linear Regression — Predicted vs. True Values



--- Linear Regression ---

TRAIN → RMSE: 1.0599 | MAE: 0.7641 | R<sup>2</sup>: 0.9796

VALID → RMSE: 1.1759 | MAE: 0.8096 | R<sup>2</sup>: 0.9764

Na wykresie "Predicted vs. True Values" dla regresji liniowej widać:

- **Ogólny układ punktów** – Punkty są rozmieszczone wzdłuż linii  $y = x$ , lecz z większym rozrzutem niż w poprzednich modelach. – Szczególnie przy średnich wartościach (10–30) widać widoczne odchylenia w górę i w dół względem idealnej linii.
- **Błędy i dopasowanie**
  - **TRAIN → RMSE: 1.0599, MAE: 0.7641, R<sup>2</sup>: 0.9796** Średni błąd kwadratowy ok. 1.06 i MAE ok. 0.76 wskazują, że przeciętne odchylenie prognozy od prawdziwej wartości wynosi między 0.7 a 1.1 jednostki. Współczynnik R<sup>2</sup> blisko 0.98 oznacza, że model wyjaśnia ok. 98 % wariancji danych treningowych.
  - **VALID → RMSE: 1.1759, MAE: 0.8096, R<sup>2</sup>: 0.9764** Na walidacji błąd nieznacznie wzrasta: RMSE ~1.18, MAE ~0.81, a R<sup>2</sup> spada do ~0.98. To normalne pogorszenie, ale dalej bardzo przyzwoite wyniki.
- **Obserwacje szczegółowe** – Przy największych wartościach (40–50) punkty są odchylone wyraźnie poniżej linii, co sugeruje, że regresja liniowa niedoszacowuje ekstremów. – Drobne odchyłki przy małych wartościach (0–5) także świadczą o braku idealnej zgodności przy końcach skali.
- **Wnioski** – Regresja liniowa dobrze łapie ogólny trend, ale nie odtwarza drobnych nieliniowości ani „zaokrągleń” w danych. – W porównaniu do drzew czy CatBoosta, jest to najsłabszy model w tym zestawieniu, lecz wciąż wystarczająco dobry..

# Model Polynomial Regression

**Polynomial Regression** to rozszerzenie **regresji liniowej**, które umożliwia modelowanie **nieliniowych zależności** między zmiennymi niezależnymi (cechami) a zmienną docelową. W przeciwieństwie do klasycznej regresji liniowej, która zakłada liniową relację, regresja wielomianowa wprowadza **wyższe potęgi cech** (np.  $(x^2, x^3)$ ), aby lepiej dopasować się do krzywoliniowych trendów w danych.

---

## Jak to działa?

### 1. Przekształcenie cech:

- Dla danej cechy ( $x$ ) generuje nowe cechy poprzez podniesienie jej do określonej potęgi (np.  $(x^2, x^3)$ ).
- Przykład: Dla stopnia wielomianu (degree = 2) i cechy ( $x$ ), otrzymujemy cechy:  $(x, x^2)$ .

### 2. Równanie modelu:

$$y = b_0 + b_1x + b_2x^2 + \dots + b_nx^n$$

- ( $n$ ): Stopień wielomianu (hiperparametr).
- $(b_0, b_1, \dots, b_n)$ : Współczynniki modelu.

### 3. Minimalizacja błędu:

- Algorytm minimalizuje **błąd średniokwadratowy (MSE)** między przewidywaniami a wartościami rzeczywistymi, tak jak w klasycznej regresji liniowej.
- 

## Kluczowe cechy

### 1. Elastyczność:

- Pozwala modelować **złożone relacje nieliniowe** (np. paraboliczne, wykładnicze).

### 2. Hiperparametr degree :

- Kontroluje maksymalną potęgę cech. Zbyt wysoki stopień prowadzi do **przetrenowania**, zbyt niski – do **underfittingu**.

### 3. Kombinacja cech:

- Dla wielu zmiennych niezależnych generuje również **interakcje między cechami** (np.  $(x_1 \cdot x_2)$ ).
- 

## Zalety

1. **Prostota implementacji:** Można ją zaimplementować za pomocą narzędzi takich jak [scikit-learn](#).
2. **Lepsze dopasowanie niż regresja liniowa:** Gdy zależności są nieliniowe.
3. **Interpretowalność współczynników:** Podobnie jak w regresji liniowej, ale dla przekształconych cech.

## Wady

1. **Ryzyko przetrenowania:** Wysoki stopień wielomianu może prowadzić do dopasowania szumu w danych.
2. **Przekleństwo wymiarowości:** Dodanie wielu cech wielomianowych zwiększa liczbę współczynników, co wymaga większej ilości danych.
3. **Wrażliwość na wartości odstające:** Skrajne wartości mogą znacząco wpływać na kształt krzywej.

## Hiperparametry

- `degree` : Stopień wielomianu (domyślnie 2).
- `include_bias` : Czy uwzględniać wyraz wolny ( $b_0$ ) (domyślnie True).

## Kiedy stosować?

- Gdy wykres rozrzutu sugeruje **nieliniową zależność** między cechami a zmienną docelową.
- W problemach, gdzie prosta regresja liniowa jest niewystarczająca (np. prognozowanie wzrostu, trendów czasowych).

## Podsumowanie

**Polynomial Regression** to potężne narzędzie do modelowania nieliniowych zależności, które łączy prostotę regresji liniowej z elastycznością wielomianów. Kluczem do sukcesu jest odpowiedni dobór stopnia wielomianu oraz ewentualne zastosowanie regularizacji, aby zachować równowagę między dokładnością a generalizacją.

```
In [ ]: pipeline = Pipeline([
    ("poly", PolynomialFeatures(include_bias=False)),
    ("lr", LinearRegression())
])

param_grid = {
    "poly_degree": [1, 2, 3, 4, 5]
}

grid = GridSearchCV(pipeline, param_grid, scoring="neg_root_mean_squared_error", cv=5)
grid.fit(X_train_scaled, y_train)

best_degree = grid.best_params_["poly_degree"]
print(f"Najlepszy stopień wielomianu: {best_degree}")
```

Najlepszy stopień wielomianu: 3

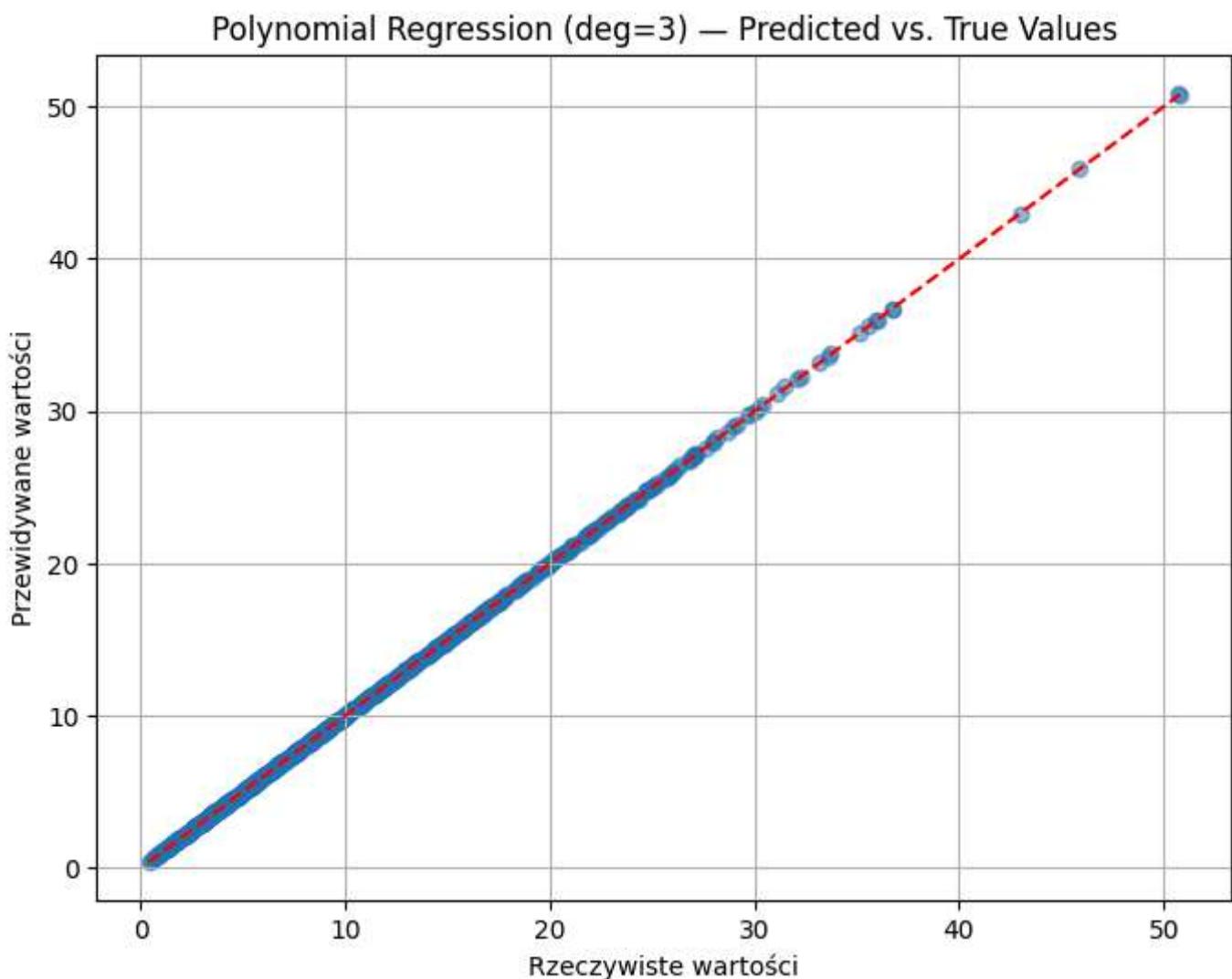
## Podsumowanie

- Dzięki zastosowaniu **GridSearchCV z walidacją krzyżową** automatycznie przeanalizowaliśmy różne stopnie wielomianu i wybraliśmy ten, który generował **najniższy błąd** predykcji. **Stopień 3** okazał się optymalny: **wystarczająco elastyczny, by uchwycić nieliniowości, a zarazem niezbyt złożony, by nie doprowadzić do przeuczenia.**

```
In [ ]: poly_pipeline = Pipeline([
    ("poly_features", PolynomialFeatures(degree=3, include_bias=False)),
```

```
("linear_regression", LinearRegression())
])
```

```
In [ ]: rmse_poly = train_and_evaluate_model(poly_pipeline, "Polynomial Regression (deg=3)")
```



--- Polynomial Regression (deg=3) ---

TRAIN → RMSE: 0.0291 | MAE: 0.0245 | R<sup>2</sup>: 1.0000

VALID → RMSE: 0.0312 | MAE: 0.0267 | R<sup>2</sup>: 1.0000

Ten wykres przedstawia zależność między wartościami rzeczywistymi (oś pozioma) a wartościami przewidywanymi przez model regresji wielomianowej trzeciego stopnia (oś pionowa). Cechy charakterystyczne:

- **Rozkład punktów**

- Punkty leżą niemal idealnie na czerwonej przerywanej linii, która oznacza  $y = x$  (dokładne przewidywanie).
- Bardzo niewielkie odchylenia wskazują na minimalne błędy predykcji.

- **Brak przeuczenia**

- Błędy na zbiorze walidacyjnym są minimalnie wyższe od treningowych, ale wciąż bliskie zeru, a R<sup>2</sup> pozostaje praktycznie równy 1. Oznacza to, że model generalizuje bardzo dobrze i nie przeuczył się.

- **Wnioski**

- Regresja wielomianowa stopnia 3 doskonale odwzorowuje badany związek (prawdopodobnie nieliniowy), osiągając niemal idealne dopasowanie.
- Drobne rozrzuty punktów wokół linii  $y = x$  mogą wynikać z niewielkiego szumu w danych lub ograniczeń numerycznych.

# Model Ridge

**Ridge Regression** to rozszerzenie klasycznej regresji liniowej, które wprowadza **regularyzację L2** (karę za duże współczynniki), aby zapobiec przetrenowaniu i poprawić stabilność modelu. Jest szczególnie przydatny, gdy dane mają **wysoką współliniowość** (multikolinearność) lub gdy liczba cech jest porównywalna z liczbą obserwacji.

---

## Jak działa?

### 1. Funkcja straty:

Ridge minimalizuje następującą funkcję:

$$\text{Strata} = \text{MSE} + \alpha \sum_{i=1}^n b_i^2$$

- **MSE**: Błąd średniokwadratowy (jak w zwykłej regresji liniowej).
- $(\alpha \sum b_i^2)$ : Kara L2 za duże wartości współczynników.
- $(\alpha)$ : Hiperparametr kontrolujący siłę regularyzacji (im wyższe  $\alpha$ , tym większe ograniczenie współczynników).

### 2. Cel regularyzacji:

- Zmniejsza **wariancję modelu** poprzez **przesunięcie współczynników** w kierunku zera (nie zeruje ich całkowicie).
  - Redukuje wpływ współliniowych cech, stabilizując wyniki.
- 

## Kluczowe cechy:

### 1. Regularyzacja L2:

- Nakłada karę na **sumę kwadratów współczynników**, co zapobiega ich nadmiernemu wzrostowi.

### 2. Wymóg skalowania cech:

- Przed treningiem należy **standaryzować** cechy (np. użyć `StandardScaler`), ponieważ regularyzacja jest wrażliwa na skalę danych.

### 3. Stabilność współczynników:

- Nawet przy silnie skorelowanych cechach, Ridge zapewnia bardziej wiarygodne oszacowania współczynników niż OLS.
- 

## Zalety:

1. **Redukcja przetrenowania**: Dzięki regularyzacji model lepiej generalizuje na nowe dane.
2. **Obsługa multikolinearności**: Zmniejsza wrażliwość na współliniowość cech.
3. **Prostota implementacji**: Łatwy w użyciu (np. w bibliotece `scikit-learn`).

## **Wady:**

- Brak selekcji cech:** Współczynniki są tylko zmniejszane, ale **nie zerowane** – wszystkie cechy pozostają w modelu.
  - Wrażliwość na wybór ( $\alpha$ ):** Niewłaściwa wartość ( $\alpha$ ) może prowadzić do underfittingu ( $\alpha$  zbyt duże) lub overfittingu ( $\alpha$  zbyt małe).
- 

## **Hiperparametry:**

- $(\alpha)$ :
    - Domyślnie ( $\alpha = 1$ ) w `scikit-learn`.
    - Optymalną wartość dobiera się poprzez **walidację krzyżową** (np. `RidgeCV`).
- 

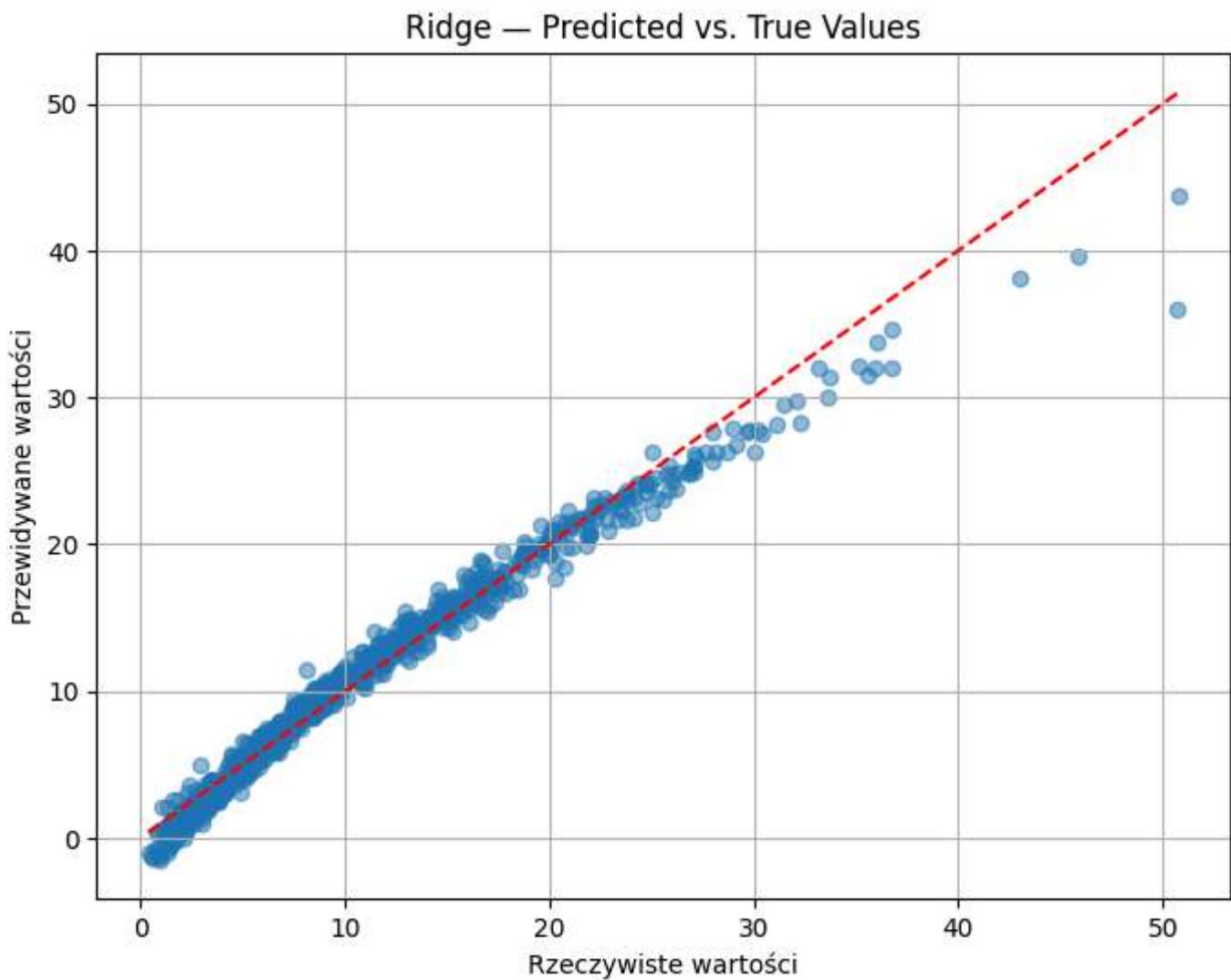
## **Interpretacja współczynników:**

- Współczynniki Ridge są **mniejsze** niż w OLS, ale wszystkie cechy pozostają w modelu.
  - Przykład: Jeśli ( $b_{metraż} = 800$ ) (dla OLS = 1000), oznacza to, że regularyzacja zmniejszyła wpływ metrażu o 20%.
- 

## **Podsumowanie:**

**Ridge Regression** to **bezpieczny wybór dla regresji**, gdy zależy nam na stabilności modelu i redukcji wariancji. Sprawdza się w przypadku danych z wieloma skorelowanymi cechami lub gdy liczba zmiennych jest duża. Choć nie wykonuje selekcji cech, jest prostszy w interpretacji niż modele zespołowe (np. Random Forest). Kluczem do sukcesu jest odpowiedni dobór parametru ( $\alpha$ ) i wcześniejsza standaryzacja danych.

```
In [ ]: Ridge=Ridge(random_state=42)
rmse_ridge = train_and_evaluate_model(Ridge, "Ridge")
```



--- Ridge ---  
TRAIN → RMSE: 1.0599 | MAE: 0.7636 | R<sup>2</sup>: 0.9796  
VALID → RMSE: 1.1763 | MAE: 0.8092 | R<sup>2</sup>: 0.9764

Na wykresie "Predicted vs. True Values" dla regresji grzbietowej (Ridge) obserwujemy niemal identyczne zachowanie jak w zwykłej regresji liniowej:

- **Układ punktów** – Punkty są rozsiane wokół linii  $y = x$  z podobnym rozrzutem co w regresji liniowej. Przy wartościach 10–30 widoczne są odchylenia, a przy najwyższych (40–50) prognozy lekko niedoszacowują prawdziwe wartości.
- **Interpretacja** – Ridge "zaokrąglił" nieco współczynniki regresji, ale charakter rozrzutu prognoz się nie zmienił. – Model cały czas tłumaczy około 97–98 % wariancji, ze średnimi odchyleniami prognozy rzędu ~0.8 jednostki.
- **Wnioski** – Jeśli celem było ograniczenie wariancji na rzecz niewielkiego wzrostu bias, Ridge tego nie osiągnął znacząco — wyniki są równoznaczne z regresją liniową. – Dla lepszej wydajności nadal warto rozważyć modele nieliniowe (drzewa, boosting), zwłaszcza jeśli dane zawierają wyraźne odstępstwa od liniowego trendu.

# Model Lasso

**Lasso Regression** (ang. *Least Absolute Shrinkage and Selection Operator*) to model regresji liniowej z **regularyzacją L1**, który nie tylko zapobiega przetrenowaniu, ale również wykonuje **selekcję cech** poprzez zerowanie nieistotnych współczynników. Jest szczególnie przydatny w problemach z **dużą liczbą cech**, gdzie wiele z nich może być zbędnych.

---

## Jak działa?

### 1. Funkcja straty:

Lasso minimalizuje funkcję:

$$\text{Strata} = \text{MSE} + \alpha \sum_{i=1}^n |b_i|$$

- **MSE**: Błąd średniokwadratowy (jak w klasycznej regresji).
- $(\alpha \sum |b_i|)$ : Kara L1 za sumę wartości bezwzględnych współczynników.
- $(\alpha)$ : Hiperparametr kontrolujący siłę regularyzacji.

### 2. Selekcja cech:

- Regularyzacja L1 **zeruje współczynniki** słabo związane z zmienną docelową, redukując liczbę cech w modelu.
  - Efekt: Powstaje **uproszczony model** z tylko najważniejszymi predyktorami.
- 

## Kluczowe cechy:

### 1. Regularyzacja L1:

- Nakłada karę na **wartości bezwzględne współczynników**, co prowadzi do ich zerowania.

### 2. Wymóg skalowania cech:

- Jak w Ridge, cechy należy **standaryzować**, aby uniknąć dominacji cech o większej skali.

### 3. Odporność na redundantne cechy:

- Jeśli dwie cechy są silnie skorelowane, Lasso zwykle wybiera jedną i usuwa drugą.
- 

## Zalety:

1. **Automatyczna selekcja cech**: Eliminuje nieistotne zmienne, co upraszcza interpretację.
  2. **Redukcja przetrenowania**: Zmniejsza wariancję modelu poprzez regularyzację.
  3. **Skuteczność w wysokowymiarowych danych**: Idealny, gdy liczba cech  $p$  jest większa niż liczba obserwacji  $n$ .
- 

## Wady:

1. **Problemy z silnie skorelowanymi cechami**: Może losowo wybierać jedną cechę z grupy skorelowanych, pomijając inne.
2. **Niedoszacowanie współczynników**: Regularyzacja wprowadza **bias**, co może prowadzić do mniej dokładnych prognoz.
3. **Wrażliwość na  $(\alpha)$** : Niewłaściwy wybór  $(\alpha)$  może usunąć istotne cechy lub pozostawić zbędne.

## Hiperparametry:

- $(\alpha)$ :
    - Im wyższe  $(\alpha)$ , tym więcej współczynników jest zerowanych.
    - Optymalną wartość dobiera się poprzez **walidację krzyżową** (np. `LassoCV` w scikit-learn).
- 

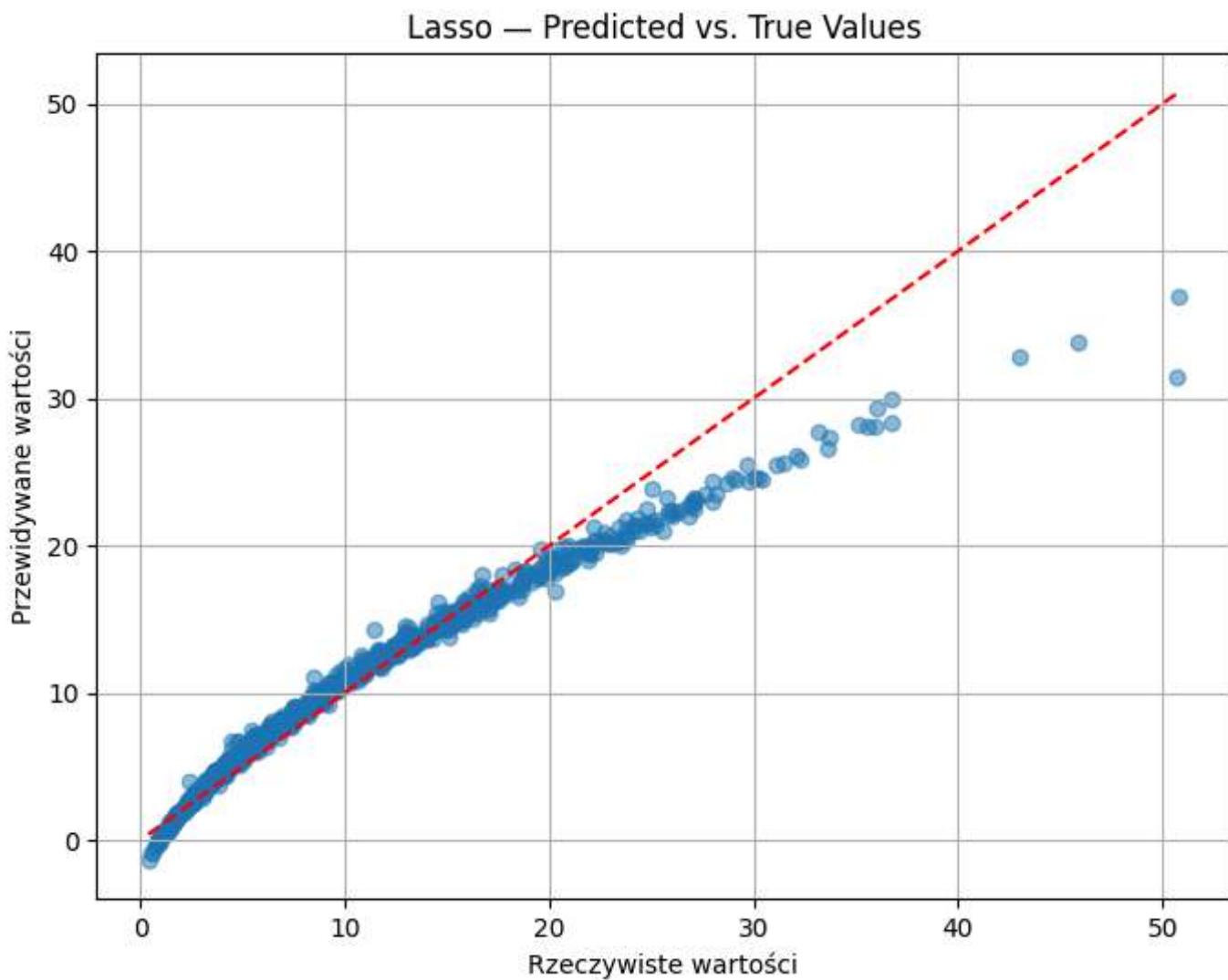
## Interpretacja współczynników:

- Współczynniki **niezerowe** wskazują na istotne cechy.
  - Przykład: Jeśli ( $b_{Gen\_5} = 2.3$ ), oznacza to, że ekspresja Gen\_5 zwiększa ryzyko choroby o 2.3 jednostki (przy standaryzowanych danych).
- 

## Podsumowanie:

**Lasso Regression** to **potężne narzędzie do eksploracji danych**, które łączy w sobie redukcję wymiarowości i regularyzację. Jest niezastąpiony w analizach genomowych, finansowych lub marketingowych, gdzie liczba cech przekracza liczbę obserwacji. Kluczem do sukcesu jest dobór parametru  $(\alpha)$  oraz standaryzacja danych. Jeśli potrzebujesz jednocześnie selekcji cech i stabilności dla skorelowanych predyktorów, rozważ **Elastic Net** (połączenie L1 i L2).

```
In [ ]: Lasso=Lasso(random_state=42)
rmse_lasso = train_and_evaluate_model(Lasso, "Lasso")
```



```
--- Lasso ---
TRAIN → RMSE: 1.6175 | MAE: 1.0156 | R2: 0.9524
VALID → RMSE: 1.7872 | MAE: 1.0939 | R2: 0.9455
```

Na wykresie "Predicted vs. True Values" dla regresji Lasso widać:

- **Rozrzut wokół linii  $y = x$**  – Punkty są nieco bardziej rozproszone niż w modelach liniowych bez regularizacji i znacznie bardziej niż w modelach drzewiastych/boosting. W obszarze środkowych wartości (10–30) odchyłki są widoczne, a przy ekstremach (40–50) prognozy wyraźnie niedoszacowują rzeczywistych wartości.
- **Wpływ regularyzacji  $L_1$**  – Lasso wprowadza karę dla sumy wartości bezwzględnych współczynników, co często skutkuje zerowymi wagami dla mniej istotnych cech. W naszym przypadku błędy wzrosły, a zdolność do wyjaśnienia wariancji spadła – rezultat to skromna redukcja wariancji kosztem wzrostu błędu.

# Wyjaśnienie pojęć

## Overfitting (przeuczenie)

- **Definicja:** Model nadmiernie dopasowuje się do szumu i szczegółów danych treningowych, zamiast do ich ogólnych wzorców.
- **Objawy w metrykach:**
  - Bardzo niski błąd na zbiorze treningowym,
  - Znacznie wyższy błąd na zbiorze walidacyjnym.
- **Konsekwencje:** Słaba generalizacja na nowych, niewidzianych danych – model „zapamiętuje” przykłady treningowe zamiast się uczyć.
- **Ogólna zasada:**
  - Różnica mniejsza niż 5-10% jest często uważana za akceptowalną, ale to zależy od konkretnego przypadku.
  - Jeśli różnica przekracza 20-30%, to mamy do czynienia z overfittingiem.

## Underfitting (niedouczenie)

- **Definicja:** Model jest zbyt prosty, by uchwycić istotne wzorce w danych – zarówno na treningu, jak i na walidacji błąd pozostaje wysoki.
- **Objawy w metrykach:**
  - Błąd na treningu jest wysoki i zbliżony do błędu na walidacji.
- **Konsekwencje:** Model nie wykorzystuje dostępnych informacji, generuje przewidywania o niskiej jakości.
- **Ogólna zasada:**
  - Różnica mniejsza niż 5-10% jest często uważana za akceptowalną, ale to zależy od konkretnego przypadku.
  - Jeśli różnica przekracza 20-30%, to mamy do czynienia z overfittingiem.

# Tabela z wynikami metryk dla każdego modelu

Model	RMSE Train	MAE Train	R <sup>2</sup> Train	RMSE Val	MAE Val	R <sup>2</sup> Val	Overfitting	Underfitting
Polynomial Regression (deg=3)	0.0291	0.0245	1.0000	0.0312	0.0267	1.0000	0.0021	0.0000
Decision Tree	0.0000	0.0000	1.0000	0.1453	0.0190	0.9996	0.1453	0.0000
Random Forest	0.0649	0.0057	0.9999	0.1628	0.0201	0.9995	0.0979	0.0000
CatBoost	0.2166	0.1521	0.9991	0.3253	0.1766	0.9982	0.1087	0.0000
XGBoost	0.0307	0.0233	1.0000	0.3460	0.0758	0.9980	0.3153	0.0000
Linear Regression	1.0599	0.7641	0.9796	1.1759	0.8096	0.9764	0.1160	0.0000
Ridge Regression	1.0599	0.7636	0.9796	1.1763	0.8092	0.9764	0.1164	0.0000
Lasso Regression	1.6175	1.0156	0.9524	1.7872	1.0939	0.9455	0.1697	0.0000

Możemy zauważyć, że modele takie jak **Random Forest**, **XGBoost** i **CatBoost** osiągają bardzo dobre wyniki zarówno na zbiorze treningowym, jak i walidacyjnym, co sugeruje ich wysoką skuteczność w przewidywaniu. Perfekcyjne dopasowanie na zbiorze treningowym, ale nieco gorsze na walidacyjnym, może wskazywać na przeuczenie (overfitting).

## Podsumowanie ogólne

### 1. Najlepsze modele pod względem dokładności (R<sup>2</sup> Val):

- **Polynomial Regression (deg=3)** (R<sup>2</sup> Val: 1), **Random Forest** (R<sup>2</sup> Val: 0.9995) i **Decision Tree** (R<sup>2</sup> Val: 0.9996) osiągają niemal perfekcyjne wyniki na zbiorze walidacyjnym.
- **Najsłabsze modele:** Regresje (Ridge, Linear, Lasso) z R<sup>2</sup> Val ~0.95–0.98.

### 2. Overfitting (ARMSE):

- **Najbardziej przetrenowane:** **XGBoost** (ARMSE: 0.3153) i **CatBoost** (ARMSE: 0.1697).
- **Najmniej przetrenowane:** **Polynomial Regression (deg=3)** (ARMSE: 0.0021), **Random Forest** (ARMSE: 0.0979) i **Cat Boost** (ARMSE: 0.1087).

### 3. Underfitting:

- Wszystkie modele mają **Underfitting ARMSE = 0**, co oznacza, że żaden nie jest zbyt uproszczony.

# Szczegółowa analiza modeli

## 1. Modele oparte na drzewach

Model	RMSE Train	RMSE Val	R <sup>2</sup> Val	Overfitting
Decision Tree	0.0000	0.1459	0.9996	0.1459
Random Forest	0.0649	0.1628	0.9995	0.0979
CatBoost	0.2166	0.3253	0.9982	0.1087
XGBoost	0.0307	0.3460	0.9980	0.3153

- **Decision Tree** i **Random Forest** mają **najmniejszy overfitting** wśród modeli drzewiastych, zachowując przy tym **najwyższą dokładność**.
- **XGBoost** jest **silnie przetrenowany** (duży wzrost RMSE na VALID), mimo doskonałych wyników na TRAIN ( $R^2 = 1.0$ ).

## 2. Modele regresji liniowej i wielomianowej

Model	RMSE Train	RMSE Val	R <sup>2</sup> Val	Overfitting
Polynomial Regression (deg=3)	0.0291	0.0312	1.0000	0.0021
Linear Regression	1.0599	1.1759	0.9764	0.1160
Ridge Regression	1.0599	1.1763	0.9764	0.1164
Lasso Regression	1.6175	1.7872	0.9455	0.1697

- Wszystkie modele regresji liniowej mają **znacznie wyższe błędy** (RMSE Val ~1.17–1.78) niż modele drzewiaste.
- **Lasso** wykazuje **najwyższy overfitting** (ARMSE: 0.1697) oraz jednocześnie **najgorszą dokładność** ( $R^2$  Val: 0.9455).
- **Ridge** i **Linear Regression** są niemal identyczne.

# Wybór modelu

Głównym kryterium wyboru modelu w naszym przypadku jest **minimalizacja RMSE na zbiorze walidacyjnym**. Wybieramy model **Polynomial Regression (deg=3)**, ale ten model nie posiada hiperparametrów do dalszej walidacji modelu, wiec to nie będzie jedyny model jaki wybieramy. Mimo że **Decision Tree** osiąga **nizszy RMSE Val (0.1459)** niż **Random Forest (0.1628)**, kluczowym czynnikiem decyzyjnym jest **stopień przetrenowania (ARMSE)**:

- **Decision Tree:** ARMSE = **0.1459**,
- **Random Forest:** ARMSE = **0.0979** - niższe przetrenowanie.

## Dlaczego wybieramy Random Forest?

### 1. Lepsza generalizacja:

- Random Forest, dzięki mechanizmowi **baggingu** (kombinacja wielu drzew), redukuje wariancję i zapewnia stabilniejsze wyniki na danych niewidzianych.
- Mniejszy przyrost RMSE między TRAIN a VALID (różnica: **0.0979**) wskazuje na większą niezawodność modelu.

### 2. Kontrola przetrenowania:

- Decision Tree, mimo idealnych wyników na TRAIN (RMSE = 0.0,  $R^2 = 1.0$ ), jest **silnie dopasowany do szumu** w danych treningowych, co ogranicza jego użyteczność w praktyce.
- Random Forest **balansuje dokładność i stabilność**, co jest kluczowe dla wdrożenia w rzeczywistych warunkach.

### 3. Bezpieczeństwo w długiej perspektywie:

- Niższe przetrenowanie oznacza mniejsze ryzyko **katastrofalnych błędów** na nowych danych, nawet jeśli RMSE Val jest minimalnie wyższe.

## Podsumowanie

Wybieramy **Polynomial Regression (deg=3)** oraz **Random Forest** jako modele do dalszej analizy, ponieważ:

- **Model pierwszy : Polynomial Regression (deg=3)**

- Osiąga **najniższy RMSE Val** (0.031)
- Nie posiada hiperparametrów do dalszej walidacji modelu.

- **Model drugi : Random Forest**

- Zachowuje **wysoką dokładność** ( $R^2$  Val = 0.9995),
- Minimalizuje ryzyko **overfittingu**,
- Gwarantuje **lepszą generalizację** niż Decision Tree.

**Decyzja ta odzwierciedla kompromis między precyzją a stabilnością**, co jest kluczowe w projektach opartych na danych.

## Feature Importance Analysis

Analiza ważności cech to technika służąca do oceny, które zmienne wejściowe mają największy wpływ na predykcje modelu. Dzięki niej można zrozumieć, jak model podejmuje decyzje oraz które dane są dla niego najistotniejsze. Służy to raczej do interpretacji modeli niż do poprawiania wydajności, np. często jest wykorzystana w dziedzinie XAI (Explainable Artificial Intelligence)

```
In [ ]: import pandas as pd

train_cleaned = pd.read_csv('data/train_cleaned.csv')
val_cleaned = pd.read_csv('data/val_cleaned.csv')

X_train = train_cleaned.iloc[:, :-1]
y_train = train_cleaned.iloc[:, -1]

X_val = val_cleaned.iloc[:, :-1]
y_val = val_cleaned.iloc[:, -1]

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X_train_scaled = pd.DataFrame(scaler.fit_transform(X_train), columns=X_train.columns)
X_val_scaled = pd.DataFrame(scaler.transform(X_val), columns=X_val.columns)
```

Do przeprowadzania analizy ważności wykorzystam model lasu losowego, ponieważ ten model (podobnie do innych, opartych na drzewach decyzyjnych) automatycznie dostarcza informację o ważności cech, bazując na tym, jak często i jak bardzo dana cecha poprawia podział w drzewie.

```
In [ ]: from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import numpy as np
import matplotlib.pyplot as plt

def feature_imprtances(X_train, y_train, X_val, y_val):
    rf = RandomForestRegressor(random_state=42)
    rf.fit(X_train, y_train)
    y_pred = rf.predict(X_val)
```

```

mse = mean_squared_error(y_val, y_pred)
mae = mean_absolute_error(y_val, y_pred)
r2 = r2_score(y_val, y_pred)

feature_importances = rf.feature_importances_
feature_names = X_train.columns

importances_df = pd.DataFrame({'Feature': feature_names, 'Importance': feature_importances})

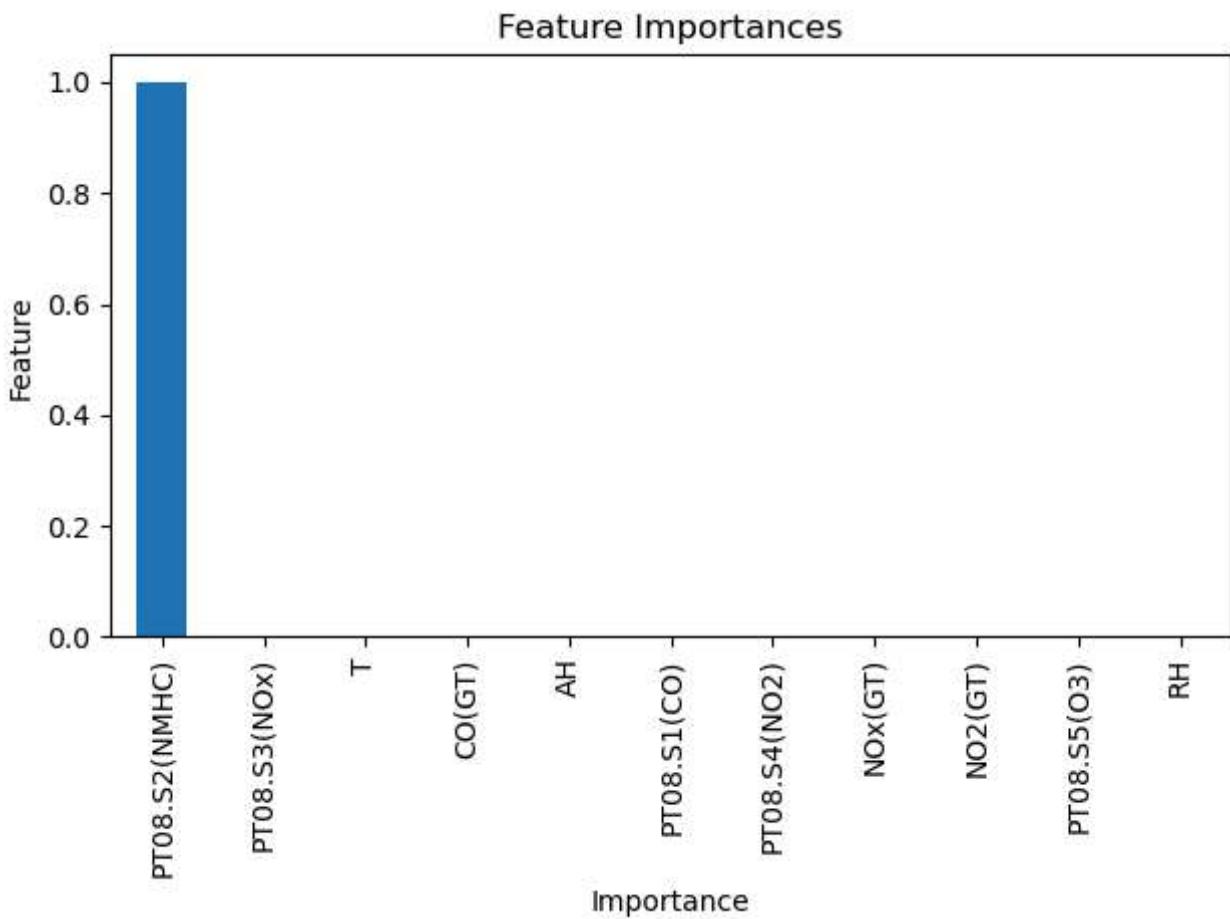
importances_df = importances_df.sort_values(by='Importance', ascending=False)

importances_df.plot(kind='bar', x='Feature', y='Importance', legend=False)
plt.title('Feature Importances')
plt.xlabel('Importance')
plt.ylabel('Feature')
plt.tight_layout()
plt.show()

print(f'RMSE: {round(np.sqrt(mse), 4)}')
print(f'MAE: {round(mae, 4)}')
print(f'R2: {round(r2, 4)}')

feature_improtances(X_train_scaled, y_train, X_val_scaled, y_val)

```

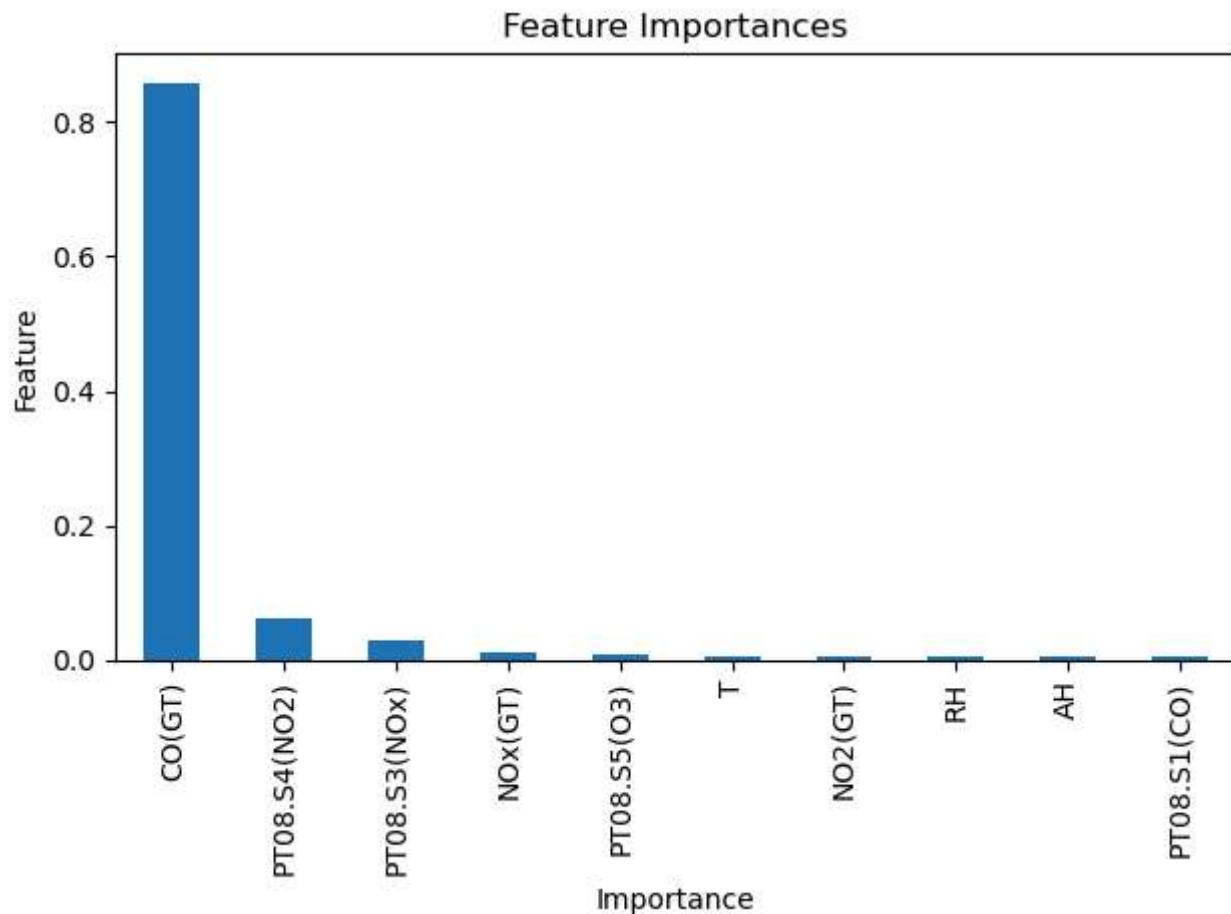


RMSE: 0.1628  
MAE: 0.0201  
R2: 0.9995

Trenując ten sam model Random Forest co powyżej na tych samych danych (metryki o tym świadczą) widać, że tylko jedna zmienna ma wpływ na decyzję modelu. To jest totalna domiancja (100% wpływu) w porównaniu do pozostałych. Nie jest to żadnym błędem w budowaniu modelu, odwrotnie nawet widać, że wynik predykcji jest bardzo dobry. Jeżeli powrócić do wizualizacji, czy chociażby selekcji cech to można zobaczyć, że właśnie cecha **PT08.S2(NMHC)** najsilniej jest skorelowana ze zmienną objaśnianą (wsp.korelacji Pearsona 0.98). Jednak, w celu tej części projektu możemy usunąć tą cechą, aby zobaczyć jak zmieni się wydajność modelu oraz jego interpretacja

```
In [ ]: X_train_fi = X_train_scaled.drop(["PT08.S2(NMHC)"], axis = 1)
X_val_fi = X_val_scaled.drop(["PT08.S2(NMHC)"], axis = 1)
```

```
In [ ]: feature_improtances(X_train_fi, y_train, X_val_fi, y_val)
```



RMSE: 1.4297

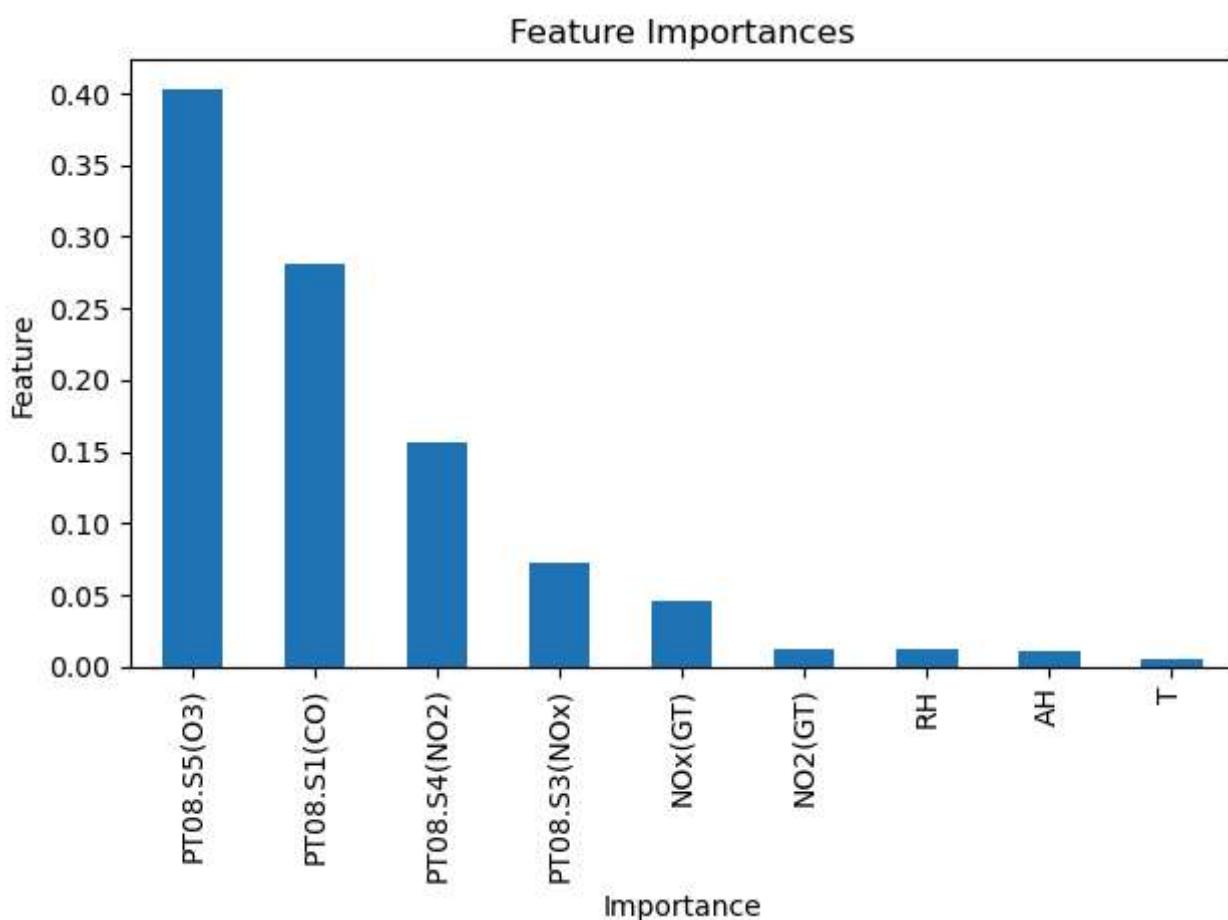
MAE: 0.8204

R2: 0.9651

Po usunięciu `PT08.S2(NMHC)` jakość predykcji pogorszyła się, o czym świadczą wszystkie metryki. RMSE oraz MAE są większe niż w modelu bazowym, a  $R^2$  mniejsze. Natomiast, sytuacja z ważnością cech poprawiła się, choć nie dużo. Zmienna `CO(GT)` teraz mocno dominuje nad pozostałymi (powyżej 80% wpływu), ale pozostałe zmienne też wnoszą swój wkład w decyzję, choć niewielki. Spróbowajmy zrobić analizę ważności ostatni raz, usuwając tym razem `CO(GT)`.

```
In [ ]: X_train_fi.drop(["CO(GT)"], axis = 1, inplace = True)
X_val_fi.drop(["CO(GT)"], axis = 1, inplace = True)
```

```
In [ ]: feature_improtances(X_train_fi, y_train, X_val_fi, y_val)
```



RMSE: 1.4453

MAE: 0.8142

R2: 0.9644

Ponieważ tracimy dodatkową ważną информацию dla modelu (całą cechę), wydajność modelu spada. Predykcje oraz wyjaśnienie wariancji zmiennej objaśnianej przez zmienne objasniające pogorszyły się, choć niewiele. Jednak, wykres z wpływem cech zmienił się w lepszą stronę. Po usunięciu zmiennej, która dominowała nad pozostałymi wpływ na decyzję stał bardziej rozproszony pomiędzy pozostałymi cechami. Teraz np. cechy `PT08.S5(O3)`, `PT08.S1(CO)` oraz `PT08.S4(NO2)` rodziły między sobą wpływ na decyzję (40%, 27%, 15% odpowiednio). Kosztem jakości predykcji zbudowaliśmy bardziej interpretowalny model, który odpowiedziłby na pytanie: "Jakie cechy najwięcej wpływają na predykcje wartości benzenu?". Jednak nie będziemy używać takiego modelu do tuningu, ponieważ nie jest to głównym celem projektu.

## Hyperparameter tuning

Tuning hyperparametrów polega na znalezieniu optymalnych wartości, które są ustawione przed trenowaniem analitykiem/inżynierem i w żaden sposób nie są zmieniane czy optymalizowane podczas trenowania modelu. Istnieje dwie najpopularniejsze metody poszukiwania takich wartości:

- Grid Search
- Random Search

Z nazw wynika w jaki sposób działają i może dla kogoś wydać się śmieszna druga metoda, bo "jak można losowo wybierając parametry znaleźć najlepszy". Jednak, metoda ta jest bardzo wygodna podczas tuningu złożonych modeli (np. sieci neuronowych), bo trenowanie jednego bazowego modelu może trwać długo, a metoda siatki parametrów (czyli sprawdzanie każdej kombinacji) wydłuża ten czas kilka lub kilkudziesiąt razy. Metoda random search natomiast już kilka lat jest często używana i wiele razy udowodniła swoją przewagę nad Grid Search.

W naszym przypadku robimy tuning modelu `Random Forest`, który został wybrany wcześniej. Ponieważ trenowania bazowego modelu nie zajmuje dużo czasu, wykorzystamy Grid Search, polegającym na utworzeniu siatki parametrów ze wszystkimi kombinacjami

## Optymalizacja Random Forest

In [ ]:

```
from sklearn.model_selection import GridSearchCV

param_grid = {
    'n_estimators': [200, 300, 500],           # Liczba drzew w lesie
    'max_depth': [None, 10, 20],             # maksymalna głębokość drzewa
    'min_samples_split': [2, 5],            # minimalna liczba próbek do podziału węzła
    'min_samples_leaf': [1, 2],              # minimalna liczba próbek w liściu
    'max_features': ['sqrt', 'log2', 1.0],
}

rf = RandomForestRegressor(random_state=42)

grid_search = GridSearchCV(
    param_grid=param_grid,
    estimator=rf,
    cv=5,
    scoring='neg_root_mean_squared_error',
    n_jobs=-1,
)

grid_search.fit(X_train_scaled, y_train)

best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_val_scaled)

mse = mean_squared_error(y_val, y_pred)
print("Najlepsze parametry:", grid_search.best_params_)
print("RMSE na zbiorze walidacyjnym:", np.sqrt(mse))
print("RMSE na zbiorze treningowym:", np.sqrt(mean_squared_error(y_train, best_model.predict(X_1
```

Najlepsze parametry: {'max\_depth': 10, 'max\_features': 1.0, 'min\_samples\_leaf': 2, 'min\_samples\_split': 2, 'n\_estimators': 500}  
RMSE na zbiorze walidacyjnym: 0.08623124179471915  
RMSE na zbiorze treningowym: 0.11464284210695994

Poprzez optymalizację parametrów modelu otrzymaliśmy lepszą predykcję modelu (mniejszy RMSE niż 0.1628 w modelu bazowym). RMSE na zbiorze treningowy stał większy (0.0649 w bazowym), jednak nie jest to żaden problem, nawet odwrotnie taki wynik oznacza mniejsze przeuczenie modelu na zbiorze treningowym ponieważ różnica pomiędzy RMSE walidacyjnym oraz RMSE treningowym zmniejszyła się.

In [ ]:

```
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import Pipeline

poly_pipeline = Pipeline([
    ("poly_features", PolynomialFeatures(degree=3, include_bias=False)),
    ("linear_regression", LinearRegression())
])

poly_pipeline.fit(X_train_scaled, y_train)
y_pred = poly_pipeline.predict(X_val_scaled)
mse = mean_squared_error(y_val, y_pred)

print("RMSE na zbiorze treningowym:", round(np.sqrt(mean_squared_error(y_train, poly_pipeline.pr
best_model = poly_pipeline
```

```
print("\nRMSE na zbiorze walidacyjnym:", round(np.sqrt(mse),4))
print("MAE na zbiorze walidacyjnym:", round(mean_absolute_error(y_val, y_pred),4))
print("R2 na zbiorze walidacyjnym:", round(r2_score(y_val, y_pred),4))
```

RMSE na zbiorze treningowym: 0.0291

RMSE na zbiorze walidacyjnym: 0.0312  
MAE na zbiorze walidacyjnym: 0.0267  
R2 na zbiorze walidacyjnym: 1.0

Mimo tego że udało się uzyskać lepsze wyniki dla modelu Random Forest , model Polynomial Regression jest nadal najlepszym modelem według RMSE zarówno na zbiorze walidacyjnym oraz treningowym (0.0312 oraz 0.0291 odpowiednio).

## Ewaluacja najlepszego modelu na zbiorze testowym

Po wszystkich iteracjach i zmianach w danych lub modelach, można testować najlepszy uzyskany model na zbiorze testowym. Ani model, ani my żadnego razu nie wykorzystaliśmy ten zbiór do modelowania, dlatego jakość predykcji na tym zbiorze pokaże jak model poradzi sobie z nowymi danymi.

```
In [ ]: test_cleaned = pd.read_csv('data/test_cleaned.csv')

X_test = test_cleaned.iloc[:, :-1]
y_test = test_cleaned.iloc[:, -1]

X_test_scaled = pd.DataFrame(scaler.transform(X_test), columns=X_test.columns)
y_pred = best_model.predict(X_test_scaled)

mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print("RMSE na zbiorze testowym:", round(np.sqrt(mse),4))
print("MAE na zbiorze testowym:", round(mae,4))
print("R2 na zbiorze testowym:", round(r2,4))
```

RMSE na zbiorze testowym: 0.0319  
MAE na zbiorze testowym: 0.027  
R2 na zbiorze testowym: 1.0

Porównując uzyskane wyniki z wynikami predykcji na zbiorze walidacyjnymi, możemy zrobić wniosek, że model nie jest przeuczony ani na zbiorze treningowym ani na walidacyjnym. Wartości metryk są podobne dla wszystkich zbiorów.

```
In [ ]: target = pd.concat([y_train, y_val, y_test], axis=0)
target.describe()
```

count	8991.000000
mean	10.083105
std	7.449820
min	0.100000
25%	4.400000
50%	8.200000
75%	14.000000
max	63.700000
Name:	C6H6(GT), dtype: float64

Ponieważ wartości zmiennej objaśnianej C6H6(GT) są w przedziale od 0.1 do 63.7 ze średnią 10.08 oraz odchyleniem standardowym 7.45, wartość RMSE (czyli przeliczonym już na jednostki Y) na zbiorze testowym jest bardzo niska - 0.0319. To oznacza, że zbudowaliśmy model, który może bardzo dokładnie określić wartość benzenu w powietrzu na podstawie wartości pozostałych elementów.

# Eksport modelu

```
In [ ]: import joblib  
joblib.dump(best_model, 'best_model.pkl')  
['best_model.pkl']
```

## Podsumowanie projektu

### 1. Wstępna eksploracja i wizualizacja danych

- *Wstępna eksploracja*
  - *Statystyki opisowe*
  - Obliczyliśmy podstawowe statystyki (średnia, mediana, odchylenie standardowe, min/max, kwartyle) dla każdej zmiennej.
- *Wizualizacje*
  - Zwizualizowaliśmy rozkłady wszystkich zmiennych oraz zależności pomiędzy nimi, by zidentyfikować ewentualne anomalia i zrozumieć kształt danych.

### 2. Czyszczenie i weryfikacja danych

#### 1. Sprawdzanie typów danych

- Zweryfikowaliśmy, że wszystkie kolumny pomiarowe są typu numerycznego.

#### 2. Usunięcie kolumn daty i czasu

- Kolumny Date i Time zostały odrzucone, ponieważ analizujemy dane w ujęciu godzinowym bez uwzględniania sezonowości czasowej na tym etapie.

#### 3. Usuwanie zduplikowanych wierszy

- Wykryliśmy i usunęliśmy duplikaty, aby uniknąć sztucznego zwiększenia liczebności próby.

#### 4. Podział na zbiory

- Dane podzielono w proporcji (np. 80% trening, 10% walidacja, 10% test), z zachowaniem losowości i unikaniem wycieku informacji (data leakage).

#### 5. Wartości brakujące

- Wykonaliśmy dokładną analizę wartości brakujących oraz zastosowaliśmy metodę KNN do wypełnienia niektórych

#### 6. Wartości odstające

- Chociaż zidentyfikowaliśmy obserwacje odstające, zdecydowaliśmy się ich nie usuwać, by nie tracić potencjalnych sygnałów środowiskowych.

#### 7. PCA

- Wykonaliśmy PCA na danych, by zmniejszyć wymiarowość danych, lecz zdecydowaliśmy się nie korzystać z wyników, ponieważ PCA nie przyniosła istotnych korzyści w tym przypadku.

### 3. Selekcja cech i finalny zbiór

- Na podstawie analizy korelacji oraz oceny znaczenia operacyjnego odrzucono zmienną:
  - NMHC(GT)
- Ostateczny zestaw cech wejściowych do modelowania:  
CO(GT), PT08.S1(CO), PT08.S2(NMHC), NOx(GT), PT08.S3(NOx), NO2(GT), PT08.S4(NO2), PT08.S5(O3), T, RH, AH
- Zmienna docelowa: C6H6(GT)

### 4. Standaryzacja cech

- Wszystkie cechy wejściowe zostały przekształcone do postaci standaryzowanej (średnia = 0, odchylenie = 1), aby wyrównać ich zakresy i jednostki.

### 5. Klasteryzacja jakości powietrza

- *Przeprowadziliśmy analizę klastrową za pomocą algorytmu K-średnich*
- *Wykonaliśmy podział na 3 klasy* Dzięki algorytmowi klasteryzacji udało się wyodrębnić trzy naturalne grupy obserwacji:
  1. *Klasa 0 (czerwony)* – zła jakość powietrza (powietrze zanieczyszczone).
  2. *Klasa 1 (niebieski)* – dobra jakość powietrza (niski poziom zanieczyszczania).
  3. *Klasa 2 (zielony)* – jakość średnia (poziom umiarkowany).
- *Liczliwość klastrów*
  - Klasa 1: ponad 3 500 obserwacji
  - Klasa 2: około 3 200 obserwacji
  - Klasa 0: około 2 100 obserwacji
- *Rozkład pozostałych cech w klastrach* Przyjrzenie się wykresom rozkładu kilku zmiennych (CO(GT), C6H6(GT), NOx(GT), NO2(GT)) w podziale na klastry potwierdziło:
  - Dla *Klasy 0* wartości wszystkich cech są najwyższe (zarówno mediana, jak i zakres).
  - *Klasy 1 i 2* mają wartości zbliżone do siebie; w przypadku zmiennych NOx i NO2 różnice między nimi są wręcz minimalne.

To uzupełnienie podkreśla, że dane dają się klarownie podzielić na trzy grupy odpowiadające zróżnicowanej jakości powietrza, co może być podstawą np. do segmentacji obszarów czy prognozowania poziomów zagrożeń.

## 6. Modele regresyjne

Przygotowaliśmy i porównaliśmy wstępne wersje następujących algorytmów:

- *Random Forest Regressor*
- *XGBoost Regressor*
- *CatBoost Regressor*
- *Decision Tree Regressor*
- *Linear Regression*
- *Polynomial Regression*
- *Lasso Regression*
- *Ridge Regression*

Każdy model był trenowany na zbiorze treningowym, bez optymalizacji hiperparametrów na zbiorze walidacyjnym, a ocena wykonana na zbiorze walidacyjnym przy użyciu metryk RMSE, MAE i  $R^2$ .

### Podsumowanie wyboru modeli

- Główne kryterium: minimalizacja RMSE na zbiorze walidacyjnym.

Wybieramy *Polynomial Regression (deg=3)* oraz *Random Forest* jako modele do dalszej analizy, ponieważ:

Model	RMSE Train	MAE Train	$R^2$ Train	RMSE Val	MAE Val	$R^2$ Val	*Overfitting**	**Underfitting*
Polynomial Regression (deg=3)	0.0291	0.0245	1.0000	0.0312	0.0267	1.0000	0.0021	0.0000
Random Forest	0.0649	0.0057	0.9999	0.1628	0.0201	0.9995	0.0979	0.0000

- *Polynomial Regression (deg=3)*

- Osiąga najniższy RMSE Val (0.0312) oraz najwyższy  $R^2$  Val (1.0000) na zbiorze testowym i trenigowym.
- Nie wykazuje znaczącego nadmiaru lub braku dopasowującego się do danych.
- Nie posiada hiperparametrów do dalszej walidacji modelu.

- *Random Forest*

- Zachowuje wysoką dokładność ( $R^2$  Val = 0.9995),
- Małe RMSE Val (0.1628) oraz MAE Val (0.0201) na zbiorze testowym.
- Minimalizuje ryzyko overfittingu,
- Gwarantuje odpowiednią generalizację.
- Posiada hiperparametry do dalszej walidacji modelu.

## 7. Analiza ważności

- Przeprowadziliśmy analizę ważności w celu interpretacji uzyskanego modelu oraz oceny wpływu predyktorów na predykcję
- Porównaliśmy kilka modeli (po usuwaniu wybranych zmiennych) w celu zlikwidowania dominacji wpływu jednej zmiennej

## 8. Optymalizacja hiperparametrów

- Wykonaliśmy tuning hiperparametrów modelu *Random Forest* w celu poprawy jakości predykcji
- Znalezliśmy odpowiednie parametry za pomocą metody *Grid Search*, jednak wybraliśmy model *Polynomial Regression* jako ostateczny

## 9. Ewaluacja modelu na danych testowych

- Dokonaliśmy predykcję za pomocą najlepszego modelu na danych ze zbioru testowego w celu unikania przeuczenia pod czas iterowanego poprawiania metod przetwarzania danych lub optymalizacji hiperparametrów modelu
- Porównaliśmy wartości wybranych metryk (RMSE, MAE,  $R^2$ ) z rozkładem zmiennej objaśnianej

## 10. Eksport modelu

- Wyeksportowaliśmy model jako plik .pkl, aby móc go ponownie wykorzystać – np. w aplikacji internetowej jako część back-endu – bez potrzeby ponownego trenowania.