“Air Quality Analysis”

Topic Selection: The topic of air quality analysis remains highly relevant due to human health issues associated with air pollution, as well as its impact on climate change. Atmospheric pollution can lead to serious health conditions and has long-term consequences for the environment and ecosystems. This topic is crucial for scientific research, the development of effective control measures, and the promotion of public health.

Domain Description: This domain involves the study of atmospheric air composition and the detection and quantification of various pollutants. Tasks in this area include monitoring, analyzing, and evaluating the impact of pollution on human health and the environment. With the growth of industrial activity and urban development, air quality analysis has become an essential tool for implementing effective measures to mitigate negative effects on both health and the climate.

Dataset Description: In this study, we will work with an air quality dataset consisting of approximately 9,358 instances of hourly averaged responses from an array of five metal oxide chemical sensors embedded in a multisensor device designed for air quality monitoring. The sensors measure concentrations of carbon monoxide, non-methane hydrocarbons, benzene, NOx, and NO_2. The dataset also includes climate-related measurements such as temperature and relative humidity, along with sensor responses. Based on this data, we aim to predict the hourly average concentration of benzene (C_6H_6(GT)).

Objective of the Study: To develop a regression model for predicting the concentration of benzene in the air and to analyze the resulting performance.

Conducted by Maksym Alieksieiev
April 2025

Workflow:

Before building the regression models, we begin by importing the necessary libraries. We use NumPy and pandas for data manipulation, as well as Matplotlib and Seaborn for data visualization.

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

1. Data Loading and Exploratory Data Analysis (EDA)

1.1. Data Loading and Data Type Handling

First, we load the dataset from an Excel spreadsheet and perform a brief preliminary inspection of the data types.

```
data = pd.read_excel('AirQualityUCI.xlsx')
data.head().T
```

	0	1	2	3	4
Date	2004-03-10 00:00:00	2004-03-10 00:00:00	2004-03-10 00:00:00	2004-03-10 00:00:00	2004-03-10 00:00:00
Time	18:00:00	19:00:00	20:00:00	21:00:00	22:00:00
CO(GT)	2.6	2.0	2.2	2.2	1.6
PT08.S1(CO)	1360.0	1292.25	1402.0	1375.5	1272.25
NMHC(GT)	150	112	88	80	51
C6H6(GT)	11.881723	9.397165	8.997817	9.228796	6.518224
PT08.S2(NMHC)	1045.5	954.75	939.25	948.25	835.5
NOx(GT)	166.0	103.0	131.0	172.0	131.0
PT08.S3(NOx)	1056.25	1173.75	1140.0	1092.0	1205.0
NO2(GT)	113.0	92.0	114.0	122.0	116.0
PT08.S4(NO2)	1692.0	1558.75	1554.5	1583.75	1490.0
PT08.S5(O3)	1267.5	972.25	1074.0	1203.25	1110.0
T	13.6	13.3	11.9	11.0	11.15
RH	48.875001	47.7	53.975	60.0	59.575001
AH	0.757754	0.725487	0.750239	0.786713	0.788794

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9357 entries, 0 to 9356
Data columns (total 15 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Date           9357 non-null   datetime64[ns]
 1   Time           9357 non-null   object
 2   CO(GT)         9357 non-null   float64
 3   PT08.S1(CO)    9357 non-null   float64
 4   NMHC(GT)       9357 non-null   int64
 5   C6H6(GT)       9357 non-null   float64
 6   PT08.S2(NMHC)  9357 non-null   float64
 7   NOx(GT)        9357 non-null   float64
 8   PT08.S3(NOx)   9357 non-null   float64
 9   NO2(GT)        9357 non-null   float64
 10  PT08.S4(NO2)   9357 non-null   float64
 11  PT08.S5(O3)    9357 non-null   float64
 12  T              9357 non-null   float64
 13  RH             9357 non-null   float64
 14  AH             9357 non-null   float64
dtypes: datetime64[ns](1), float64(12), int64(1), object(1)
memory usage: 1.1+ MB
```

We can observe that the *Time* column is of type 'object', but upon closer inspection, it actually represents time values of the 'datetime.time' type.

```
data.loc[0, 'Time']
```

```
datetime.time(18, 0)
```

However, since the measurements in the dataset are recorded on an hourly basis, we only need the hour values. Therefore, we will transform the column accordingly.

```
data['Time'] = data['Time'].apply(lambda item: item.hour)
```

As for the *Date* column, we will split it into three separate columns containing the day, month, and year, respectively. This will allow us to examine whether variations in the target variable depend on temporal factors.

```
data['Day'] = data['Date'].dt.day
data['Month'] = data['Date'].dt.month
data['Year'] = data['Date'].dt.year
data.drop('Date', axis=1, inplace=True)
```

Let us examine the data types to ensure that all types are correctly assigned.

```
data.dtypes
```

```
Time              int64
CO(GT)          float64
PT08.S1(CO)     float64
NMHC(GT)          int64
C6H6(GT)        float64
PT08.S2(NMHC)   float64
NOx(GT)         float64
PT08.S3(NOx)    float64
NO2(GT)         float64
PT08.S4(NO2)    float64
PT08.S5(O3)     float64
T               float64
RH              float64
AH              float64
Day               int64
Month             int64
Year              int64
dtype: object
```

1.2. Handling Missing Values

We now proceed with the exploratory data analysis (EDA). First, we will perform statistical summary analysis of all columns (we consider all columns at once, as they are all numeric).

```
data.describe().T
```

	count	mean	std	min	25%	50%	75%
Time	9357.0	11.498557	6.923182	0.0	5.000000	11.000000	18.000000
CO(GT)	9357.0	-34.207524	77.657170	-200.0	0.600000	1.500000	2.600000
PT08.S1(CO)	9357.0	1048.869652	329.817015	-200.0	921.000000	1052.500000	1221.250000
NMHC(GT)	9357.0	-159.090093	139.789093	-200.0	-200.000000	-200.000000	-200.000000
C6H6(GT)	9357.0	1.865576	41.380154	-200.0	4.004958	7.886653	13.636091
PT08.S2(NMHC)	9357.0	894.475963	342.315902	-200.0	711.000000	894.500000	1104.750000
NOx(GT)	9357.0	168.604200	257.424561	-200.0	50.000000	141.000000	284.200000
PT08.S3(NOx)	9357.0	794.872333	321.977031	-200.0	637.000000	794.250000	960.250000
NO2(GT)	9357.0	58.135898	126.931428	-200.0	53.000000	96.000000	133.000000
PT08.S4(NO2)	9357.0	1391.363266	467.192382	-200.0	1184.750000	1445.500000	1662.000000
PT08.S5(O3)	9357.0	974.951534	456.922728	-200.0	699.750000	942.000000	1255.250000
T	9357.0	9.776600	43.203438	-200.0	10.950000	17.200000	24.075000
RH	9357.0	39.483611	51.215645	-200.0	34.050000	48.550000	61.875000
AH	9357.0	-6.837604	38.976670	-200.0	0.692275	0.976823	1.296223
Day	9357.0	15.876884	8.808653	1.0	8.000000	16.000000	23.000000
Month	9357.0	6.310356	3.438160	1.0	3.000000	6.000000	9.000000
Year	9357.0	2004.240141	0.427192	2004.0	2004.000000	2004.000000	2004.000000

We observe that none of the columns contain NaN values; however, most columns have a minimum value of -200, which, according to the dataset documentation, indicates missing data. Therefore, we will replace -200 with NaN to accurately assess the extent of missing values in the dataset.

```
data.replace(-200, np.nan, inplace=True)

def count_null(data):
    len_data = len(data)
    null_count = pd.DataFrame(data.isna().sum(), columns=['Count'])
    null_count['Percentage'] = round(null_count['Count'] / len_data * 100, 2)
    return null_count

count_null(data)
```

	Count	Percentage
Time	0	0.00
CO(GT)	1683	17.99
PT08.S1(CO)	366	3.91
NMHC(GT)	8443	90.23
C6H6(GT)	366	3.91
PT08.S2(NMHC)	366	3.91
NOx(GT)	1639	17.52
PT08.S3(NOx)	366	3.91
NO2(GT)	1642	17.55
PT08.S4(NO2)	366	3.91
PT08.S5(O3)	366	3.91
T	366	3.91
RH	366	3.91
AH	366	3.91
Day	0	0.00
Month	0	0.00
Year	0	0.00

First of all, we observe that the *NMHC(GT)* column contains an excessive amount of missing values (90%), and therefore, we remove this entire column from the dataset.

```
data.drop('NMHC(GT)', axis=1, inplace=True)
```

Regarding the missing values in the target variable (*C6H6(GT)*), they account for only 4%. Therefore, we will remove every record with a missing target value and then verify how many (and where) missing values remain in the dataset.

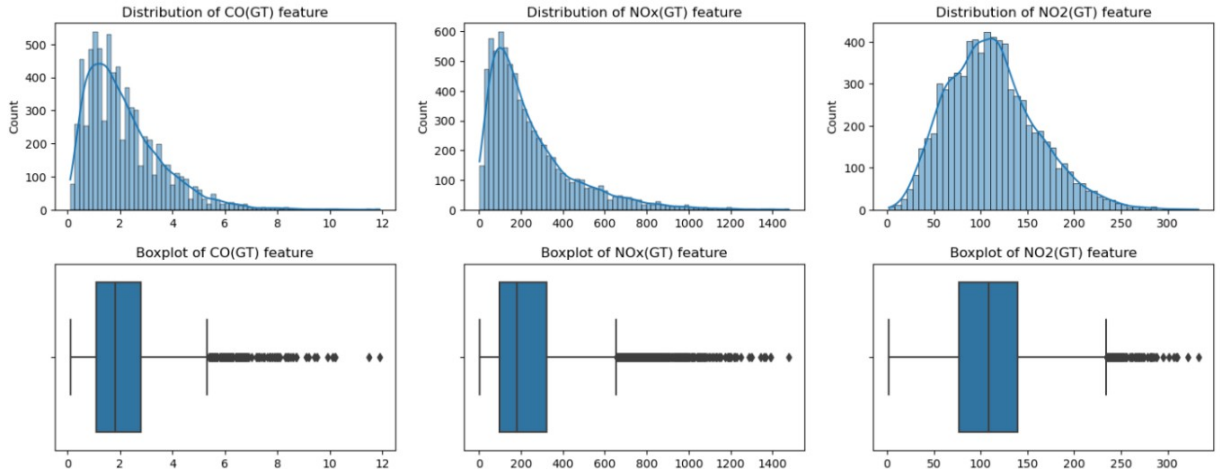
```
data.dropna(subset='C6H6(GT)', inplace=True)
count_null(data)
```

	Count	Percentage
Time	0	0.00
CO(GT)	1647	18.32
PT08.S1(CO)	0	0.00
C6H6(GT)	0	0.00
PT08.S2(NMHC)	0	0.00
NOx(GT)	1595	17.74
PT08.S3(NOx)	0	0.00
NO2(GT)	1598	17.77
PT08.S4(NO2)	0	0.00
PT08.S5(O3)	0	0.00
T	0	0.00
RH	0	0.00
AH	0	0.00
Day	0	0.00
Month	0	0.00
Year	0	0.00

We observe that the only columns containing missing values are *CO(GT)*, *NOx(GT)*, and *NO2(GT)*. However, removing the corresponding samples is not feasible due to their large quantity. Therefore, we will attempt to impute the missing values using three different methods and evaluate which approach performs best.

```
data_copy = data.copy()
```

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

def make_hist_boxplots(data, cols):
    fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(15, 6))
    axes = axes.flatten()
    for i, col in enumerate(cols):
        sns.histplot(data=data, x=col, kde=True, ax=axes[i])
        axes[i].set_title(f'Distribution of {col} feature')
        axes[i].set_xlabel('')
    for i, col in enumerate(cols):
        sns.boxplot(data=data, x=col, ax=axes[i+3])
        axes[i+3].set_title(f'Boxplot of {col} feature')
        axes[i+3].set_xlabel('')
    fig.tight_layout(pad=1.5)
    plt.show()

make_hist_boxplots(data_copy, cols_with_missing)
```

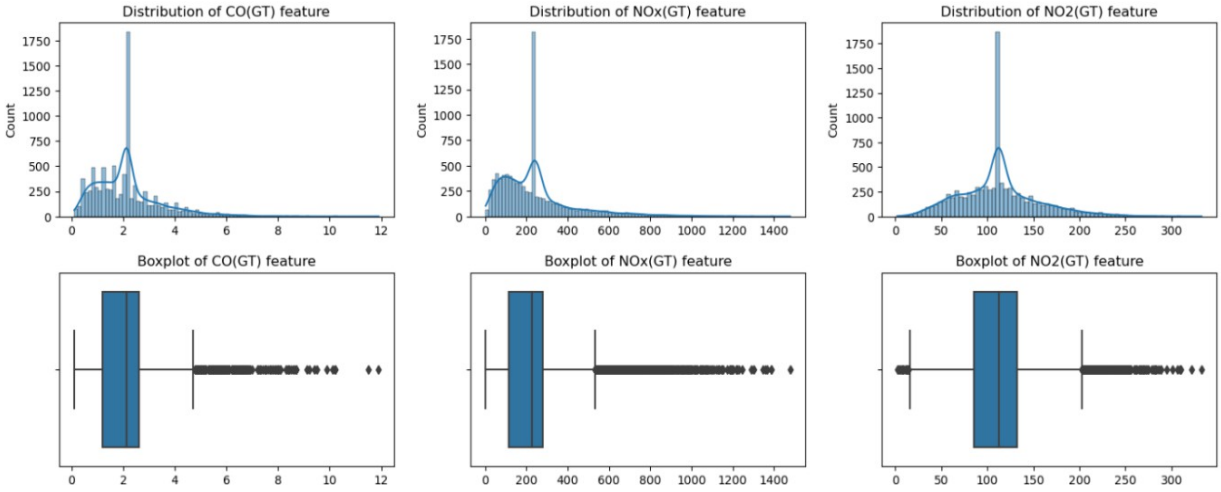


We observe that all three features exhibit a normal distribution (despite the presence of outliers). Therefore, after imputation, we aim to preserve statistical characteristics of these distributions, such as the median and the outliers.

1.2.1. Imputation Using the Mean Value

The first method we will use is the most straightforward — replacing missing values with the mean. We will apply this to a copied version of the dataset and then examine the resulting distribution.

```
data_missing_mean = data_copy.copy()
for col in cols_with_missing:
    data_missing_mean[col] = data_missing_mean[col].fillna(data_missing_mean[col].mean())
make_hist_boxplots(data_missing_mean, cols_with_missing)
```

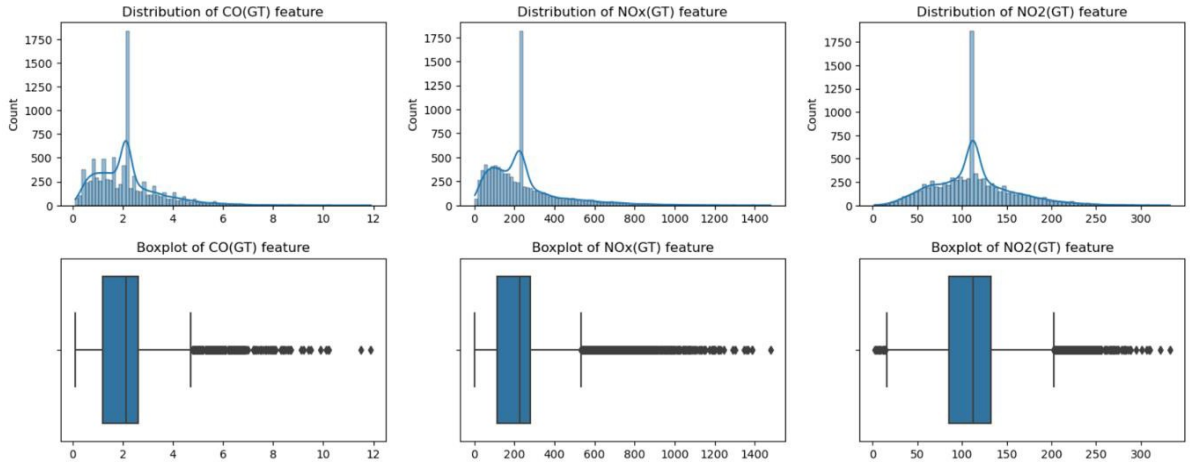


We observe that the distribution has changed significantly (mainly because the “mode density” is now different). Additionally, the median and outliers have shifted. Thus, although this method is simple, it proves to be ineffective.

1.2.2. Imputation Using the Median

Next, we will impute missing values using the median, as the median is more robust to the presence of outliers.

```
data_missing_median = data_copy.copy()
for col in cols_with_missing:
    data_missing_median[col] = data_missing_median[col].fillna(data_missing_mean[col].median())
make_hist_boxplots(data_missing_median, cols_with_missing)
```

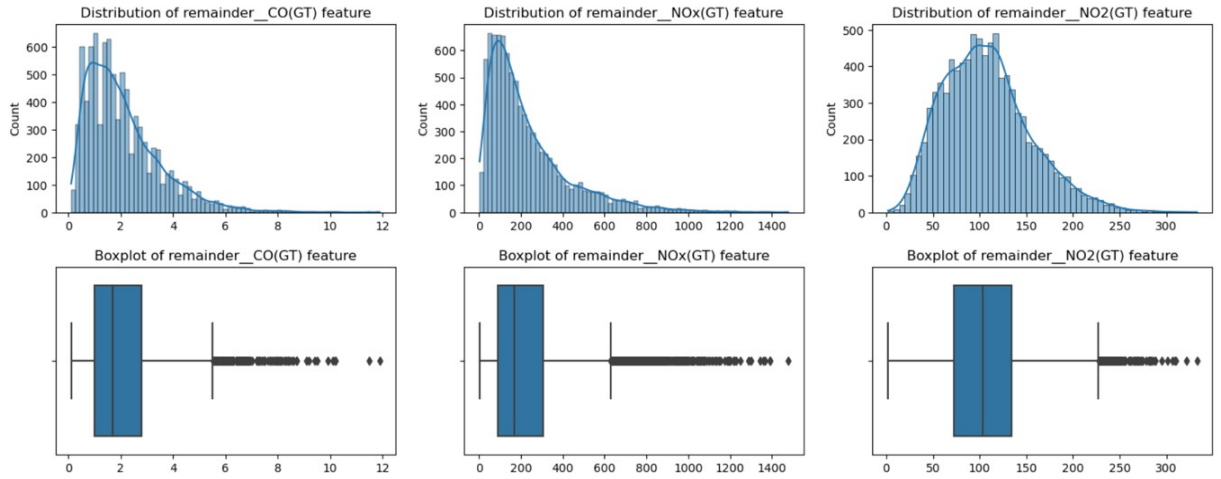


We observe that even with median imputation, many statistical measures have changed. This is likely due to the large proportion of missing values.

1.2.3. Imputation Using k-Nearest Neighbors (kNN) Imputer

Finally, we will apply the *KNNImputer* to estimate missing values based on the nearest neighbors. However, prior to this, we need to rescale the other variables to correctly identify the nearest points. For this purpose, we will use the *RobustScaler*, as other variables may contain outliers.

```
from sklearn.impute import KNNImputer
from sklearn.preprocessing import RobustScaler
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline

cols_without_missing = data_copy.drop(cols_with_missing, axis=1).columns.to_list()
knn_imputer = Pipeline([
    ('scaler', ColumnTransformer([
        ('scaler', RobustScaler(), cols_without_missing)
    ], remainder='passthrough')),
    ('knn_imputer', KNNImputer())
])
data_missing_knn = knn_imputer.fit_transform(data_copy)
data_missing_knn = pd.DataFrame(data_missing_knn, columns=knn_imputer.named_steps['scaler'].get_feature_names_out())

make_hist_boxplots(data_missing_knn, ['remainder__CO(GT)', 'remainder__NOx(GT)', 'remainder__NO2(GT)'])
```



We observe that this method preserved the distribution, median, and outliers; therefore, we will use it for imputing the missing values. Finally, we will verify that no missing values remain.

```
data_copy.reset_index(drop=True, inplace=True)
data_missing_knn.reset_index(drop=True, inplace=True)

for col in cols_with_missing:
    data_copy[col] = data_missing_knn[f'remainder__{col}']

count_null(data_copy)
```

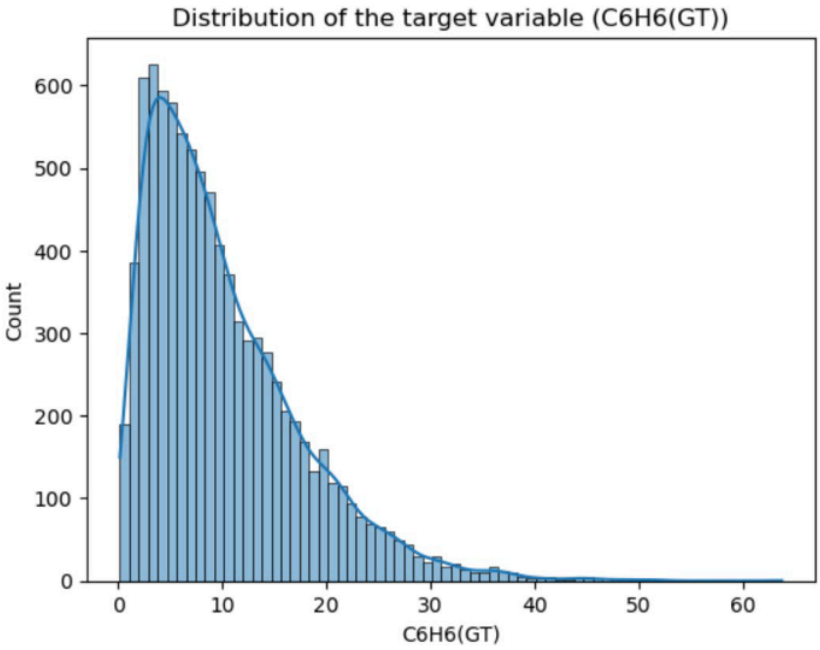
	Count	Percentage
Time	0	0.0
CO(GT)	0	0.0
PT08.S1(CO)	0	0.0
C6H6(GT)	0	0.0
PT08.S2(NMHC)	0	0.0
NOx(GT)	0	0.0
PT08.S3(NOx)	0	0.0
NO2(GT)	0	0.0
PT08.S4(NO2)	0	0.0
PT08.S5(O3)	0	0.0
T	0	0.0
RH	0	0.0
AH	0	0.0
Day	0	0.0
Month	0	0.0
Year	0	0.0

No missing values remain, so we proceed to univariate analysis.

1.3. Univariate Analysis of the Target Variable

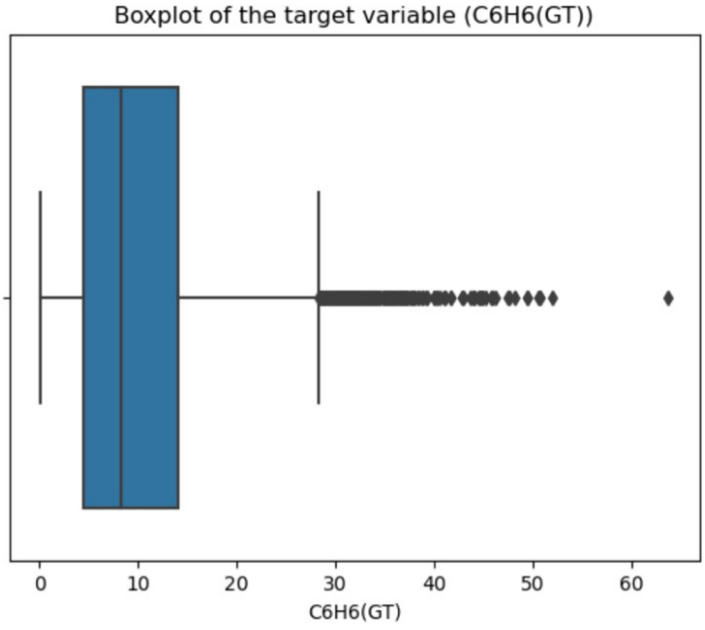
Let us begin by examining the distribution of the target variable

```
sns.histplot(data=data_copy, x='C6H6(GT)', kde=True)
plt.title('Distribution of the target variable (C6H6(GT))')
plt.show()
```



We observe that the target variable follows a normal distribution (which is favorable, as it satisfies one of the assumptions of linear regression), although it contains several outliers.

To further investigate the outlier issue, we will plot a graph.



We confirm that the assumption regarding outliers is correct. Let us now determine the actual number of outliers present in our case.

```
def calculate_limits(ser):
    q1 = ser.quantile(0.25)
    q3 = ser.quantile(0.75)
    iqr = q3 - q1
    return (q1 - 1.5 * iqr, q3 + 1.5 * iqr)

def outliers_mask(ser):
    min_lim, max_lim = calculate_limits(ser)
    return (min_lim <= ser) & (ser <= max_lim)

def count_outliers(ser):
    mask = ~outliers_mask(ser)
    return mask.sum()

target_outliers = count_outliers(data_copy['C6H6(GT)'])
print(f'There are {target_outliers} ({round(target_outliers / len(data_copy) * 100, 2)}%) outliers')
```

```
There are 230 (2.56%) outliers
```

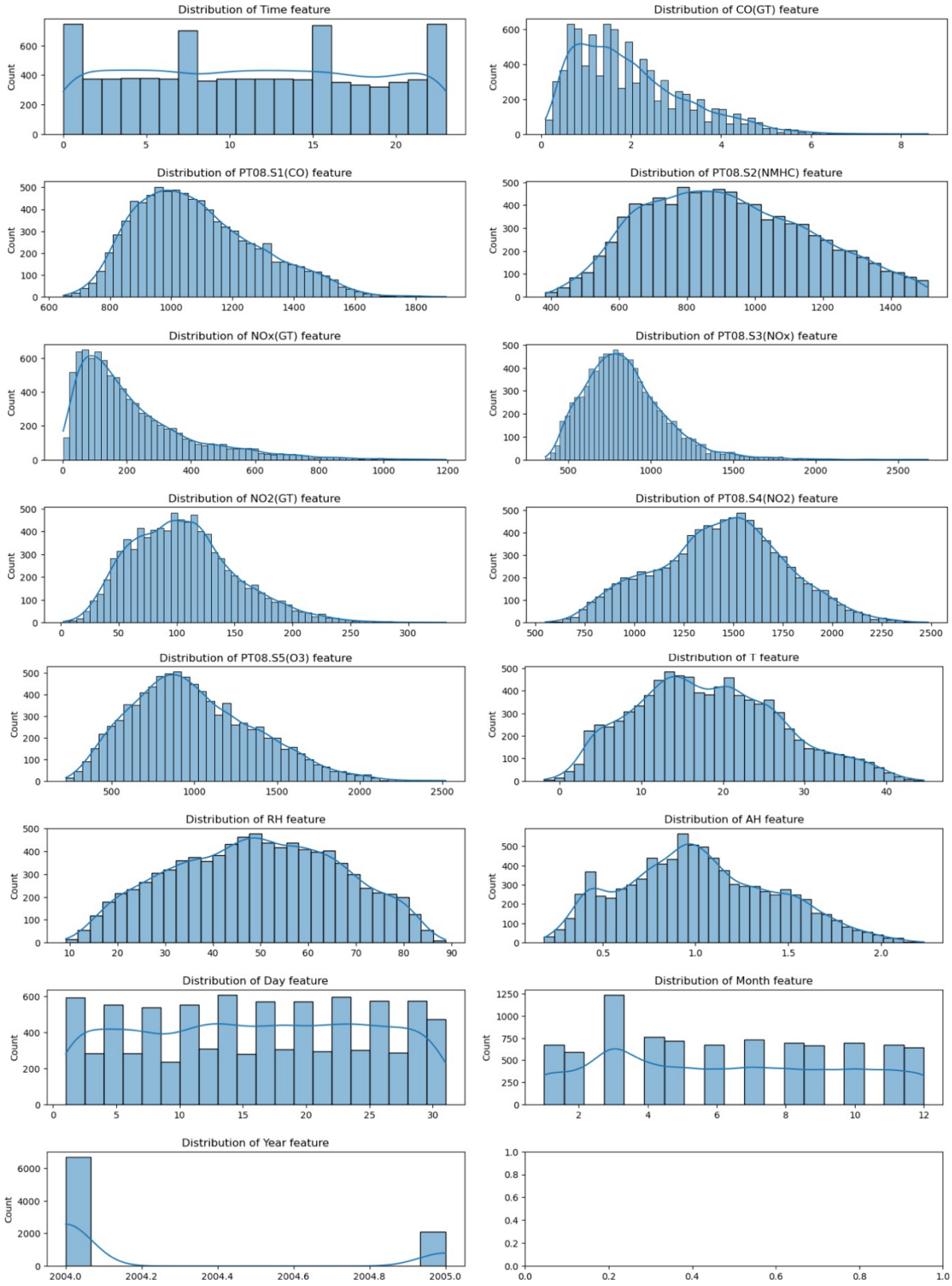
The number of outliers is very small, and since outliers can pose challenges in building linear models, we may remove them.

```
data_copy = data_copy[outliers_mask(data_copy['C6H6(GT)'])]
```

1.4. Univariate Analysis of Other Variables

Let us plot the distribution graphs of the other variables.

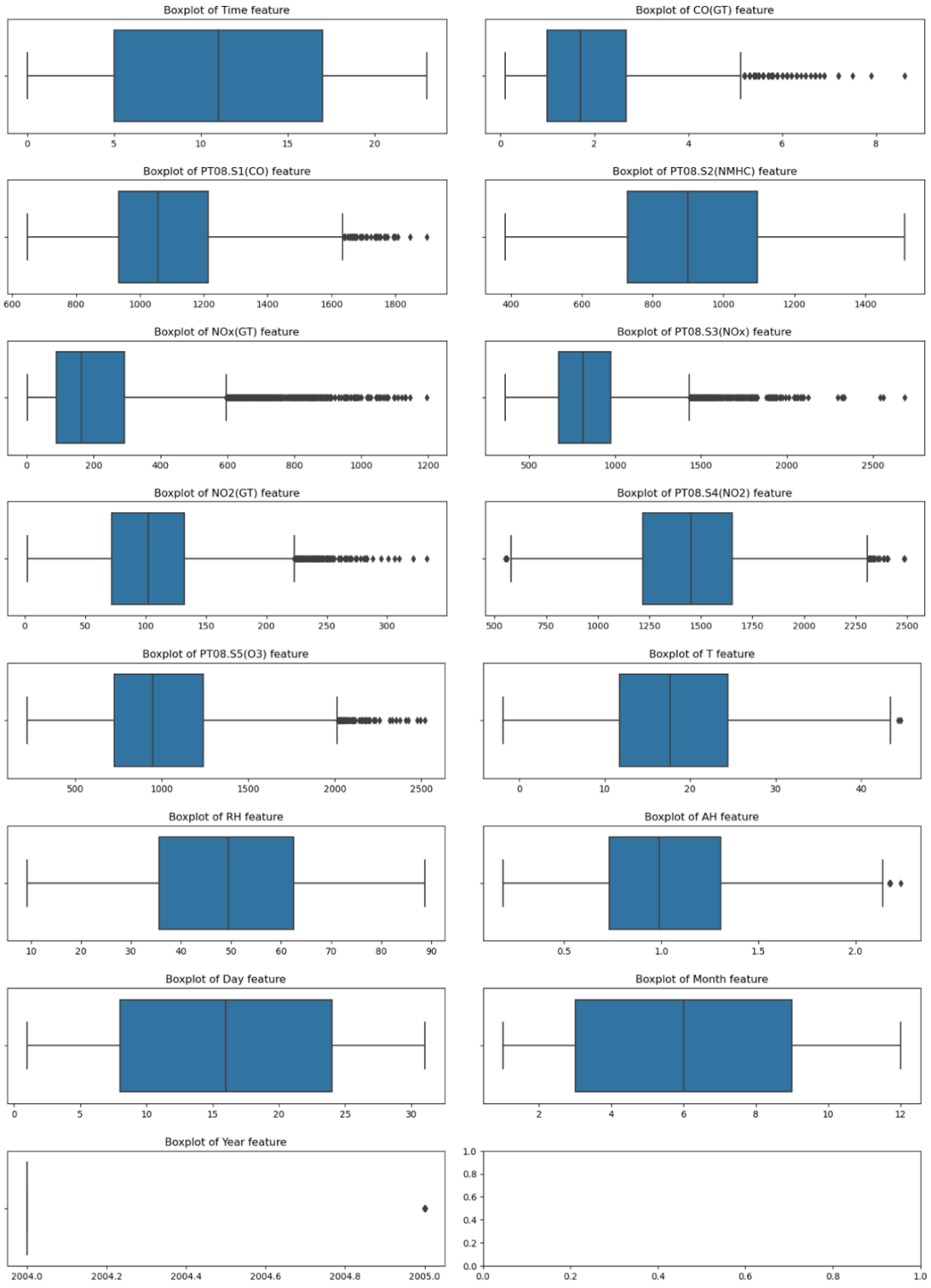
```
fig, axes = plt.subplots(nrows=8, ncols=2, figsize=(15, 20))
axes = axes.flatten()

cols = data_copy.drop('C6H6(GT)', axis=1).columns.to_list()

for i, col in enumerate(cols):
    sns.histplot(data=data_copy, x=col, kde=True, ax=axes[i])
    axes[i].set_title(f'Distribution of {col} feature')
    axes[i].set_xlabel('')

plt.tight_layout(pad=2)
plt.show()
```



We observe that most features follow a normal distribution (with the exception of date and time, which mostly exhibit a uniform distribution).

Now, let us examine the number of outliers.

```
fig, axes = plt.subplots(nrows=8, ncols=2, figsize=(15, 20))
axes = axes.flatten()

for i, col in enumerate(cols):
    sns.boxplot(data=data_copy, x=col, ax=axes[i])
    axes[i].set_title(f'Boxplot of {col} feature')
    axes[i].set_xlabel('')

plt.tight_layout(pad=2)
plt.show()
```



```
len_data_copy = len(data_copy)


for col in cols:
    target_outliers = count_outliers(data_copy[col])
    print(f'For feature {col}: {target_outliers} ({round(target_outliers / len_data_copy * 100, 2)}%) outliers')
```

```
For feature Time: 0 (0.0%) outliers
For feature CO(GT): 131 (1.5%) outliers
For feature PT08.S1(CO): 44 (0.5%) outliers
For feature PT08.S2(NMHC): 0 (0.0%) outliers
For feature NOx(GT): 435 (4.97%) outliers
For feature PT08.S3(NOx): 247 (2.82%) outliers
For feature NO2(GT): 134 (1.53%) outliers
For feature PT08.S4(NO2): 22 (0.25%) outliers
For feature PT08.S5(O3): 70 (0.8%) outliers
For feature T: 2 (0.02%) outliers
For feature RH: 0 (0.0%) outliers
For feature AH: 4 (0.05%) outliers
For feature Day: 0 (0.0%) outliers
For feature Month: 0 (0.0%) outliers
For feature Year: 2086 (23.81%) outliers
```

It appears that only the "Year" column contains a significant number of outliers, while the others have very few.

However, we need to verify how many records actually contain outliers.

```
mask_final = pd.Series([True] * len_data_copy)
cols_without_year = cols.copy()
cols_without_year.remove('Year')

for col in cols_without_year:
    mask_final = mask_final & outliers_mask(data_copy[col])

samples_outliers = (~mask_final).sum()

print(f'There are {samples_outliers} ({round(samples_outliers / len_data_copy * 100, 2)}%) samples with outliers')
```

```
There are 1272 (14.52%) samples with outliers
```

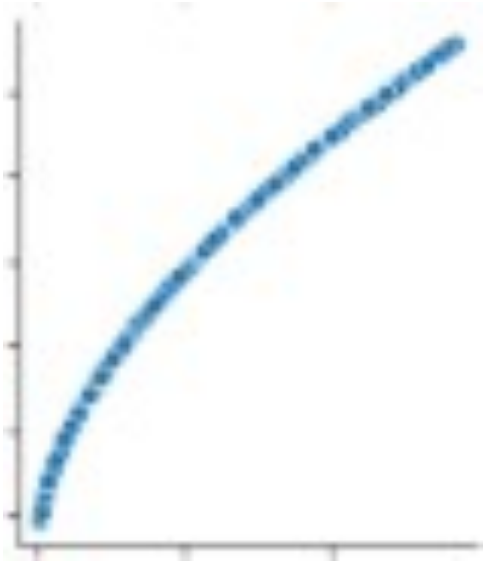
In fact, there are many outliers, so we cannot simply remove them. However, they may pose challenges during scaling (to address this, we will use RobustScaler) and when fitting linear models.

1.5. Multivariate Analysis

We begin the multivariate analysis by creating a pair plot.

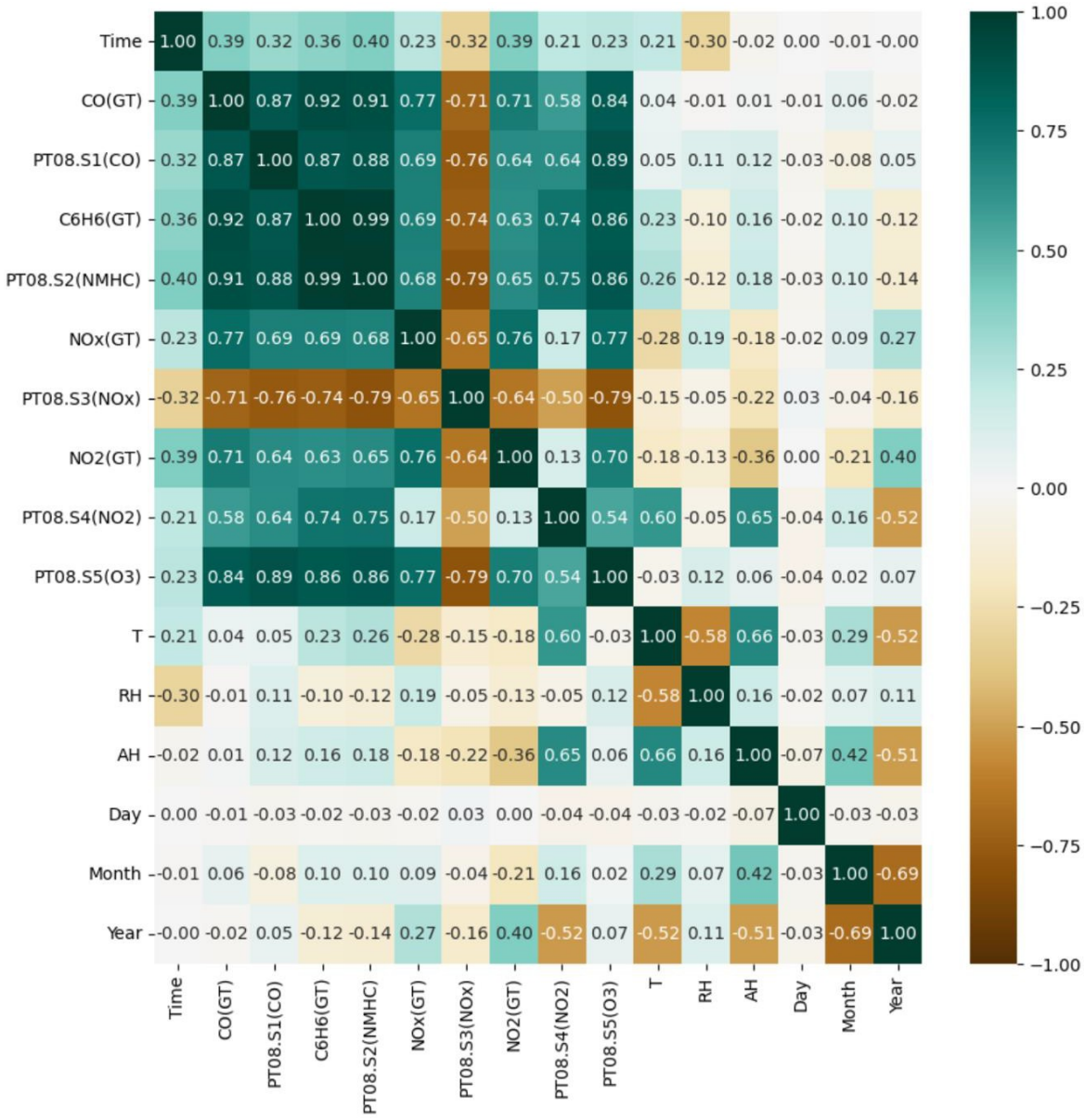
```
sns.pairplot(data=data_copy)
plt.show()
```

The first observation is that there is a very strong correlation between the target variable and the feature PT08.S2(NMHC), as evidenced by the monotonic trend line in one of the scatter plots.



Let us verify this assumption by constructing the Pearson correlation matrix.

```
plt.subplots(figsize=(10, 10))
sns.heatmap(data_copy.corr(), annot=True, vmin=-1, vmax=1,  fmt='.2f', cmap='BrBG')
plt.show()
```



Since the Pearson correlation between C6H6(GT) and PT08.S2(NMHC) is almost equal to one, there is indeed a (nonlinear) correlation between these two variables.

But what is the nature of this correlation? To try to explain this, we need to examine what data this feature represents.

C6H6(GT) (the target variable) contains data on the true average hourly concentration of benzene, whereas PT08.S2(NMHC) stores data on the average hourly sensor response of a titanium oxide sensor.

According to information found online, the sensor response time for gases is the time required for the sensor output signal to change from the previous state to a final steady-state value and is calculated by a specific formula.
Although the proof of my next assumption was not fully verified by the search, it seems that PT08.S2(NMHC) might be calculated after measuring the benzene concentration. If this is the case, we cannot use this feature in regression models, since regression models aim to predict the target variable, and under our assumption, it would be impossible to know the value of PT08.S2(NMHC) before the model predicts the target variable.
Therefore, in future modeling, we will exclude the PT08.S2(NMHC) feature.
Regarding the other features, it appears that they are also correlated, which may pose a problem because linear models assume no multicollinearity among independent variables.
Hence, we need to keep this in mind when building models.

Key conclusions:

- Missing values should be imputed using the k-NN imputer.
- The target variable contains some outliers that can be removed.
- Other variables have many records with outliers.
- The target variable and the feature PT08.S2(NMHC) exhibit suspiciously strong correlation and should likely be excluded from the final model.

2. Model Building

2.1. Preprocessing Steps Before Model Creation

Before we begin building the first model, we need to split the dataset into training (for training our models), validation (for evaluating model performance during development), and test sets (for obtaining the final model accuracy results).

After that, we will remove outliers from the training set to prevent the models from learning on these anomalous data points.

```
from sklearn.model_selection import train_test_split

X = data.drop('C6H6(GT)', axis=1)
y = data['C6H6(GT)']

X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.2, random_state=111)
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.2, random_state=111)

mask = outliers_mask(y_train)
X_train = X_train[mask]
y_train = y_train[mask]
```

2.2. Linear Regression

First, we will create the simplest model: linear regression. However, before we can do this, we need to scale all feature values (we will use RobustScaler since some features contain outliers) and impute missing values (using kNN imputation).

The reason we perform imputation after scaling is that the kNN imputer requires already scaled features, and RobustScaler can handle missing values.

We apply these preprocessing steps only on the training data to avoid data leakage from the validation and test sets during model training.

```
X_train_rest = X_train.drop('PT08.S2(NMHC)', axis=1)
X_val_rest = X_val.drop('PT08.S2(NMHC)', axis=1)
```

```
preprocessing = Pipeline([
    ('scaler', RobustScaler()),
    ('imputer', KNNImputer())
])

X_train_rest_ready = pd.DataFrame(preprocessing.fit_transform(X_train_rest), columns=X_train_rest.columns, index=X_train_rest.index)
X_val_rest_ready = pd.DataFrame(preprocessing.transform(X_val_rest), columns=X_val_rest.columns, index=X_val_rest.index)
```

Now, we are ready to build the model.

```
multiple_linear = sm.OLS(y_train, X_train_rest_ready).fit()

multiple_linear.summary()
```

OLS Regression Results

Dep. Variable:	C6H6(GT)	**R-squared (uncentered):**	0.885
Model:	OLS	**Adj. R-squared (uncentered):**	0.885
Method:	Least Squares	**F-statistic:**	3068.
Date:	Tue, 05 Dec 2023	**Prob (F-statistic):**	0.00
Time:	15:16:59	**Log-Likelihood:**	-15481.
No. Observations:	5597	**AIC:**	3.099e+04
Df Residuals:	5583	**BIC:**	3.108e+04
Df Model:	14		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Time	0.7403	0.108	6.825	0.000	0.528	0.953
CO(GT)	2.0308	0.229	8.883	0.000	1.583	2.479
PT08.S1(CO)	1.4707	0.211	6.963	0.000	1.057	1.885
NOx(GT)	-0.1511	0.156	-0.967	0.334	-0.458	0.155
PT08.S3(NOx)	6.5955	0.117	56.162	0.000	6.365	6.826
NO2(GT)	-1.9631	0.156	-12.604	0.000	-2.268	-1.658
PT08.S4(NO2)	16.1556	0.283	57.086	0.000	15.601	16.710
PT08.S5(O3)	2.3254	0.201	11.574	0.000	1.932	2.719
T	-0.3781	0.285	-1.327	0.185	-0.937	0.181
RH	-2.5500	0.227	-11.221	0.000	-2.995	-2.104
AH	-4.6553	0.242	-19.274	0.000	-5.129	-4.182
Day	1.2955	0.094	13.779	0.000	1.111	1.480
Month	15.1001	0.208	72.621	0.000	14.692	15.508
Year	27.9708	0.261	107.335	0.000	27.460	28.482

Omnibus:	1679.047	**Durbin-Watson:**	1.572
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	9527.755
Skew:	-1.316	**Prob(JB):**	0.00
Kurtosis:	8.825	**Cond. No.**	14.8

From the model fitting results, we observe that the p-values for the features NOx(GT) and T are quite high (greater than 0.05), which may indicate multicollinearity with other variables and suggest that these features should likely be removed.

However, before dropping these columns, we will calculate the evaluation metrics on the validation set.

```
print_metrics(y_val, multiple_linear.predict(X_val_rest_ready))
```

```
R squarred: 0.6502735600950318
MAE: 3.2442003526469145
RMSE: 4.354190623934889
```

Now, we will sequentially remove these features, starting with the NOx(GT) column, and retrain the model.

```
X_train_rest.drop('NOx(GT)', axis=1, inplace=True)
X_train_rest_ready.drop('NOx(GT)', axis=1, inplace=True)
X_val_rest.drop('NOx(GT)', axis=1, inplace=True)
X_val_rest_ready.drop('NOx(GT)', axis=1, inplace=True)
```

```
multiple_linear = sm.OLS(y_train, X_train_rest_ready).fit()

multiple_linear.summary()
```

```
print('For train set')
print_metrics(y_train, multiple_linear.predict(X_train_rest_ready))
print('\nFor validate set')
print_metrics(y_val, multiple_linear.predict(X_val_rest_ready))
```

```
For train set
R squarred: 0.6243640799572074
MAE: 3.06626956736577
RMSE: 3.8464748258448473

For validate set
R squarred: 0.650423859578805
MAE: 3.244674872325823
RMSE: 4.353254888376766
```

OLS Regression Results

Dep. Variable:	C6H6(GT)	**R-squared (uncentered):**	0.885
Model:	OLS	**Adj. R-squared (uncentered):**	0.885
Method:	Least Squares	**F-statistic:**	3304.
Date:	Tue, 05 Dec 2023	**Prob (F-statistic):**	0.00
Time:	15:18:58	**Log-Likelihood:**	-15482.
No. Observations:	5597	**AIC:**	3.099e+04
Df Residuals:	5584	**BIC:**	3.108e+04
Df Model:	13		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Time	0.7469	0.108	6.900	0.000	0.535	0.959
CO(GT)	1.9500	0.213	9.165	0.000	1.533	2.367
PT08.S1(CO)	1.4583	0.211	6.917	0.000	1.045	1.872
PT08.S3(NOx)	6.5967	0.117	56.175	0.000	6.366	6.827
NO2(GT)	-2.0065	0.149	-13.452	0.000	-2.299	-1.714
PT08.S4(NO2)	16.1857	0.281	57.542	0.000	15.634	16.737
PT08.S5(O3)	2.2883	0.197	11.604	0.000	1.902	2.675
T	-0.3820	0.285	-1.340	0.180	-0.941	0.177
RH	-2.5794	0.225	-11.453	0.000	-3.021	-2.138
AH	-4.6455	0.241	-19.251	0.000	-5.119	-4.172
Day	1.2962	0.094	13.787	0.000	1.112	1.481
Month	15.0305	0.195	77.058	0.000	14.648	15.413
Year	27.9023	0.251	111.262	0.000	27.411	28.394

Omnibus:	1681.790	**Durbin-Watson:**	1.571
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	9543.441
Skew:	-1.318	**Prob(JB):**	0.00
Kurtosis:	8.828	**Cond. No.**	13.6

The model's performance did not change significantly, so now we will remove the T column.

```
X_train_rest.drop('T', axis=1, inplace=True)
X_train_rest_ready.drop('T', axis=1, inplace=True)
X_val_rest.drop('T', axis=1, inplace=True)
X_val_rest_ready.drop('T', axis=1, inplace=True)
```

```
multiple_linear = sm.OLS(y_train, X_train_rest_ready).fit()

multiple_linear.summary()
```

OLS Regression Results

Dep. Variable:	C6H6(GT)	**R-squared (uncentered):**	0.885
Model:	OLS	**Adj. R-squared (uncentered):**	0.885
Method:	Least Squares	**F-statistic:**	3579.
Date:	Tue, 05 Dec 2023	**Prob (F-statistic):**	0.00
Time:	15:19:49	**Log-Likelihood:**	-15483.
No. Observations:	5597	**AIC:**	3.099e+04
Df Residuals:	5585	**BIC:**	3.107e+04
Df Model:	12		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Time	0.7461	0.108	6.892	0.000	0.534	0.958
CO(GT)	2.0075	0.208	9.633	0.000	1.599	2.416
PT08.S1(CO)	1.4334	0.210	6.826	0.000	1.022	1.845
PT08.S3(NOx)	6.5958	0.117	56.165	0.000	6.366	6.826
NO2(GT)	-2.0235	0.149	-13.614	0.000	-2.315	-1.732
PT08.S4(NO2)	16.0634	0.266	60.365	0.000	15.542	16.585
PT08.S5(O3)	2.3380	0.194	12.070	0.000	1.958	2.718
RH	-2.3073	0.098	-23.651	0.000	-2.499	-2.116
AH	-4.8505	0.187	-25.983	0.000	-5.216	-4.485
Day	1.2900	0.094	13.737	0.000	1.106	1.474
Month	14.9688	0.190	78.969	0.000	14.597	15.340
Year	27.8526	0.248	112.291	0.000	27.366	28.339

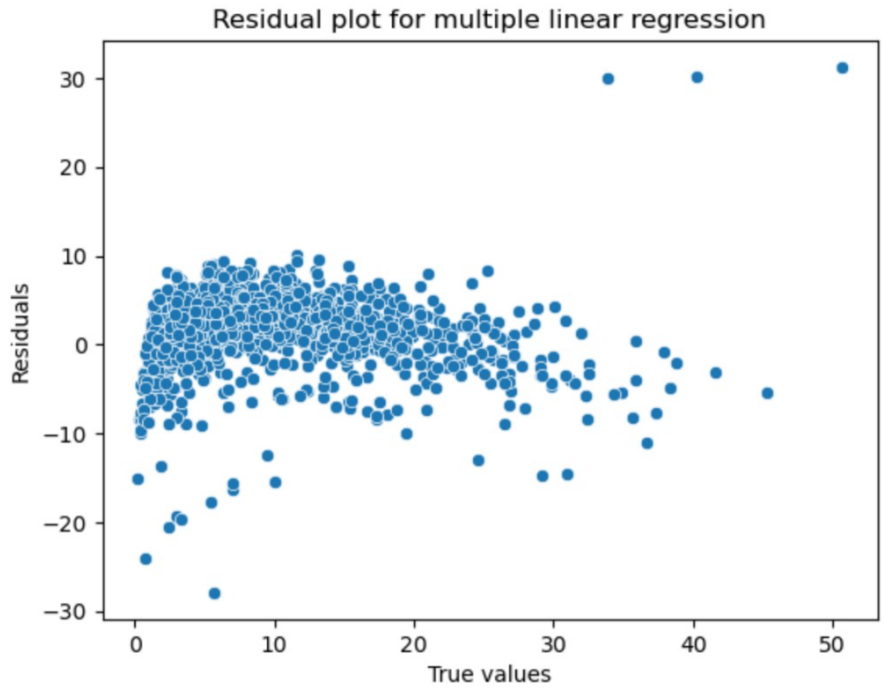
Omnibus:	1685.063	**Durbin-Watson:**	1.572
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	9560.222
Skew:	-1.321	**Prob(JB):**	0.00
Kurtosis:	8.832	**Cond. No.**	13.2

```
print('For train set')
print_metrics(y_train, multiple_linear.predict(X_train_rest_ready))
print('\nFor validate set')
print_metrics(y_val, multiple_linear.predict(X_val_rest_ready))
```

```
For train set
R squarred: 0.6242432674240165
MAE: 3.066355508380153
RMSE: 3.847093330338465

For validate set
R squarred: 0.6507384412841495
MAE: 3.243061970818393
RMSE: 4.351295712080002
```

We see that removing the column did not affect the model's accuracy. Let's conclude the work with the linear regression model by plotting the error graph.

```
residual_plot(y_val, multiple_linear.predict(X_val_rest_ready), 'multiple linear regression')
```



We see that the errors are mostly randomly scattered, which means that a linear model may be sufficient to describe the target variable. However, the model explains only about 60% of the dataset variance, so perhaps we can improve this result.

2.3. Linear model with Lasso regularization

Let's try to improve the predictive performance of the model by adding Lasso regularization. But first, we need to find the optimal value of the regularization parameter (alpha).

```
from sklearn.model_selection import KFold
from sklearn.linear_model import LassoCV

kfold = KFold(n_splits=5)

lasso_model = LassoCV(alphas=10**np.linspace(-2, 2, 100), cv=kfold, random_state=111).fit(X_train_rest_ready, y_train)
lasso_model.alpha_
```

```
0.01
```

We found that among all the values we tested, the optimal alpha is 0.01. This means the model requires very little regularization. Let's look at the coefficients of the new model.

```
pd.Series(lasso_model.coef_, index=X_train_rest_ready.columns)
```

```
Time            0.000000
CO(GT)          2.656348
PT08.S1(CO)     0.451342
PT08.S3(NOx)    0.000000
NO2(GT)        -0.000000
PT08.S4(NO2)    6.649210
PT08.S5(O3)     1.154865
RH             -0.827751
AH             -2.762541
Day             0.230685
Month           3.605040
Year            5.050605
dtype: float64
```

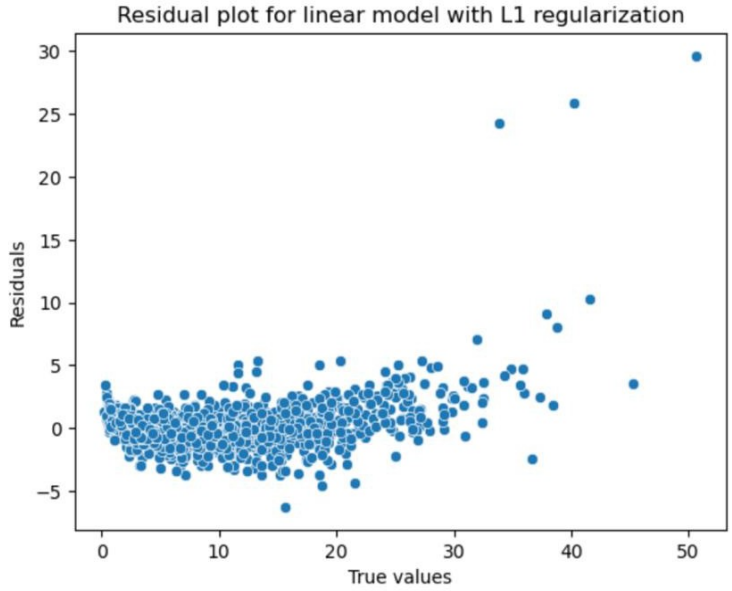
We see that even with very minimal regularization, the model eliminated some features: Time, PT08.S3(NOx), and NO2(GT), setting their coefficients to zero. From this, we can conclude that these features are not important for predicting the target variable.

Let's evaluate the model's performance.

```
print('For train set')
print_metrics(y_train, lasso_model.predict(X_train_rest_ready))
print('\nFor validate set')
print_metrics(y_val, lasso_model.predict(X_val_rest_ready))
```

```
For train set
R squarred: 0.9619497860633196
MAE: 0.9151877744052858
RMSE: 1.224215541205518

For validate set
R squarred: 0.9365409905148723
MAE: 1.0507401842619113
RMSE: 1.8547690883900332
```

```
residual_plot(y_val, lasso_model.predict(X_val_rest_ready), 'linear model with L1 regularization')
```



We see that after adding very slight regularization, the model's performance has significantly improved. As for the error plot, the errors are distributed just as randomly as they were without regularization.

2.4. Linear Model with Ridge Regularization

Now let's see how the model performs with Ridge regularization. Again, we need to find the optimal alpha value.

```
from sklearn.linear_model import RidgeCV

ridge_model = RidgeCV(alphas=10**np.linspace(-2, 2, 100), cv=kfold).fit(X_train_rest_ready, y_train)
ridge_model.alpha_
```

```
0.01
```

The optimal regularization parameter remained unchanged. However, let's take a look at how the coefficients have changed.

```
pd.Series(ridge_model.coef_, index=X_train_rest_ready.columns)
```

```
Time            0.062084
CO(GT)          2.380723
PT08.S1(CO)     0.487593
PT08.S3(NOx)    0.283572
NO2(GT)        -0.110050
PT08.S4(NO2)    7.527720
PT08.S5(O3)     1.167539
RH             -0.858798
AH             -3.193380
Day             0.316925
Month           4.408224
Year            6.416081
dtype: float64
```

Comparing the coefficients, we see that Ridge regularization does not tend to eliminate features but instead assigns them small coefficients.

Finally, let's look at the metrics and the error plot.

```
print('For train set')
print_metrics(y_train, ridge_model.predict(X_train_rest_ready))
print('\nFor validate set')
print_metrics(y_val, ridge_model.predict(X_val_rest_ready))
```

```
For train set
R squarred: 0.9628709809662797
MAE: 0.9166249979668911
RMSE: 1.209305629501642

For validate set
R squarred: 0.938090581513667
MAE: 1.046655060931906
RMSE: 1.8319835394361927
```

```
residual_plot(y_val, ridge_model.predict(X_val_rest_ready), 'linear model with L2 regularization')
```

We see that the model results have not changed significantly.

2.5. Combining L1 and L2 Regularization

Now, let's try to improve the model by combining both regularizations, implementing an Elastic Net model. For this model, we also need to optimize the ratio between L1 and L2 regularization.

```
from sklearn.linear_model import ElasticNetCV

elastic_net_model = ElasticNetCV(l1_ratio=[.1, .5, .7, .9, .95, .99, 1], alphas=10**np.linspace(-2, 2, 100)).fit(X_train_rest_ready, y_train)
print(f'Optimal alpha: {elastic_net_model.alpha_}')
print(f'Optimal l1_ratio {elastic_net_model.l1_ratio_}')
```

```
Optimal alpha: 0.01
Optimal l1_ratio 1.0
```

We see that the alpha value remains unchanged, and the L1 ratio equals 1, which means the model fully implements Lasso regression. Therefore, the results of this model will be the same as those of the previously built Lasso model.

2.6. Polynomial Regression

Now, let's build a polynomial regression model by creating second-degree features. The only issue is that this will significantly increase the number of features, so we also need to use Lasso regularization to eliminate a large number of unimportant features.

```
poly = PolynomialFeatures(degree=2).fit(X_train_rest_ready)

X_train_rest_poly = pd.DataFrame(poly.transform(X_train_rest_ready), columns=poly.get_feature_names_out(), index=X_train_rest_ready.index)
X_val_rest_poly = pd.DataFrame(poly.transform(X_val_rest_ready), columns=poly.get_feature_names_out(), index=X_val_rest_ready.index)

lasso_polynomial_model = LassoCV(alphas=10**np.linspace(-2, 2, 100), cv=kfold, random_state=111).fit(X_train_rest_poly, y_train)
lasso_polynomial_model.alpha_
```

```
0.01
```

We can see that even with polynomial features, the model requires only a small amount of regularization. Let's check how many features remain.

```
poly_coef_all = pd.Series(lasso_polynomial_model.coef_, index=X_train_rest_poly.columns)
poly_coef_left = poly_coef_all[poly_coef_all != 0]

print(f'Out of {len(poly_coef_all)} features only {len(poly_coef_left)} features were left.')
```

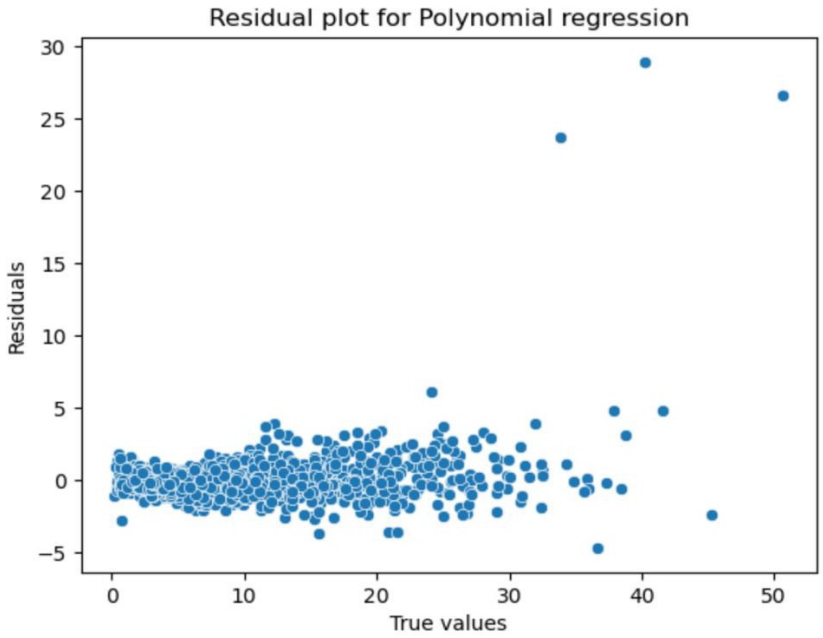
```
Out of 91 features only 43 features were left.
```

We can see that this model eliminated about half of all the features.

```
print('Train set')
print_metrics(y_train, lasso_polynomial_model.predict(X_train_rest_poly))
print('Test set')
print_metrics(y_val, lasso_polynomial_model.predict(X_val_rest_poly))
```

```
Train set
R squarred: 0.980648640479165
MAE: 0.6376258683574003
RMSE: 0.8730418808431631
Test set
R squarred: 0.9559505483566814
MAE: 0.7484567095366079
RMSE: 1.5453023249517142
```

```
residual_plot(y_val, lasso_polynomial_model.predict(X_val_rest_poly), 'Polynomial regression')
```



On the error plot, we can see that the residuals are distributed a bit more randomly. Considering this and the high R-squared value, we conclude that the polynomial regression fits better than the linear one.

2.7. K-Nearest Neighbors Model

The last regression model we will try is KNN. But before that, we need to optimize the number of neighbors.

```
knn_gs = GridSearchCV(KNeighborsRegressor(), {'n_neighbors': np.arange(5, 151, 5)}, cv=kfold).fit(X_train_rest_ready, y_train)

print(f'Best parameters: {knn_gs.best_params_} (gives score {knn_gs.best_score_})')

knn = knn_gs.best_estimator_
```

```
Best parameters: {'n_neighbors': 5} (gives score 0.9639684800887807)
```

```
print('For train set')
print_metrics(y_train, knn.predict(X_train_rest_ready))
print('\nFor validate set')
print_metrics(y_val, knn.predict(X_val_rest_ready))
```

```
For train set
R squarred: 0.9783697555982825
MAE: 0.6529753142616052
RMSE: 0.9230177448181968

For validate set
R squarred: 0.9219602932869116
MAE: 1.042832478902928
RMSE: 2.056842413116951
```

We see that the constructed KNN model explains 90% of the dataset variance, which is an excellent result (although not the best among all the models we trained).

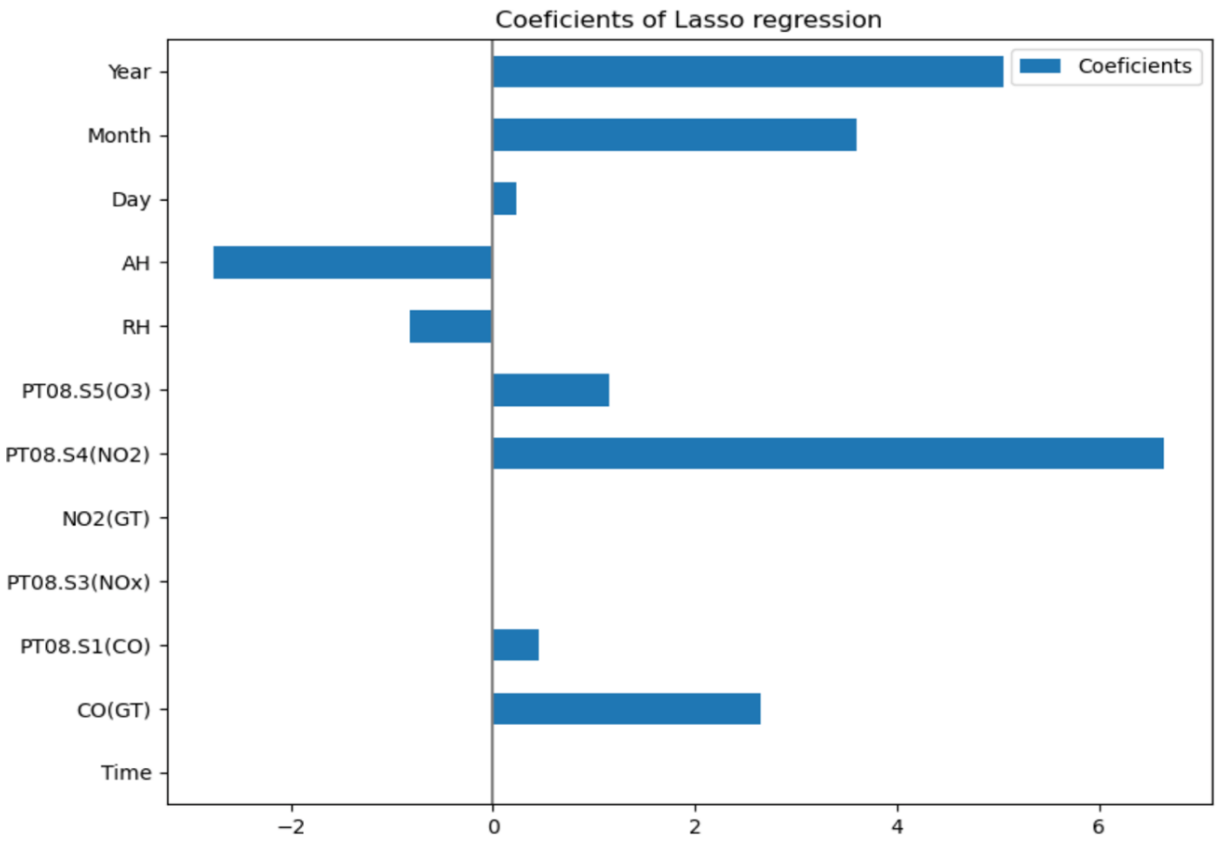
3. Analysis of Model Results and Accuracy

3.1. Feature Importance

Before building the final model, it is worth looking at which features the model uses most frequently for predictions.

To do this, we will examine the coefficients from the Lasso-regularized linear model, since this model attempts to eliminate unimportant features by assigning them coefficients as close to zero as possible.

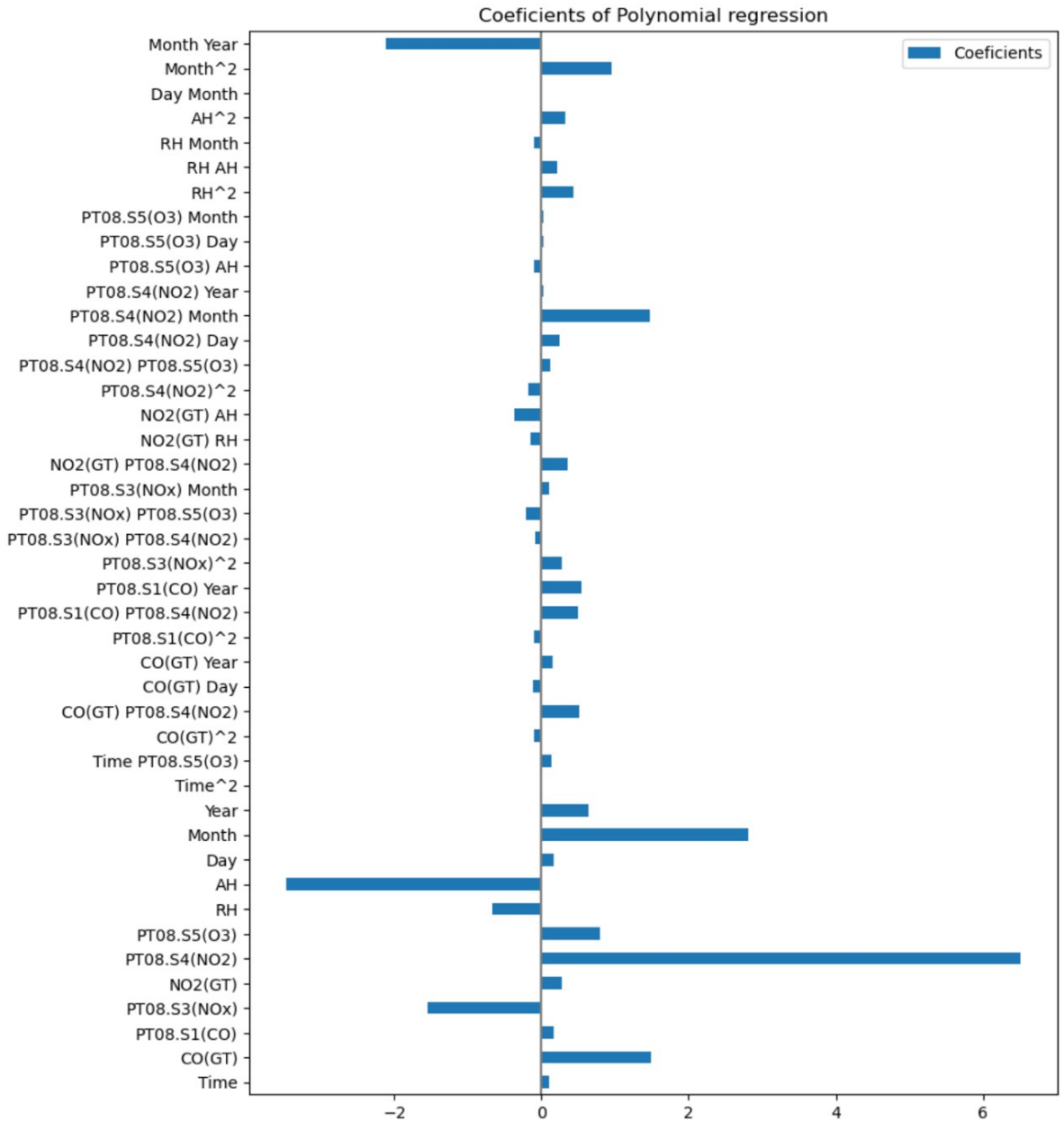
```
def coeficient_plot(coef, title, size=(9, 7)):
    coef.plot.barh(label='Coeficients', figsize=size)
    plt.axvline(x=0, color='grey')
    plt.legend()
    plt.title(f'Coeficients of {title}')
    plt.show()

coeficient_plot(pd.Series(lasso_model.coef_, index=X_train_rest_ready.columns), 'Lasso regression')
```



We see that PT08.S4(NO2) is the most important feature, but other features with high coefficients are Year and Month, indicating that changes are indeed dependent on time. We also observe that the model heavily relies on features such as the true average hourly concentration of carbon monoxide (CO(GT)), relative and absolute humidity (RH and AH, respectively), while other features were almost completely discarded.

Additionally, we will also look at the coefficients of the polynomial regression model.

```
coeficient_plot(poly_coef_left, 'Polynomial regression', (9, 12))
```



We see that among the original first-degree features, the subset of important features has remained relatively unchanged (PT08.S4(NO2) remains the most important, while time and humidity features are also significant). As for the other features, they are not as important as the original ones, with the most important being Month², Month*Year, and **PT08.S4(NO2)Month*. All this indicates that the model mainly relied on the hourly averaged response of the tungsten oxide sensor and the current month.

3.2. Choosing the Best Model

Now let's compare the results of all the models. To do this, we will use the R-squared metric to see how much of the dataset each model explains:

- Linear Regression: 0.6507
- Lasso Regression: 0.9365
- Ridge Regression: 0.9380
- Polynomial Regression: 0.9559
- K-Nearest Neighbors Regression: 0.9219

From these models, we see that polynomial regression performs the best.

Let's combine our model along with the preprocessing steps performed earlier into a single pipeline, and measure the final model metrics using the test set, which we have completely reserved.

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import FunctionTransformer

def drop_column(X, col):
    return X.drop(col, axis=1)

def restre_column_names(X):
    col_names = ['Time', 'CO(GT)', 'PT08.S1(CO)', 'NOx(GT)', 'PT08.S3(NOx)', 'NO2(GT)', 'PT08.S4(NO2)', 'PT08.S5(O3)', 'T', 'RH', 'AH', 'Day
    return pd.DataFrame(X, columns=col_names)

final_pipeline = Pipeline([
    ('drop_PT08.S2(NMHC)', FunctionTransformer(func=drop_column, kw_args={'col': 'PT08.S2(NMHC)'})),
    ('preprocessing', preprocessing),
    ('restore_column_names', FunctionTransformer(func=restre_column_names)),
    ('drop_NO2(GT)_T', FunctionTransformer(func=drop_column, kw_args={'col': ['NOx(GT)', 'T']})),
    ('polynomial_features', poly),
    ('model', lasso_polynomial_model)
])

print_metrics(y_test, final_pipeline.predict(X_test))
```

```
R squarred: 0.9822794206248432
MAE: 0.6778709682749746
RMSE: 0.9906394084590425
```

We see that the final model explained 98% of the test set variance and predicted values with an average error of 0.68 µg/m³.

Conclusions:

During the implementation of this project, a regression model was developed to predict benzene concentration in the air. The study can be summarized as follows:

1. **Data Preprocessing:**

- The dataset was thoroughly examined, and missing values were primarily imputed using the k-nearest neighbors (k-NN) imputation technique, while other missing entries were excluded from the analysis.
- An initial exploratory data analysis revealed that the features generally follow a normal distribution but contain some notable outliers.
- A suspiciously high correlation was identified between the target variable and the feature PT08.S2(NMHC), suggesting potential data leakage, which required further investigation.

2. **Model Development:**

- The modeling process began with the implementation of a baseline linear regression model, which demonstrated suboptimal predictive performance.
- Subsequent models incorporated regularization techniques (Lasso and Ridge), which significantly enhanced model accuracy and robustness by mitigating overfitting and feature multicollinearity.
- The final modeling step involved augmenting the feature space with polynomial terms, further improving predictive performance by capturing nonlinear relationships within the data.

3. **Model Comparison and Final Evaluation:**

- To determine the best performing model, multiple regression approaches were compared using the R-squared metric. Among these, the polynomial regression model exhibited the highest explanatory power (Polynomial Regression).
- The final step combined the optimal model with the previously applied preprocessing steps into a single pipeline.
- Evaluation on the fully reserved test set demonstrated that the final model explained 98% of the variance and predicted benzene concentration with an average error of 0.68 $\mu g/m^3$.