

PREDICTING CRITICAL PARAMETERS ON BLAST FURNACE USING MACHINE LEARNING- PYTHON-T

A report submitted as a part of the Industrial Orientation Training in

IT & ERP Department,

Visakhapatnam Steel Plant

By

KHUSHI DHRUW



Trainee No: 100029709

Department of Computer Science Engineering

Under the guidance of

MS. SRIYA BASU MALIK

(Manager)



RASHTRIYA ISPAT NIGAM LIMITED (RINL), Visakhapatnam

(Duration: 27 MAY 2024 to 22 JUNE 2024)

CERTIFICATE

This is to certify that Khushi Dhruw of Kalinga Institute of Industrial Technology, Bhubaneswar, is engaged in the project work titled
“PREDICTING CRITICAL PARAMETERS OF BLAST FURNACE USING MACHINE LEARNING-PYTHON-T”
(from 27 May 2024 to 22 June 2024.)

Khushi Dhruw

TR No:100029709

In partial fulfilment of the degree BACHELORS OF TECHNOLOGY in Computer Science and Engineering stream in Kalinga Institute of Industrial Technology, Bhubaneswar is a record of Bonafide work carried out under the guidance and supervision of Sriya Basu Malik during the period from 27 May 2024 to 22 June 2024.

Date: 22 June 2024

Place: Visakhapatnam

MS. SRIYA BASU MALIK

IT & ERP Department

RINL-VSP

DECLARATION

I am the student under the guidance of **Ms. Sriya Basu Malik (Manager)**, IT & ERP department, Visakhapatnam, steel plant, hereby declare that the project entitled “**PREDICTING CRITICAL PARAMETERS ON BLAST FURNACE AND PREDICTING CO/CO₂ RATIO USING MACHINE LEARNING**” is an original work done at **Rashtriya Ispat Nigam Limited (RINL) Visakhapatnam Steel Plant**, submitted in partial fulfilment of the requirements for the award of Industrial Training project work. I assure that this project is not submitted in any university or college.

Place: VISHAKAPATNAM

ACKNOWLEDGEMENT

We would like to express our deep gratitude for the valuable guidance of our guide MS. SRIYA BASU MALIK, MANAGER, IT and ERP department, Visakhapatnam, Steel plant for all her guidance, help and ongoing support throughout the course of this work by explaining basic concepts of predicting critical parameters of blast furnace system and their functioning along with industry requirements. We are immensely grateful to you madam. Without whose inspiration and valuable support our training would never have taken off.

We sincerely thank the Training and Development Centre (T&DC) for their guidance during safety training and encouragement in successful completion of our training.

Khushi Dhruw

TR NO:100029709

ABSTRACT

This internship project aims to apply machine learning techniques to predict critical parameters in blast furnace operations, with a specific focus on the CO/CO₂ ratio. The CO/CO₂ ratio is a vital indicator of the efficiency and environmental impact of the blast furnace process. Due to the complexity and interdependence of variables in blast furnace operations, traditional modelling approaches often fall short in accuracy and reliability.

This project seeks to overcome these challenges by utilizing advanced machine learning algorithms. Throughout the internship, data from various sensors and control systems within the blast furnace will be collected and pre-processed to create a robust dataset.

Multiple machine learning models, including regression algorithms and neural networks, will be developed and validated using this dataset. Feature selection techniques will be employed to identify the most influential parameters affecting the CO/CO₂ ratio, enabling the creation of a more targeted and efficient predictive model. The expected outcome of this project is to demonstrate that machine learning models can significantly improve the accuracy of predictions for the CO/CO₂ ratio and other critical parameters compared to traditional methods.

The successful application of these machine learning models is expected to markedly improve the accuracy of predictions for the CO/CO₂ ratio and other vital metrics, outperforming traditional methods.

Keywords: Machine Learning Models, Data Preprocessing, Feature Regression Algorithms, Model Training and Validation, Hyperparameter Tuning Time Series Analysis, Predictive Analytics, Scikit-Learn, Performance Metrics. implementation of these models in a blast furnace environment has the potential to enhance operational efficiency, reduce energy consumption, and lower emissions.

TABLE OF CONTENTS

A Brief Overview of Steel Plant	07-08
---------------------------------------	-------

PREDICTING CRITICAL PARAMETERS ON BLAST FURNACE AND PREDICTING CO/CO₂ RATIO USING MACHINE LEARNING

Introduction.....	09-14
Data Cleaning and Manipulation	15-23
Prediction.....	24-29

YOLO

YOLO Introduction	30-35
Training	36-38
Prediction and Results.....	39-40

PPE (PERSONAL PROTECTIVE EQUIPEMENT) DETECTION WEBSITE

Introduction.....	40-42
Frontend.....	43-44
Backend	45-46
Conclusion	47-48

Brief Overview of Steel Plant:



Visakhapatnam Steel Plant (VSP) is the integrated steel plant of Rashtriya Ispat Nigam Limited in Visakhapatnam, founded in 1971. VSP strikes everyone with a tremendous sense of awe, wonder, and amazement as it presents a wide array of excellence in all its facets in scenic beauty, technology, human resources, management, and product quality. On the coast of the Bay of Bengal and by the side of scenic Gangavaram beach has tall and massive structures of technological architecture, the Visakhapatnam Steel Plant. Nevertheless, the vistas of excellence do not rest with the inherent beauty of location over the sophistication of technology-they march ahead, parading one aspect after another. The decision of the Government of India to set up an integrated steel plant at Visakhapatnam was announced by then Prime Minister Smt. Indira Gandhi in Parliament on 17 January 1971. VSP is the first coastal- based integrated steel plant in India, 16 km west of the city of destiny,

Visakhapatnam, bestowed with modern technologies; VSP has an installed capacity of 3 million tons per annum of liquid steel and 2.656 million tons of saleable steel. The saleable steel here is in the form of wire rod coils, Structural, Special Steel, Rebar, Forged Rounds, etc. At VSP, there lies emphasis on total automation, seamless integration, and efficient up-gradation. This results in a wide range of long and structural products to meet 8 stringent demands of discerning customers in India & abroad; SP product meets exalting international Quality Standards such as JIS, DIN, BIS, BS, etc.

VSP has become the first integrated steel plant in the country to be certified to all the three international standards for quality (ISO -9001), Environment Management (ISO-14001) & Occupational Health & Safety (OHSAS—18001). The certificate covers quality systems of all operational, maintenance, and service units besides purchase

systems, Training, and Marketing functions spreading over 4 Regional Marketing offices, 20 branch offices & 22 stockyards located all over the country. VSP, by successfully installing & operating efficiently Rs.460 crores worth of pollution and Environment Control Equipment and converting the barren landscape more than 3 million plants have made the steel plant, steel Township, and VP export Quality Pig Iron & Steel products Sri Lanka, Myanmar, Nepal, Middle East, USA. & South East Asia

(Pig iron). RINL—VSP was awarded —Star Trading HOUSE status during 19972000. Having established a fairly dependable export market, VP Plans to make a continuous presence in the export market.

Different sections at the RINL VSP:

- Coke oven and coal chemicals plant
- Sinter plant
- Blast Furnace
- Steel Melt Shop
- Continuous casting machine
- Light and medium machine mills
- Calcining and refractive materials plant
- Rolling mills
- Thermal power plant
- Chemical power plant

Introduction:

Predictive data analytics contains a variety of statistical methods from data mining, predictive modelling, and ML that analyse historical facts to make predictions about prospects or unidentified occasions.

CRISP-DM breaks the process of data mining into six major phases:

- Business Understanding
- Data Understanding
- Data Preparation
- Modelling
- Evaluation
- Deployment

➤ The fourth stage (modelling) is where machine learning (ML) algorithms are engaged to build predictions for Lead time. In particular, ML is defined as an automated computational method that —learns and extracts information and patterns directly from (historical) data. There are four approaches

- 1) Supervised
- 2) Unsupervised
- 3) Semi-Supervised
- 4) Reinforcement Machine Learning.

● **Supervised ML** – Is a learning task with a full set of labelled data while training an algorithm. In other words, it accepts that training instances are classified or labelled (learning affiliation between a set of descriptive features and a target feature). In Supervised ML, the model needs to be trained on a classified dataset that means we have both raw input data as well as its outcomes. We divided our data into a training dataset and test dataset, where the training dataset is used to train our system, whereas the test dataset turns into new data for predicting results or to see the accuracy of our model. Supervised ML is relatively a simpler, highly accurate, and trustworthy method. There are three key areas where supervised learning is beneficial: Classification Techniques, Regression, and Forecasting. Classification techniques use the algorithm to predict a discrete value, focused on predicting a qualitative reaction by analysing data and identifying patterns. On the other hand, regression techniques used continuous data. The technique classically used in predicting, forecasting, and finding relationships amongst quantitative data. Forecasting is the process of making predictions about the prospect based on the past and present data. It is most commonly used to analyse trends.

● **Unsupervised ML**- Concerns the analysis of unclassified examples. In unsupervised learning, a deep learning model is offered a dataset without categorical

commands on what to do with it. The training dataset in this approach is a collection of instances without a specific desired outcome or an 10 accurate answer. Unsupervised learning is computationally compound, less accurate, and reliable method.

- **Semi-Supervised ML (SSL)** - It is expected that there is also unlabeled data available at the time of training in addition to the labeled data. The objective of SSL methods is to excerpt information from the unlabeled data that could ease learning a discriminative model with greater performance.
- **Reinforcement Machine Learning** - Is a category of machine learning techniques that enables an agent (Artificial Intelligent agent) to learn in a collaborating environment by trial and error using response from its own activities and experiences.

Regression technique in supervised machine learning predicts a single and continuous target output value using training data. In our scenario, modelling the association between the continuous variable (e.g. lead-time) and one or more predictors (For example, supplier origin, Buyer destination, Order type, etc.) using a linear function is the most suitable approach. One of the major strengths of using a regression algorithm is that the Outputs always have a probabilistic analysis and can be normalized to avoid overfitting. However, the Weaknesses of Logistic regression may underperform when there are numerous or nonlinear decision limitations. This method is not malleable, so it does not capture more complex.

Background on Blast Furnace:

A blast furnace is a crucial component in the iron and steel industry, used primarily for the extraction of iron from its ores. The process involves the chemical reduction of iron ore (mostly consisting of iron oxides) to produce molten iron, commonly known as pig iron, which can then be further refined to produce steel.

Operation and Components of a Blast Furnace

1. Structure and Design

A blast furnace is a large, vertical cylindrical structure designed to process iron ore into molten iron. It is lined with refractory materials to withstand the high temperatures required for the smelting process. The furnace is divided into several zones, each playing a specific role in the smelting process.

2. Raw Materials

Iron Ore: The primary raw material, typically in the form of hematite (Fe_2O_3) or magnetite (Fe_3O_4).

Coke: A form of carbon derived from coal, used as both a fuel and a reducing agent.

Limestone: Added as a flux to remove impurities in the form of slag.

Hot Blast: Preheated air injected into the furnace to aid combustion and maintain high temperatures.

3. Process Stages

Charging: Raw materials are added from the top of the furnace in alternating layers of iron ore, coke, and limestone.

Reduction Zone: As materials descend, coke burns with the hot blast, producing carbon monoxide (CO) which reduces the iron ore to molten iron.

Smelting Zone: At higher temperatures, the reduced iron melts and collects at the bottom of the furnace (hearth).

Slag Formation: Limestone reacts with impurities to form slag, which floats on the molten iron and can be removed.

Tapping: Molten iron and slag are periodically tapped from the furnace for further processing.

4. Gas Flow

The upward flow of gases (primarily CO, CO_2 , and N_2) generated from the combustion of coke and reduction reactions is crucial for the efficiency of the process.

Importance of Key Parameters

- **Temperature:** Critical for maintaining the proper chemical reactions. Multiple temperature measurements (such as CB_TEMP, STEAM_TEMP, TOP_TEMP) help monitor and control thermal conditions within different zones of the furnace.
- **Pressure:** Ensures the proper flow of gases and materials. Measurements like CB_PRESS, O2_PRESS, and TOP_PRESS are vital for operational stability.
- **Gas Composition:** Parameters such as CO, CO_2 , and O_2 content provide insights into combustion efficiency and environmental impact.

- Flow Rates: Parameters like CB_FLOW, STEAM_FLOW, and O2_FLOW help maintain the right balance of inputs for optimal operation.
- Auxiliary Conditions: Factors like atmospheric humidity (ATM_HUMID) and Pulverized Coal Injection (PCI) rate influence overall efficiency and need to be finely controlled.

Modern Challenges and Innovations in Blast Furnace Operations

Traditional Challenges:

Operating a blast furnace involves managing complex interdependencies between various parameters such as temperature, pressure, gas composition, and flow rates. Manual control of these parameters is challenging due to their dynamic nature and the need for precise adjustments to maintain optimal performance.

Modern Advancements:

Recent technological advancements have focused on automating and optimizing blast furnace operations using modern technology. Key innovations include:

1. **Automation:** Automation reduces the need for manual intervention, allowing for more consistent and precise control over the blast furnace parameters.
2. **Machine Learning and Data Analytics:** These technologies are becoming integral in predicting and controlling blast furnace parameters. They enhance efficiency, reduce energy consumption, and minimize environmental impact by providing deeper insights into furnace operations.

Leveraging Machine Learning:

By utilizing data from sensors and control systems, machine learning models can predict critical outcomes and allow for proactive adjustments. These models analyze the complex interactions between parameters to optimize the operation of the furnace. For example, predicting the CO/CO₂ ratio is crucial as it is a key indicator of furnace performance and environmental impact.

Project Focus: Enhancing Efficiency and Sustainability

This project aims to enhance the efficiency and sustainability of blast furnace operations through machine learning techniques. The primary goal is to predict critical parameters, particularly the CO/CO₂ ratio. By developing robust predictive models, the project seeks to:

1. **Analyze the Dataset:** The dataset contains various operational parameters collected from the blast furnace. Analyzing this data is the first step towards building effective machine learning models.
2. **Predict Critical Parameters:** Focus on predicting the CO/CO₂ ratio to gain insights into furnace performance and environmental impact.
3. **Optimize Operational Parameters:** Use the predictions to make proactive adjustments, leading to more sustainable and efficient iron production.

Benefits:

- Enhanced Efficiency: Improved control over the furnace operations leads to higher efficiency and lower energy consumption.
- Reduced Environmental Impact: Optimizing the CO/CO₂ ratio and other parameters helps in minimizing the environmental footprint of the blast furnace operations.
- Sustainability: The use of advanced technologies ensures a more sustainable iron production process, aligning with modern environmental standards and regulations.

By leveraging machine learning and data analytics, the project sets a path for future advancements in blast furnace technology, ultimately contributing to more efficient and environmentally friendly iron production processes.

Let's go through each column in the dataset and explain what they likely represent, and then discuss possible relationships between them. The dataset we use is related to an industrial process, possibly in a chemical or power plant, involving combustion and heat exchange processes.

DATASET EXPLANATION

1. **CB_FLOW:** The flow rate of the combustion air or fuel in the combustion chamber, typically measured in cubic meters per second (m³/s) or similar units.

2. **CB_PRESS**: The pressure within the combustion chamber, usually measured in pascals (Pa) or bar.
3. **CN_TEMP**: The temperature at a specific point in the combustion process, possibly the combustion nozzle or a critical combustion zone, measured in degrees Celsius (°C) or Fahrenheit (°F).
4. **STEAM_FLOW**: The flow rate of steam produced, typically measured in kilograms per hour (kg/h) or similar units.
5. **STEAM_TEMP**: The temperature of the steam, measured in degrees Celsius (°C) or Fahrenheit (°F).
6. **STEAM_PRESS**: The pressure of the steam, usually measured in pascals (Pa) or bar.
7. **O2_PRESS**: The pressure of the oxygen supply, measured in pascals (Pa) or bar.
8. **O2_FLOW**: The flow rate of oxygen, typically measured in cubic meters per second (m³/s) or similar units.
9. **O2_PER**: The percentage of oxygen in the combustion process, indicating oxygen concentration, usually measured as a percentage (%).
10. **PCI**: The Lower Heating Value (LHV) or the net calorific value of the fuel used, typically measured in megajoules per kilogram (MJ/kg).
11. **ATM_HUMID**: Atmospheric humidity, usually measured as a percentage (%), indicating the moisture content in the air.
12. **HB_TEMP**: Temperature at a specific point in the heat exchange process, possibly the heat boiler or heat exchanger, measured in degrees Celsius (°C) or Fahrenheit (°F).
13. **HB_PRESS**: Pressure within the heat boiler or heat exchanger, usually measured in pascals (Pa) or bar.
14. **TOP_PRESS**: Pressure at the top of a column or vessel in the process, typically measured in pascals (Pa) or bar.
15. **TOP_TEMP1, TOP_TEMP2, TOP_TEMP3, TOP_TEMP4**: Temperatures at different points or stages at the top of a column or vessel, each measured in degrees Celsius (°C) or Fahrenheit
16. **TOP_SPRAY**: Flow rate or activation status of a spray system at the top of a column or vessel, typically measured in cubic meters per second (m³/s) or as a binary indicator.
17. **TOP_TEMP**: An aggregate or specific temperature at the top of a column or vessel, measured in degrees Celsius (°C) or Fahrenheit (°F).

18. **TOP_PRESS_1**: Another pressure measurement at the top of a column or vessel, providing additional data for accuracy or redundancy, measured in pascals (Pa) or bar.
19. **CO**: Concentration of carbon monoxide in the exhaust or flue gas, usually measured in parts per million (ppm).
20. **CO2**: Concentration of carbon dioxide in the exhaust or flue gas, usually measured in percentage (%).
21. **H2**: Concentration of hydrogen in the exhaust or flue gas, usually measured in parts per million (ppm).
22. **SKIN_TEMP_AVG**: Average temperature of the outer surface of equipment or vessels, usually measured in degrees Celsius (°C) or Fahrenheit (°F).

Data Cleaning, Analysis, and Pre-Processing

Data Cleaning

Ensuring the dataset's quality and integrity is critical before performing any analysis or modeling. The data cleaning process involved several key steps, utilizing key Python libraries such as pandas, numpy, and matplotlib:

1. Resampling the Data

- **Original Dataset**: The initial dataset consisted of sensor readings recorded every 10 minutes over a five-month period.
- **Resampling to Hourly Intervals**: Using the pandas library, the data was resampled to hourly intervals. This transformation involved aggregating the 10-minute readings into one-hour periods using the ``resample`` function in pandas. This function allowed for the calculation of averages, sums, or other relevant statistics to ensure that the transformed dataset accurately represented hourly trends.

2. Handling Missing Values

- **Identification of Missing Data**: The dataset was examined for any missing or incomplete data points using the ``isnull`` and ``sum`` functions in pandas. These functions helped identify the extent of missing data across different columns.
- **Imputation or Removal**: Depending on the extent and pattern of missing data, appropriate strategies were employed. Missing values were filled using the ``fillna`` or ``interpolate`` functions, or rows with excessive missing data were removed using the ``dropna`` function in pandas.

3. Feature Scaling and Normalization

- Standardization: The numpy library, along with the `StandardScaler` from the `sklearn.preprocessing` module, was used for standardizing the data. This ensured that all features contributed equally to the model training process by scaling features to have a mean of zero and a standard deviation of one.

4. Outlier Detection and Treatment

- Identification of Outliers: Outliers in the data were identified using statistical methods or visualization techniques. The `boxplot` function in the `matplotlib.pyplot` module was used to create box plots, which helped visualize the spread of the data and detect any outliers.

- Handling Outliers: Depending on the nature of the outliers, they were either removed or treated. Treatment could involve capping values or using robust statistical methods to minimize their impact.

-

Data Analysis and Pre-Processing

Once the data was cleaned, an in-depth analysis was conducted to uncover patterns and relationships within the dataset. The data analysis phase included the following steps:

1. Exploratory Data Analysis (EDA)

- Descriptive Statistics: Summary statistics were computed using pandas functions to understand the central tendencies, dispersion, and overall distribution of the data. Metrics such as mean, median, standard deviation, and percentiles were calculated.

- Data Visualization: Various visualization techniques were employed to explore the data visually using matplotlib. This included:

- Histograms: Created to understand the distribution of individual features.

- Box Plots: Used to detect outliers and visualize the spread of the data.

- Time Series Plots: Employed to observe trends and patterns over time for the hourly resampled data.

- Scatter Plots: Used to explore relationships between different features.

2. Correlation Analysis

- **Correlation Matrix:** A correlation matrix was generated using the ``corr`` function in pandas to quantify the linear relationships between different features. This helped identify which features had strong positive or negative correlations with each other.
- **Heatmaps:** Visual representations of the correlation matrix were created using the ``heatmap`` function from the seaborn library, making it easy to identify significant correlations.

3. Feature Engineering

- **Creating New Features:** Based on the insights from the exploratory data analysis, new features were engineered. For instance, to predict the CO/CO₂ ratio, four new columns representing the ratio for the 1st, 2nd, 3rd, and 4th hours were created.
- **Transformation of Existing Features:** Existing features were transformed or combined to create more meaningful representations that could improve the model's predictive power.

4. Pattern and Trend Analysis

- **Temporal Patterns:** Analysis was performed to detect temporal patterns, such as daily or weekly cycles, in the data.
- **Anomaly Detection:** Any anomalies or unusual patterns were identified and investigated to ensure they were understood and appropriately handled.

Modules Used:

- **pandas:** Used for data manipulation and analysis with dataframes, akin to spreadsheets. It handles creating new columns, handling missing values, and reading/writing to/from Excel files.
- **numpy:** Used for numerical operations and handling arrays.
- **matplotlib.pyplot:** Part of the Matplotlib library and used for creating visualizations, such as the correlation matrix plot.
- **seaborn:** Enhances Matplotlib's capabilities, providing stylish visualizations. While not directly called in the code, Matplotlib likely utilizes Seaborn's color palettes for the correlation matrix plot.

These steps collectively ensure that the data is clean, well-understood, and ready for building predictive models to enhance the efficiency and sustainability of blast furnace operations.

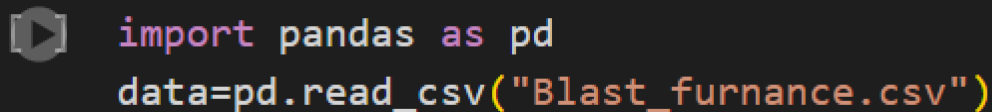
Code for Data Cleaning and Analysis:

This code imports necessary libraries (pandas, matplotlib.pyplot, and seaborn) and reads data from an Excel file located at

```
data=pd.read_excel('C:/Users/Jyothika/Downloads/bf3_data_2022_01_07.xlsx')
```

into a pandas dataframe (data). This sets up the environment for further data analysis and visualization.

Import Libraries and Read Data:import pandas as pd import numpy as np
from datetime import datetime from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor from sklearn.metrics import
r2_score, mean_squared_error, mean_absolute_error import matplotlib.pyplot as plt



```
import pandas as pd  
data=pd.read_csv("Blast_furnance.csv")
```

Importing Libraries: The necessary libraries are imported for data manipulation (`pandas`, `NumPy`), date and time handling (`datetime`), machine learning (`scikit-learn`), and plotting (`matplotlib`).

Reading Data: The data is read from an Excel file (`Blast_furnance.csv`) into a Data Frame (`data`). The first few rows and the info (column names, non-null counts, data types) are printed to get an overview.

Data Inspection: These commands are used to further inspect the data:

- `df.head(6)`: Displays the first 6 rows of the DataFrame.
- `df.columns`: Lists the column names.
- `df.shape`: Displays the dimensions of the DataFrame.
- `df.describe()`: Provides summary statistics for numeric columns.
- `df.info()`: Provides a summary of the DataFrame (already called earlier).

```
data.head()
data.tail()
```

	DATE_TIME	CB_FLOW	CB_PRESS	CB_TEMP	STEAM_FLOW	STEAM_TEMP	STEAM_PRESS	O2_PRESS	O2_FLOW	O2_PER	...	TOP_TEMP2	TOP_TEMP3	TOP_TEMP4
25400	31-12-21 23:10	278198.0	2.75	76.0	2.0	189.0	2.92	2.79	2628.0	22.25	...	116.0	108.0	122.0
25401	31-12-21 23:20	286486.0	2.80	77.0	1.0	190.0	2.97	2.84	2590.0	22.22	...	113.0	105.0	119.0
25402	31-12-21 23:30	284500.0	2.81	77.0	0.0	191.0	2.98	2.85	2592.0	22.20	...	116.0	108.0	115.0
25403	31-12-21 23:40	284455.0	2.83	77.0	1.0	190.0	3.00	2.87	2582.0	22.21	...	119.0	113.0	121.0
25404	31-12-21 23:50	274728.0	2.73	77.0	2.0	189.0	2.90	2.78	2593.0	22.23	...	120.0	111.0	121.0

5 rows × 26 columns

Data Preprocessing: Preprocessing is a crucial step in data analysis and machine learning because it helps clean and transform raw data into a suitable format for modelling.

a. Dropping Unnecessary Columns

We drop the column `SKIN_TEMP_AVG` as it is not needed for our analysis.

```
[ ] data.drop(columns=['SKIN_TEMP_AVG'],inplace=True)
data.head()
```

	DATE_TIME	CB_FLOW	CB_PRESS	CB_TEMP	STEAM_FLOW	STEAM_TEMP	STEAM_PRESS	O2_PRESS	O2_FLOW	O2_PER	...	TOP_TEMP1	TOP_TEMP2	TOP_TEMP3
0	01-07-21 00:10	311727.0	3.15	129.0	4.0	213.0	3.34	3.20	7296.0	23.08	...	112.0	135.0	107.0
1	01-07-21 00:20	315163.0	3.16	129.0	4.0	208.0	3.35	3.20	7829.0	23.08	...	120.0	143.0	109.0
2	01-07-21 00:30	314595.0	3.16	128.0	4.0	205.0	3.35	3.21	7904.0	23.08	...	123.0	138.0	110.0
3	01-07-21 00:40	312465.0	3.16	127.0	4.0	200.0	3.35	3.21	7919.0	23.08	...	119.0	128.0	102.0
4	01-07-21 00:50	302981.0	3.11	126.0	4.0	194.0	3.29	3.18	7938.0	23.08	...	125.0	139.0	112.0

5 rows × 25 columns

b. Converting Date-Time Format

We convert the `DATE_TIME` column to a datetime format.

```
[ ] data['DATE_TIME'] = pd.to_datetime(data['DATE_TIME'], format="mixed")
```

c. Handling Missing Values

We drop any rows with missing values.

```
[ ] data.dropna(inplace=True)
```

d. Resampling Data Hourly:

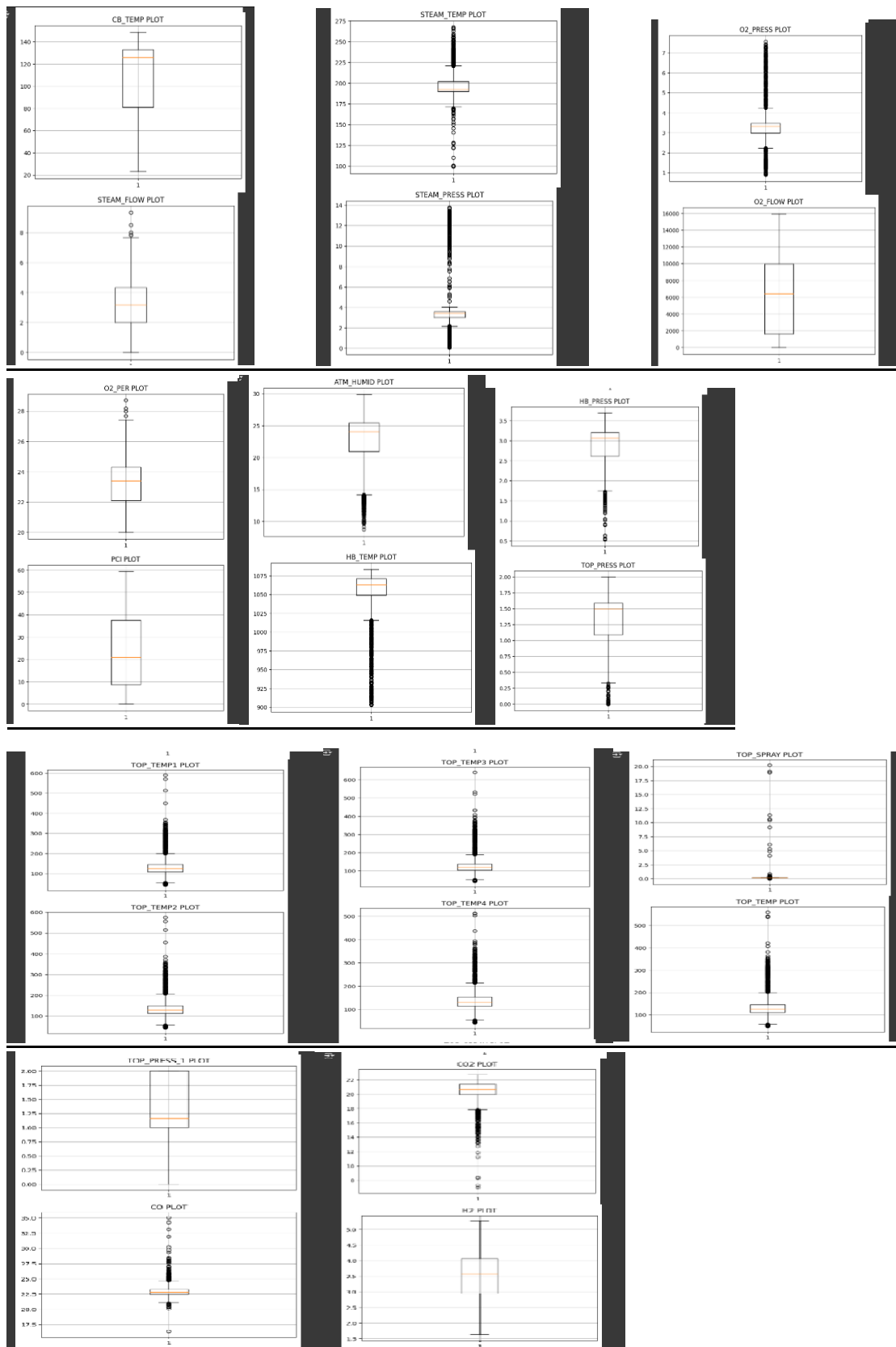
- Datetime Conversion: Convert the specified date-time column into a `Datetime` object.
- Calculate CO Ratio: Compute the ratio of CO to CO2 concentrations for each row.
- Convert Datetime to Epoch Milliseconds: Convert the `Datetime` to milliseconds since the epoch for easier time-based calculations.
- Aggregate Data Hourly: Group the data by hour, computing the mean values for each hour.

```
[ ] data_hourly = data.resample('H').mean().reset_index()
```

Box Plots for Each Feature

We create box plots for each feature to check for outliers.

```
[ ] import matplotlib.pyplot as plt
    for feature in data_hourly.columns.drop('DATE_TIME'):
        plt.figure()
        plt.boxplot(x=data_hourly[feature])
        plt.grid()
        plt.title(feature+' PLOT')
```



Histograms: We plot histograms to visualize the distribution of each feature.

```
print(data_hourly.describe())

data_hourly.hist(bins=50, figsize=(20, 15))
plt.show()
```

	count	DATE_TIME	CB_FLOW	CB_PRESS	CB_TEMP	\
count	4416	4416.000000	4416.000000	4416.000000	4416.000000	
mean	2021-09-30 23:30:00	281778.393210	2.856341	111.085658		
min	2021-07-01 00:00:00	172.000000	0.000000	23.000000		
25%	2021-08-15 23:45:00	284876.933333	2.760000	81.166667		
50%	2021-09-30 23:30:00	308544.433333	3.230000	126.000000		
75%	2021-11-15 23:15:00	324807.650000	3.370000	133.166667		
max	2021-12-31 23:00:00	354210.666667	3.871667	148.500000		
std	NaN	82003.914119	0.942368	31.304226		

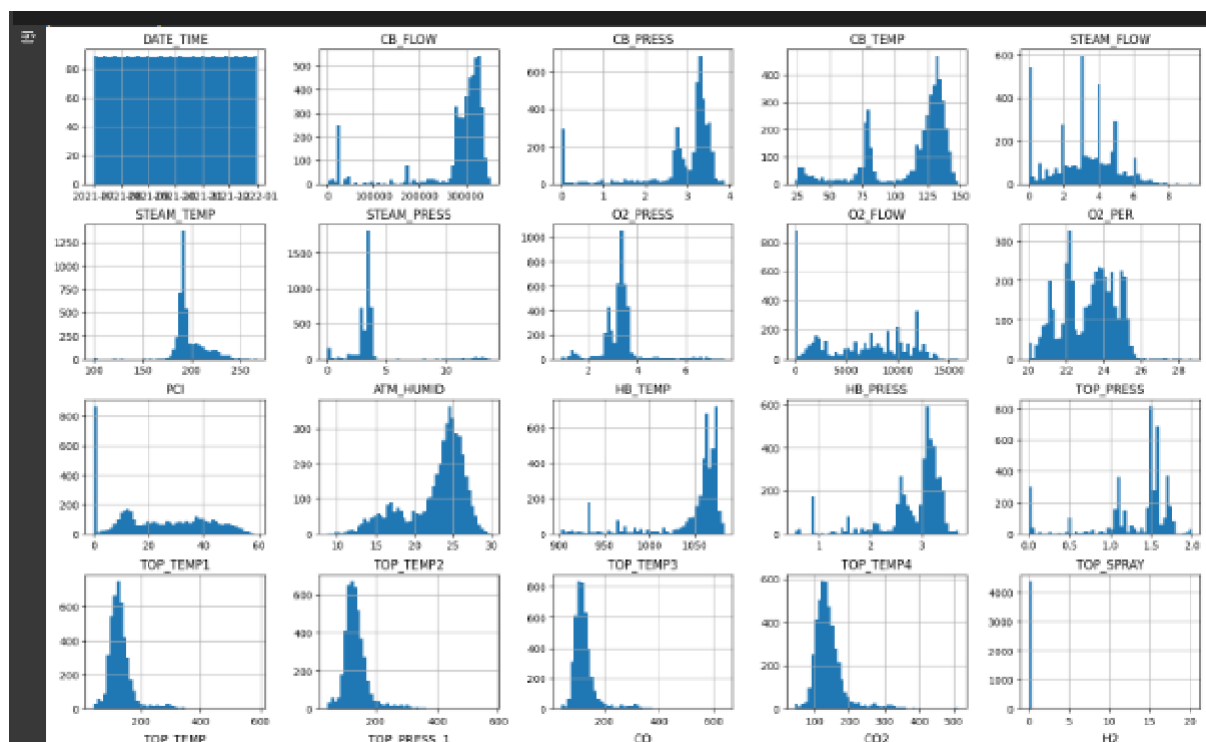
	STEAM_FLOW	STEAM_TEMP	STEAM_PRESS	O2_PRESS	O2_FLOW	\
count	4416.000000	4416.000000	4416.000000	4416.000000	4416.000000	
mean	3.093367	197.403499	3.586923	3.310851	5939.493694	
min	0.000000	99.333333	0.070000	0.890000	0.000000	
25%	2.000000	190.000000	3.025000	2.977250	1647.916667	
50%	3.166667	192.833333	3.443333	3.317333	6421.166667	
75%	4.333333	202.400000	3.600000	3.430000	9960.166667	
max	9.333333	267.500000	13.750000	7.553333	15920.500000	
std	1.785221	15.632923	1.982798	0.817679	4487.918933	

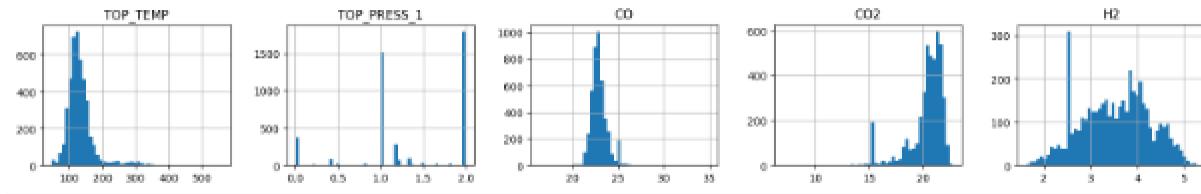
	O2_PER	...	TOP_TEMP1	TOP_TEMP2	TOP_TEMP3	TOP_TEMP4	\
count	4416.000000	...	4416.000000	4416.000000	4416.000000	4416.000000	
mean	23.192221	...	132.545686	135.622705	129.330174	140.110292	
min	20.000000	...	42.400000	42.000000	43.000000	42.666667	
25%	22.116667	...	108.650000	111.833333	104.000000	114.000000	
50%	23.410833	...	125.333333	128.666667	119.500000	131.366667	
75%	24.323500	...	145.000000	149.666667	138.541667	153.808333	
max	28.702000	...	589.333333	574.666667	640.333333	512.500000	
std	1.401998	...	43.374459	42.889989	47.175122	45.470548	

	TOP_SPRAY	TOP_TEMP	TOP_PRESS_1	CO	CO2	\
count	4416.000000	4416.000000	4416.000000	4416.000000	4416.000000	
mean	0.211878	134.401306	1.334979	22.971723	20.285166	
min	0.000000	48.333333	0.000000	16.355000	6.953333	
25%	0.200000	111.000000	1.000000	22.406250	19.960000	
50%	0.200000	126.500000	1.166667	22.826667	20.694000	
75%	0.200000	146.500000	2.000000	23.326667	21.375250	
max	20.233333	558.333333	2.000000	34.995000	22.723333	
std	0.609078	43.105445	0.636848	0.965206	1.686514	

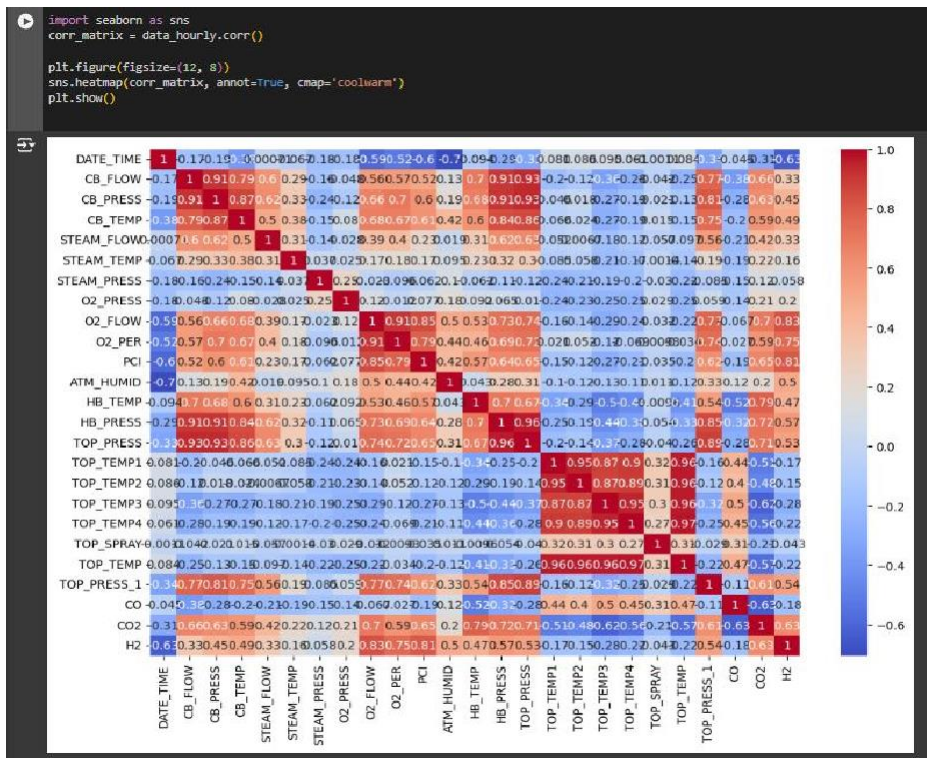
	H2
count	4416.000000
mean	3.522515
min	1.621667
25%	2.953333
50%	3.560000
75%	4.065250
max	5.264000
std	0.738551

[8 rows x 25 columns]

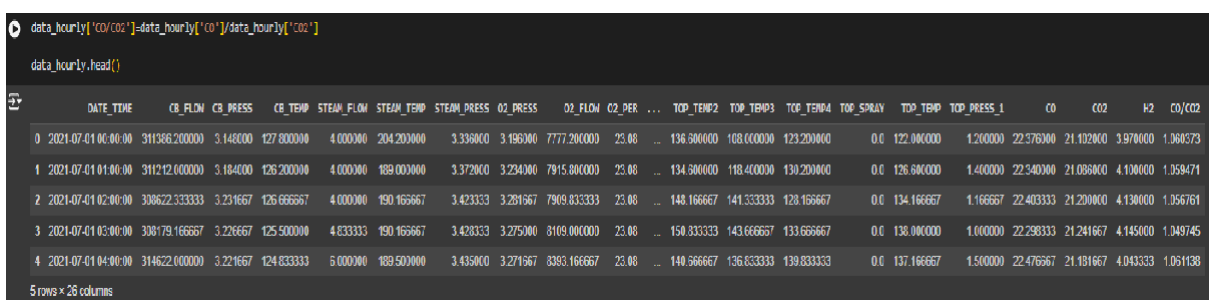




Correlation Matrix: We compute and visualize the correlation matrix to understand the relationships between features.



Feature Engineering: We create a new feature CO/CO_2 which is the ratio of CO to CO2.



Model Training:

We split the data into training and testing sets and train a Random Forest Regressor. We then evaluate the model using Mean Absolute Error, Mean Squared Error, Root Mean Squared Error, and R2 Score.

```
from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=42)

from sklearn.ensemble import RandomForestRegressor
regressor= RandomForestRegressor(n_estimators=100, random_state=42)
regressor.fit(X_train, Y_train)
Y_pred=regressor.predict(X_test)
```

```
[ ] from sklearn.metrics import mean_squared_error,mean_absolute_error,r2_score
print("Mean Absolute Error:", mean_absolute_error(Y_test, Y_pred))
print("Mean Squared Error:", mean_squared_error(Y_test, Y_pred))
print("Root Mean Squared Error:", mean_squared_error(Y_test, Y_pred, squared=False))
print("R2 Score:", r2_score(Y_test, Y_pred))
```

```
Mean Absolute Error: 0.001740767355652833
Mean Squared Error: 2.0208956673911358e-05
Root Mean Squared Error: 0.004495437317315342
R2 Score: 0.9990378193263123
```

Predicting Future Values

We create new columns to predict the CO/CO2 ratio for the next 1 to 4 hours.

```
[ ] for i in range(1, 5):
    data_hourly['f_{i}_hour'] = data_hourly['CO/CO2'].shift(-i)
```

	DATE_TIME	CB_FLOW	CB_PRESS	CB_TEMP	STEAM_FLOW	STEAM_TEMP	STEAM_PRESS	O2_PRESS	O2_FLOW	O2_PER	...	TOP_TEMP	TOP_PRESS_1	CO	CO2	H2	CO/CO2	1_hour	2_hour	3_hour	4_hour
0	2021-07-01 00:00:00	311388.200000	3.148300	127.890000	4.900000	204.200000	3.338000	3.196000	7777.200000	23.080000	...	122.000000	1.200000	22.378000	21.102000	3.970000	1.060373	1.058471	1.056751	1.048745	1.061138
1	2021-07-01 01:00:00	311212.000000	3.184000	126.290000	4.900000	189.000000	3.372000	3.234000	7915.800000	23.080000	...	126.600000	1.400000	22.340000	21.086000	4.100000	1.059471	1.056751	1.048745	1.061138	1.063399
2	2021-07-01 02:00:00	308622.333333	3.231567	126.566667	4.900000	190.166667	3.423333	3.281667	7909.833333	23.080000	...	134.166667	1.166667	22.403333	21.204000	4.130000	1.056761	1.045745	1.061138	1.063399	1.069543
3	2021-07-01 03:00:00	308179.166667	3.226967	125.500000	4.833333	190.166667	3.428333	3.275000	8109.000000	23.080000	...	138.000000	1.000000	22.293333	21.241667	4.145000	1.049745	1.061138	1.063399	1.069543	1.060743
4	2021-07-01 04:00:00	314622.000000	3.221567	124.833333	6.000000	189.500000	3.435000	3.271667	8393.166667	23.080000	...	137.166667	1.500000	22.479667	21.181667	4.043333	1.061138	1.063399	1.069543	1.060743	1.038005
...
4411	2021-12-31 19:00:00	264124.000000	2.795667	76.000000	4.833333	190.500000	2.953333	2.940000	2909.333333	22.326667	...	127.000000	1.000000	23.080000	20.513333	3.321667	1.125122	1.105874	1.096779	1.101688	1.088541
4412	2021-12-31 20:00:00	268424.833333	2.743333	76.000000	3.500000	190.833333	2.950000	2.786667	2905.333333	22.308333	...	118.333333	1.000000	22.405000	20.260000	3.188333	1.105874	1.096779	1.101688	1.088541	NaN
4413	2021-12-31 21:00:00	264148.500000	2.748333	76.000000	2.166667	192.000000	2.923333	2.791667	2896.333333	22.308333	...	115.166667	1.000000	22.309667	20.336333	3.123333	1.096779	1.101688	1.088541	NaN	NaN
4414	2021-12-31 22:00:00	266686.333333	2.820000	76.833333	3.333333	190.166667	3.010000	2.865000	2899.000000	22.328333	...	117.333333	1.000000	22.300000	20.241667	3.263333	1.101688	1.088541	NaN	NaN	NaN
4415	2021-12-31 23:00:00	262858.833333	2.790000	76.566667	1.166667	189.666667	2.960000	2.831667	2806.166667	22.221667	...	113.000000	1.000000	22.009667	20.216667	3.398333	1.088541	NaN	NaN	NaN	NaN

Recursive Prediction Function

We define a function to make predictions for each shift hour using the trained model and use the prediction function to predict the CO/CO2 ratio for the next 1 to 4 hours.


```
[ ] def prediction(X, Y, hour):
    X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=0)
    regressor= RandomForestRegressor(n_estimators=100, random_state=0)
    regressor.fit(X_train, Y_train)
    Y_pred = regressor.predict(X_test)

    print(f"Shift {hour} hour:")
    print("Mean Absolute Error:", mean_absolute_error(Y_test, Y_pred))
    print("Mean Squared Error:", mean_squared_error(Y_test, Y_pred))
    print("Root Mean Squared Error:", mean_squared_error(Y_test, Y_pred, squared=False))
    print("R2 Score:", r2_score(Y_test, Y_pred))

    return X_test, Y_test, Y_pred

[ ] X=data_hourly.drop(columns=['DATE_TIME','CO/CO2','1_hour','2_hour','3_hour','4_hour'])
Y1=data_hourly['1_hour']
Y2=data_hourly['2_hour']
Y3=data_hourly['3_hour']
Y4=data_hourly['4_hour']

[ ] X_test1,Y_test1,Y_pred1=prediction( X,Y1, hour=1)

⇌ Shift 1 hour:
Mean Absolute Error: 0.020296710334152816
Mean Squared Error: 0.002717827327609418
Root Mean Squared Error: 0.05213278553472294
R2 Score: 0.9351967846885286

[ ] X_test2,Y_test2,Y_pred2=prediction( X,Y2, hour=2)

⇌ Shift 2 hour:
Mean Absolute Error: 0.02903389318445374
Mean Squared Error: 0.01366803501666881
Root Mean Squared Error: 0.11691037172410672
R2 Score: 0.5057128667305009

[ ] X_test3,Y_test3,Y_pred3=prediction( X,Y3, hour=3)

⇌ Shift 3 hour:
Mean Absolute Error: 0.030988095785282593
Mean Squared Error: 0.009696196076741916
Root Mean Squared Error: 0.09846926462984232
R2 Score: 0.7180015070502581

[ ] X_test4,Y_test4,Y_pred4=prediction( X,Y4, hour=4)

⇌ Shift 4 hour:
Mean Absolute Error: 0.031897644324496884
Mean Squared Error: 0.00651904712520516
Root Mean Squared Error: 0.0807406163291138
R2 Score: 0.8197613090359572
```

Saving Predicted Data

We save the predicted CO/CO2 values for the next 1 to 4 hours into separate CSV files and sort the test data based on DATE_TIME to prepare for visualization.

```

▶ X_test1['1_hour '] = Y_test1
X_test1['1_hour pred'] = Y_pred1
X_test1= X_test1.reset_index()
X_test1['DATE_TIME'] = data_hourly.loc[X_test1['index'], 'DATE_TIME'].values
X_test1.to_csv('predicted_data_after_1_hours.csv', index=False)

[ ] X_test2['2_hour '] = Y_test2
X_test2['2_hour pred'] = Y_pred2
X_test2= X_test2.reset_index()
X_test2['DATE_TIME'] = data_hourly.loc[X_test2['index'], 'DATE_TIME'].values
X_test2.to_csv('predicted_data_after_2_hours.csv', index=False)

[ ] X_test3['3_hour '] = Y_test3
X_test3['3_hour pred'] = Y_pred3
X_test3= X_test3.reset_index()
X_test3['DATE_TIME'] = data_hourly.loc[X_test3['index'], 'DATE_TIME'].values
X_test3.to_csv('predicted_data_after_3_hours.csv', index=False)

[ ] X_test4['4_hour '] = Y_test4
X_test4['4_hour pred'] = Y_pred4
X_test4= X_test4.reset_index()
X_test4['DATE_TIME'] = data_hourly.loc[X_test4['index'], 'DATE_TIME'].values
X_test4.to_csv('predicted_data_after_4_hours.csv', index=False)

[ ] X_test1 = X_test1.sort_values(by='DATE_TIME')
X_test2 = X_test2.sort_values(by='DATE_TIME')
X_test3 = X_test3.sort_values(by='DATE_TIME')
X_test4 = X_test4.sort_values(by='DATE_TIME')

```

Visualizing Actual vs Predicted Values

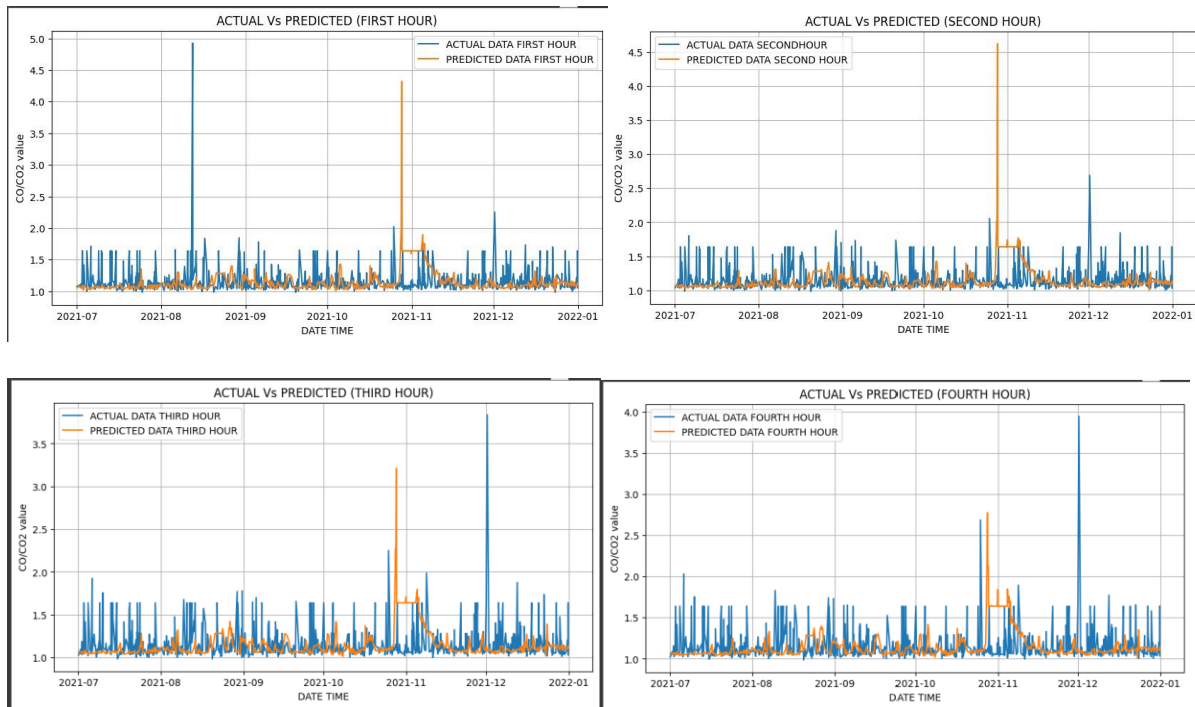
a. First Hour Prediction

We plot the actual vs. predicted CO/CO2 values for the first hour.

```

▶ plt.figure(figsize=(10,5))
plt.plot(X_test1['DATE_TIME'],Y_test1,label="ACTUAL DATA FIRST HOUR")
plt.plot(X_test1['DATE_TIME'],X_test1['1_hour pred'],label="PREDICTED DATA FIRST HOUR")
plt.legend()
plt.grid()
plt.title("ACTUAL Vs PREDICTED (FIRST HOUR)")
plt.xlabel('DATE TIME')
plt.ylabel('CO/CO2 value')
plt.show

```



Final predicted CO:CO2 Ratio:

Final Predicted co:co2 ratios on hourly basis are-

```
Predicted CO:CO2 Ratio:
1.0994441568807927
1.1551640831742378
1.1004358101516756
1.1328431946827338
1.1280065254957812
1.118902112454273
1.1355513220107551
1.1092018252862716
1.090921177995059
1.1358181971385712
1.090921177995059
1.118902112454273
1.0814739745089625
1.0946406589997004
1.1076042291920303
1.0955141820934684
1.1145389674390096
1.112306377601698
1.1558189497960814
1.1432245525945661
1.144553601085682
1.0888550134600927
```

1.0955141820934684
1.091433284817469
1.1009369533392654
1.1284340496712089
1.1585644585184813
1.1501131785574772
1.1261068230394764
1.154946807983529
1.1355513220107551
1.1688041236438067
1.1146780381622328
1.1572477421109755
1.0955141820934684
1.112306377601698
1.144553601085682
1.1298441149604834
1.1585644585184813
1.1154634446911238
1.1358181971385712
1.1009369533392654
1.0930493554264382
1.152477121865461
1.125264284323573
1.1006249193861364
1.1423441928602236
1.090921177995059
1.1261068230394764
1.1709226719514108
1.0930842964570489
1.1723121200385256
1.152477121865461
1.151983577276385
1.100406324602447
1.090095554197275
1.118902112454273
1.0994441568807927
1.1280065254957812
1.1312063359862774
1.144553601085682
1.1261068230394764
1.1423441928602236
1.1076042291920303
1.1006249193861364
1.1432245525945661
1.0888550134600927
1.1115571064013634
1.1328431946827338
1.1004358101516756
1.0829124043956895
1.1006249193861364
1.118902112454273
1.132068201169495
1.0930493554264382
1.1223049328916992
1.151983577276385
1.165252002227406
1.0719787090946828
1.0888550134600927
1.0960565905600503

YOLO

Introduction to YOLO:

YOLO (You Only Look Once) is a groundbreaking object detection system that simplifies the traditional approach to object detection. Object detection involves identifying objects within an image and drawing bounding boxes around them. Unlike previous methods that required multiple stages and were computationally expensive, YOLO transforms object detection into a single, end-to-end regression problem.

Core Concept:

The core idea of YOLO is to apply a single convolutional neural network (CNN) to the full image. This CNN divides the image into an $S \times S$ grid and, for each grid cell, predicts bounding boxes, confidence scores for those boxes, and class probabilities for each box. The confidence score reflects how confident the model is that the box contains an object and the accuracy of the box's predicted location. This single-stage detection process significantly reduces computation time, making YOLO extremely fast compared to its predecessors.

History and Evolution of YOLO:

YOLOv1 (2015):

YOLOv1 was introduced by Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi in 2015. Their paper, "You Only Look Once: Unified, Real-Time Object Detection," introduced a new paradigm in object detection.

- **Motivation and Design:** Traditional object detection systems like R-CNN, Fast R-CNN, and Faster R-CNN involve multiple stages, including region proposal and classification, which are computationally intensive. YOLOv1 proposed a single-stage detection system that predicts bounding boxes and class probabilities directly from full images in one evaluation.
- **Architecture:** YOLOv1's architecture consists of 24 convolutional layers followed by 2 fully connected layers. It uses a relatively simple and lightweight network, making it capable of real-time detection.
- **Performance:** YOLOv1 was revolutionary for its speed, capable of processing 45 frames per second on a standard GPU. However, it struggled with small objects and precise localization due to its coarse grid structure.

YOLOv2 (2016) - "YOLO9000":

YOLOv2, also known as YOLO9000, was introduced in 2016 with several enhancements.

- **Improvements:** YOLOv2 introduced batch normalization on all convolutional layers, a high-resolution classifier, and multi-scale training, improving the model's accuracy and robustness.
- **Architecture Changes:** YOLOv2 used a new backbone network called Darknet-19, which includes 19 convolutional layers and 5 max-pooling layers.

- **YOLO9000:** This version introduced YOLO9000, which can detect over 9000 object categories by combining the COCO dataset and ImageNet dataset, allowing the model to perform detection and classification simultaneously.

YOLOv3 (2018):

YOLOv3 brought significant enhancements to the YOLO architecture.

- **Further Enhancements:** YOLOv3 introduced Darknet-53, a more complex and deeper backbone network with 53 convolutional layers, improving feature extraction.
- **Multi-Scale Predictions:** YOLOv3 predicts bounding boxes at three different scales, which helps in detecting objects of various sizes more accurately.
- **Class Prediction:** Instead of using softmax for classification, YOLOv3 uses independent logistic classifiers for each label, allowing it to handle overlapping labels better.

YOLOv4 (2020):

YOLOv4, released in 2020, incorporated cutting-edge techniques to further improve performance.

- **Advanced Techniques:** YOLOv4 uses CSPDarknet53 as the backbone, PANet for path aggregation, and SPP (Spatial Pyramid Pooling) for better receptive fields. It also includes various bag-of-freebies and bag-of-specials techniques to improve training and inference.
- **Optimization:** YOLOv4 balances speed and accuracy, making it suitable for a wide range of real-time applications.

YOLOv5 (2020):

Developed by Ultralytics, YOLOv5 introduced several new features and improvements.

- **Usability and Performance:** YOLOv5 emphasizes ease of use, modularity, and deployment capabilities. It is implemented in PyTorch, making it more accessible to the PyTorch community.
- **Variants:** YOLOv5 is available in different sizes (YOLOv5s, YOLOv5m, YOLOv5l, YOLOv5x), catering to different performance and speed requirements.

YOLOv6, YOLOv7, and Beyond:

- **Continuous Evolution:** Later versions of YOLO have continued to build on the foundations laid by earlier versions, integrating newer techniques from deep learning research to enhance performance, accuracy, and efficiency.

YOLOv8 (You Only Look Once version 8):

It is an advanced object detection model known for its real-time performance and high accuracy. It builds on the success of its predecessors, incorporating state-of-the-art techniques to improve detection speed and precision. YOLOv8 uses a single neural network to predict bounding boxes and class probabilities directly from full images in

one evaluation. It leverages a streamlined architecture and optimized anchor boxes, making it efficient for various applications, from autonomous driving to surveillance. Its flexibility and robustness make it a popular choice for both academic research and industrial deployment in computer vision tasks.

Key Features and Innovations:

Single-Stage Detection:

YOLO's key innovation is framing object detection as a single regression problem. Traditional object detection methods involve multiple stages, such as region proposal, region refinement, and classification, which are computationally expensive and slow. YOLO, by contrast, applies a single neural network to the full image, which predicts bounding boxes and class probabilities simultaneously, drastically simplifying and speeding up the detection process.

Grid-Based Approach:

YOLO divides the input image into an $S \times S$ grid. Each grid cell is responsible for predicting a certain number of bounding boxes and their associated confidence scores, along with class probabilities. This grid-based approach allows YOLO to detect multiple objects within an image efficiently. The confidence score indicates the likelihood that the bounding box contains an object and the accuracy of the bounding box's coordinates.

Real-Time Performance:

One of YOLO's most notable advantages is its real-time performance. Thanks to its single-stage detection process, YOLO can process images at high frame rates, making it suitable for applications that require quick response times, such as autonomous driving, video surveillance, and robotics. For instance, YOLOv1 could process images at 45 frames per second on a standard GPU, with later versions achieving even faster speeds.

Unified Architecture:

YOLO's architecture is unified and streamlined, using a single neural network for detection. This unification leads to faster and more efficient computations compared to traditional methods, which often require separate models for region proposal and classification.

Multi-Scale Detection:

Starting from YOLOv3, the model includes multi-scale detection. This means that the model predicts bounding boxes at three different scales, helping it detect objects of varying sizes more accurately. This is particularly useful for detecting smaller objects that might be missed by single-scale detectors.

Community and Ecosystem:

YOLO has a strong and active community, with extensive resources available for researchers and developers. Numerous pre-trained models, open-source implementations, and tutorials are available, making it easier for new users to get started and for experienced users to optimize and adapt the models to their specific needs.

Applications of YOLO

- **Autonomous Vehicles:** In the realm of autonomous driving, YOLO's real-time detection capabilities are critical. Self-driving cars need to recognize and react to various objects on the road, such as other vehicles, pedestrians, traffic signs, and obstacles. YOLO's speed and accuracy enable it to process visual data in real-time, making it an ideal choice for this application.
- **Surveillance and Security:** YOLO is widely used in surveillance systems to detect and track intruders, monitor crowds, and ensure security in sensitive areas. Its ability to process video streams in real time allows for immediate response to potential threats.
- **Medical Imaging:** In healthcare, YOLO is employed to detect abnormalities in medical images, such as tumours, lesions, or fractures. This assists doctors in diagnosis and treatment planning. YOLO's accuracy and speed make it a valuable tool for analysing medical imagery quickly and effectively.
- **Retail and Inventory Management:** YOLO helps in identifying and tracking products in retail environments, improving inventory management and customer experience. For example, it can be used to monitor stock levels on shelves, detect misplaced items, and enhance automated checkout systems.
- **Robotics:** Robots equipped with YOLO-based vision systems can navigate environments, avoid obstacles, and interact with objects. This is particularly useful in industrial automation, where robots need to perform tasks such as sorting, picking, and assembling with high precision and efficiency.

Challenges and Future Directions

- **Small Object Detection:** Despite its many strengths, YOLO has historically struggled with detecting small objects, particularly when they are located close to larger objects. Research is ongoing to improve YOLO's capability to detect smaller objects more accurately, which would enhance its utility in applications where small object detection is critical.
- **Real-Time Performance:** Balancing speed and accuracy continues to be a significant challenge. While YOLO is already fast, achieving higher accuracy without sacrificing speed is an ongoing area of research. Advances in hardware, such as more powerful GPUs and specialized AI accelerators, as well as optimization techniques like model pruning and quantization, are crucial for further improvements.
- **Generalization:** Ensuring that YOLO models generalize well across diverse datasets and real-world scenarios is another important area of research. Models trained on specific datasets might not perform well on different types of images or in different

environments. Techniques such as data augmentation, transfer learning, and domain adaptation are being explored to improve generalization.

- **Integration with Other Technologies:** Combining YOLO with emerging technologies like edge computing, 5G, and AI accelerators will open up new possibilities for real-time applications. For example, edge computing can bring computation closer to the data source, reducing latency and improving real-time performance for applications like autonomous driving and surveillance.
- **Explainability and Trustworthiness:** As YOLO is used in more safety-critical applications, enhancing the interpretability and trustworthiness of its models becomes increasingly important. Researchers are exploring methods to make YOLO's decisions more transparent and to ensure that the models behave reliably under various conditions. This includes developing techniques for model explainability, robustness to adversarial attacks, and ensuring fairness in object detection.

YOLOv8: Next-Generation Object Detection

Introduction

YOLOv8 represents the latest iteration in the You Only Look Once (YOLO) series, building on the strengths and lessons learned from its predecessors. As with previous versions, YOLOv8 aims to deliver state-of-the-art object detection performance, balancing speed, accuracy, and ease of use. YOLOv8 incorporates cutting-edge techniques and optimizations to push the boundaries of what is possible in real-time object detection.

Key Features of YOLOv8

- **Architecture:** YOLOv8 continues to evolve the neural network architecture to improve performance. Here are the main architectural features and enhancements:
- **Backbone Network:** YOLOv8 uses a new and improved backbone network designed for better feature extraction. This backbone is deeper and more complex than those in earlier versions, incorporating advanced convolutional layers, normalization techniques, and activation functions.
- **Neck and Head:** The neck of YOLOv8 is designed to enhance the flow of information between the backbone and the detection head. It uses advanced feature pyramid networks (FPN) and path aggregation networks (PAN) to improve multi-scale feature fusion. The detection head, responsible for predicting bounding boxes and class probabilities, has also been optimized for better accuracy and efficiency.
- **Anchors and Anchor-Free Detection:** YOLOv8 introduces improvements in anchor generation and also explores anchor-free detection mechanisms. These changes aim to simplify the model and improve its ability to detect objects of various shapes and sizes.

Training Techniques

YOLOv8 employs several advanced training techniques to improve model performance:

- **Data Augmentation:** Extensive data augmentation strategies are used to improve the model's ability to generalize. Techniques such as mosaic augmentation, mixup, and CutMix are employed to create diverse training samples.
- **Label Smoothing:** This technique helps to regularize the model by smoothing the labels during training, which can reduce overfitting and improve generalization.
- **Advanced Loss Functions:** YOLOv8 uses sophisticated loss functions, such as CIOU (Complete Intersection over Union) loss for bounding box regression and focal loss for classification. These loss functions are designed to provide better gradients and improve the convergence of the model.
- **Self-Adversarial Training:** This technique involves training the model to be robust against adversarial examples by incorporating adversarial perturbations during training.

Performance Optimization

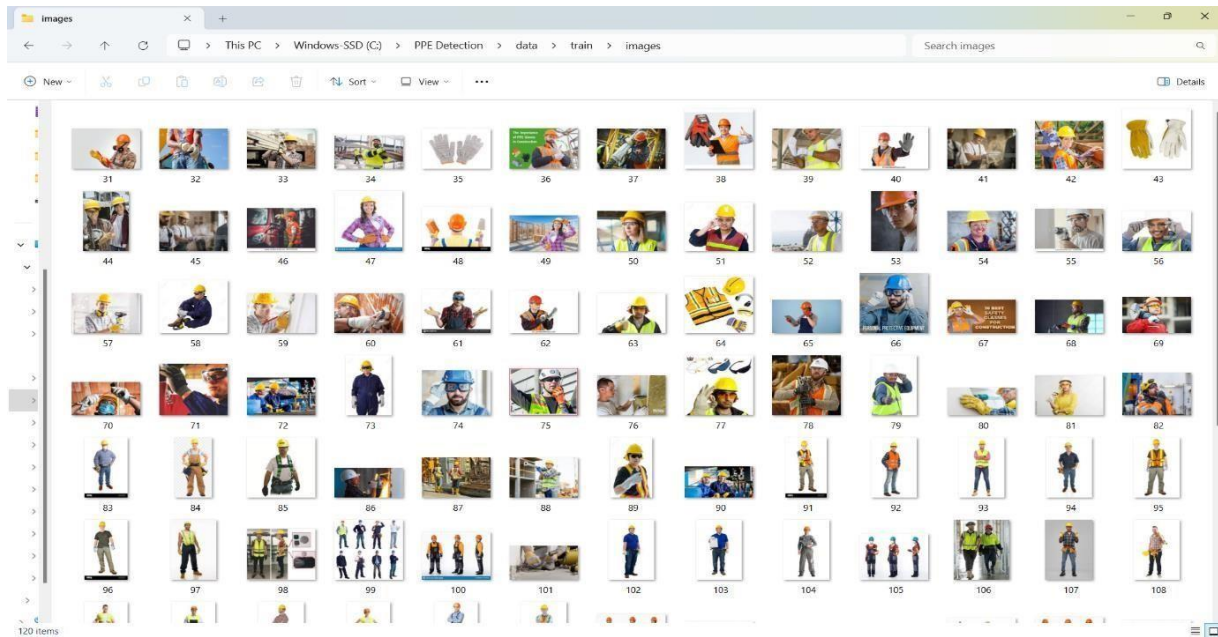
YOLOv8 is designed for high performance, both in terms of speed and accuracy. Key optimizations include:

- **Quantization and Pruning:** These techniques are used to reduce the size of the model and improve inference speed without significantly sacrificing accuracy.
- **Model Scaling:** YOLOv8 comes in various sizes, such as YOLOv8s (small), YOLOv8m (medium), YOLOv8l (large), and YOLOv8x (extra-large). This allows users to choose a model that best fits their resource constraints and performance requirements.
- **Efficient Computation:** YOLOv8 incorporates efficient layer designs and tensor operations to maximize the utilization of modern hardware, such as GPUs and AI accelerators.

Training Process:

Training YOLOv8 involves the following steps

Data Preparation: Preparing a diverse and representative dataset is crucial. Data augmentation techniques are applied to increase the variety of training samples.



Model Initialization: The model is initialized with pre-trained weights or trained from scratch, depending on the availability of a suitable pre-trained model.

Optimization: The model is trained using optimization algorithms like Adam or SGD. Learning rate scheduling, weight decay, and other regularization techniques are employed to ensure stable and efficient training.

Validation and Testing: The model is periodically evaluated on a validation set to monitor its performance. Hyperparameters are tuned based on the validation results to achieve the best performance.

CVAT (Computer Vision Annotation Tool)

CVAT (Computer Vision Annotation Tool) is an open-source, web-based tool developed by Intel for annotating images and videos. It provides a user-friendly interface and robust functionality for creating annotations that are essential for training machine learning models, especially in computer vision tasks such as object detection, image segmentation, and image classification.

File	Edit	View
0	0.646617	0.139474 0.187970 0.173684
3	0.475564	0.178947 0.154135 0.115789
3	0.633459	0.863158 0.093985 0.147368
2	0.689850	0.576316 0.266917 0.542105

Importing Libraries

“from ultralytics import YOLO”

- **ultralytics package:** This package is used for advanced deep learning tasks, particularly with the YOLO (You Only Look Once) object detection models. YOLO models are known for their speed and accuracy in object detection tasks.

Loading the Model

“model = YOLO('yolov8m.pt')”

- **Loading a pre-trained model:** This line initializes a YOLO model using the medium-sized pre-trained model weights (yolov8m.pt). The YOLO class from the ultralytics package is used to load the model. yolov8m.pt is a specific version of the YOLOv8 model that balances performance and computational efficiency.

Training the Model

“model.train(data='/kaggle/input/personal-protection-equipment/data.yaml', epochs=50)”

- **Training the model:** This command starts the training process for the loaded YOLOv8 model with the specified configuration. Here's a detailed breakdown of the parameters:
- **data='/kaggle/input/personal-protection-equipment/data.yaml':** This specifies the path to the data configuration file. The data.yaml file contains information about the dataset, such as the paths to the training and validation data, the classes, and other relevant settings.

```
from ultralytics import YOLO
model = YOLO('yolov8m.pt')
model.train(data='/kaggle/input/personal-protection-equipment/data.yaml', epochs=50)
```

Downloading <https://github.com/ultralytics/assets/releases/download/v8.2.0/yolov8m.pt> to 'yolov8m.pt'...

100% [██████████] 49.7M/49.7M [00:00<00:00, 233MB/s]

Ultralytics YOLOv8.2.35 Python-3.10.13 torch-2.1.2 CUDA:0 (Tesla T4, 15102MiB)

engine/trainer: task=detect, mode=train, model=yolov8m.pt, data=/kaggle/input/personal-protection-equipment/data.yaml, epochs=50, time=None, patience=100,

Downloading <https://ultralytics.com/assets/Arial.ttf> to '/root/.config/Ultralytics/Arial.ttf'...

100% [██████████] 755k/755k [00:00<00:00, 13.5MB/s]

2024-06-19 07:16:44,448 INFO util.py:124 -- Outdated packages:

ipywidgets==7.7.1 found, needs ipywidgets>=8

Run 'pip install -U ipywidgets', then restart the notebook server for rich notebook output.

2024-06-19 07:16:45,858 INFO util.py:124 -- Outdated packages:

ipywidgets==7.7.1 found, needs ipywidgets>=8

Run 'pip install -U ipywidgets', then restart the notebook server for rich notebook output.

Overriding model.yaml nc=80 with nc=8

	from	n	params	module	arguments
0	-1	1	1392	ultralytics.nn.modules.conv.Conv	[3, 48, 3, 2]
1	-1	1	41664	ultralytics.nn.modules.conv.Conv	[48, 96, 3, 2]
2	-1	2	111360	ultralytics.nn.modules.block.C2f	[96, 96, 2, True]
3	-1	1	166272	ultralytics.nn.modules.conv.Conv	[96, 192, 3, 2]
4	-1	4	813312	ultralytics.nn.modules.block.C2f	[192, 192, 4, True]
5	-1	1	664320	ultralytics.nn.modules.conv.Conv	[192, 384, 3, 2]
6	-1	4	3248640	ultralytics.nn.modules.block.C2f	[384, 384, 4, True]
7	-1	1	1991808	ultralytics.nn.modules.conv.Conv	[384, 576, 3, 2]
8	-1	2	3985920	ultralytics.nn.modules.block.C2f	[576, 576, 2, True]
9	-1	1	831168	ultralytics.nn.modules.block.SPPF	[576, 576, 5]
10	-1	1	0	torch.nn.modules.upsampling.Upsample	[None, 2, 'nearest']
11	[-1, 6]	1	0	ultralytics.nn.modules.conv.Concat	[1]
12	-1	2	1993728	ultralytics.nn.modules.block.C2f	[960, 384, 2]
13	-1	1	0	torch.nn.modules.upsampling.Upsample	[None, 2, 'nearest']

- **epochs=50:** This sets the number of training epochs to 50. An epoch is one complete pass through the entire training dataset. More epochs can lead to better model performance but also require more training time.

```

50 epochs completed in 0.563 hours.
Optimizer stripped from runs/detect/train/weights/last.pt, 52.0MB
Optimizer stripped from runs/detect/train/weights/best.pt, 52.0MB

Validating runs/detect/train/weights/best.pt...
Ultralytics YOLOv8.2.35 Python-3.10.13 torch-2.1.2 CUDA:0 (Tesla T4, 15102MiB)
Model summary (fused): 218 layers, 25844392 parameters, 0 gradients, 78.7 GFLOPs

```

Class	Images	Instances	Box(P)	R	mAP50	mAP50-95)
all	326	1514	0.964	0.932	0.975	0.815
Boots	246	541	0.98	0.91	0.972	0.755
Helmet	232	273	0.964	0.952	0.972	0.757
Person	246	302	0.941	0.911	0.974	0.889
Vest	307	398	0.972	0.955	0.983	0.858

```

Speed: 0.3ms preprocess, 11.9ms inference, 0.0ms loss, 12.2ms postprocess per image
Results saved to runs/detect/train

```

Run history:



Run summary:

lr/pg0	2e-05
lr/pg1	2e-05
lr/pg2	2e-05
metrics/mAP50(B)	0.97525
metrics/mAP50-95(B)	0.81455
metrics/precision(B)	0.96427
metrics/recall(B)	0.932
model/GFLOPs	79.088
model/parameters	25860952
model/speed_PyTorch(ms)	10.505
train/box_loss	0.53139
train/cls_loss	0.26913
train/df_l_loss	0.94245
val/box_loss	0.75308
val/cls_loss	0.37507
val/df_l_loss	1.06293

Explanation of data.yaml

The data.yaml file is crucial as it defines the paths to the training, validation, and testing datasets, as well as the class labels. Here's an example of what the data.yaml file looks like and an explanation of each part:

Dataset Paths

“train: ../train/images

val: ../valid/images

test: ../test/images”

- **train: ../train/images:** This specifies the relative path to the directory containing the training images. These images are used to train the model.
- **val: ../valid/images:** This specifies the relative path to the directory containing the validation images. These images are used to evaluate the model's performance during training, helping to prevent overfitting.

- **test: ../test/images:** This specifies the relative path to the directory containing the test images. These images are used to evaluate the final model performance after training is complete.

Number of Classes

“nc: 8”

- **nc: 8:** This specifies the number of classes in the dataset. In this case, there are 8 distinct classes that the model needs to recognize.

Class Names

“names: ['Boots', 'Ear-protection', 'Glass', 'Glove', 'Helmet', 'Mask', 'Person', 'Vest']”

- **names: [...]:** This provides a list of class names corresponding to the classes in the dataset. Each name represents a category that the model will be trained to detect. The list includes:
 - Boots
 - Ear-protection
 - Glass
 - Glove
 - Helmet
 - Mask
 - Person
 - Vest

Importance of data.yaml

The data.yaml file plays a vital role in the training process:

- **Dataset Organization:** It provides a structured way to organize and reference the dataset. By defining paths to training, validation, and test sets, it ensures that the model knows where to find the necessary data.
- **Class Mapping:** The names section maps numerical labels to human-readable class names. This is crucial for interpreting the model's outputs and for the training process, as the model needs to know what each label represents.
- **Flexibility:** Using a configuration file allows for easy modification and experimentation. You can change dataset paths, add new classes, or adjust other parameters without altering the training script.
- **Consistency:** It ensures consistency across different training runs. By keeping the dataset paths and class labels in a centralized configuration file, you reduce the risk of errors that might occur if these parameters were hardcoded in multiple places.

PPE (Personal Protective Equipment Detection Website)

Website is for the safety equipment check of the employees entering blast furnace through image detection using YOLO. Model is trained using YOLOv8 results are stored in best.pt. The website checks the basic parameters like HELMET, GOGGLES, JACKET, GLOVES, FOOTWEAR.

- If the all the parameters are passed by the employee it gives a green signal and the employees are good to go inside otherwise they need to rectify.
- The website consists of a Introduction Page of RINL followed by the upload page where the employee can take and upload image for detection.

Software Requirements Overview

HTML:

Hypertext Markup Language (HTML) is a markup language used to create web pages and web applications. Created by Tim Berners-Lee in 1990, HTML is straightforward to learn and modify, featuring various formatting tags that help create effective presentations. It is platform-independent, allowing it to be displayed on any device or operating system.

Key Features:

- Easy to learn and use
- Platform-independent
- Ability to add images, videos, and audio
- Capability to include hypertext within text
- Flexible design options
- Option to add links to web pages

CSS:

Cascading Style Sheets (CSS) is a language used to control the presentation of HTML documents. Invented by Håkon Wium Lie, CSS defines the look of web pages, including colors, fonts, spacing, and element positioning. It enhances web presentation capabilities by making web page code less bulky and more vibrant compared to using HTML attributes alone.

Key Features:

- Time-saving: Write CSS once and reuse across multiple HTML pages
- Easy maintenance: Global changes through the style sheet
- Search engine-friendly: Clean coding technique
- Superior styles: Wider array of attributes than HTML
- Offline browsing: Local storage using an offline cache

Python:

Python is a versatile, high-level programming language known for its readability and simplicity. It supports multiple programming paradigms and is widely used in web development, data analysis, artificial intelligence, scientific computing, and more.

Key Features:

- Easy to learn and use with simple syntax
- Versatile: Suitable for web development, data analysis, AI, machine learning, scientific computing, and automation

- Extensive standard library: Includes modules and packages for various tasks
- Cross-platform compatibility: Works on Windows, macOS, and Linux
- Integration capabilities: Easily integrates with other languages and technologies
- Supports object-oriented and procedural programming

Flask Framework:

Flask is a lightweight, micro web framework for Python designed for simplicity and flexibility. It allows for the creation of powerful web applications with minimal setup, making it popular among both beginners and advanced developers.

Key Features:

- Lightweight and flexible: Essential components with optional additional tools and libraries
- Simplicity and ease of use: Intuitive API for quick application setup
- Built-in development server and debugger: Facilitates testing and debugging
- Modular design: Extensions for added functionality like form validation, authentication, and database integration
- RESTful request handling: Ideal for developing RESTful APIs and services
- Strong community and ecosystem: Rich in extensions and plugins
- Extensible: Customizable to meet specific needs

Advantages of Using Flask for Object Detection Applications

- **Rapid Development:** Flask's simplicity and ease of use allow developers to quickly prototype and develop web applications. This is particularly beneficial for object detection projects where rapid iteration is often required.
- **Scalability:** Flask applications can be scaled up as needed. The framework allows for easy integration with other technologies, such as cloud services, which can help scale the application to handle more requests and process larger datasets.
- **Customizability:** Flask's minimalistic nature means you can add only the components you need. For object detection applications, you can integrate Flask with deep learning frameworks like YOLOv8, without unnecessary bloat.
- **Community Support and Documentation:** Flask has a large community of developers and extensive documentation. This means you can find numerous resources, tutorials, and third-party libraries to help with your development process.
- **Flexibility in Design:** With Flask, you have the flexibility to design your application architecture as you see fit. Whether you need a simple API to serve object detection results or a more complex web interface, Flask can accommodate your needs.

HTML Structure

Document Type and Metadata

- **<!DOCTYPE html>:** Declares the document type and version of HTML (HTML5 in this case).
- **<html lang="en">:** Sets the language of the document to English.

- **<head>**: Contains metadata about the document, including character set and title.
- **<meta charset="UTF-8">**: Sets the character encoding for the document to UTF-8.
- **<title>**: Sets the title of the webpage, which appears in the browser tab.
- **<style>**: Contains CSS styles for the webpage.

Body Content

- **<body>**: Contains the main content of the document.
- **<h1>**: Header element for the main title of the page.
- **<div class="input-option">**: A container for input options (radio buttons) to choose between using a webcam or uploading an image.
- **<input type="radio">**: Radio buttons for selecting input type (webcam or file upload).
- **<label>**: Labels for the radio buttons.
- **<video id="video" autoplay>**: Video element to display the webcam feed.
- **<input type="file" id="fileInput" accept="image/*" style="display:none;">**: File input for uploading an image, initially hidden.
- **<button id="start">Start Analysis</button>**: Button to start the PPE detection analysis.
- **<div id="results">**: Container for displaying the detection results.
- **<script>**: Contains JavaScript code for handling user interactions and processing.

CSS Styles

- **General Styles**: Set basic styles for the body, headers, buttons, and results div.
- **Visibility**: Video element is initially hidden, and the file input is displayed based on the selected option.
- **Buttons**: Styled for better user experience with hover effects.
- **Results**: Styled to display detection results clearly.

JavaScript Functionality

Element References and Event Listeners

- **Element References**: References to HTML elements for later manipulation.
- **Update Input Visibility**: Function to show/hide video or file input based on the selected option. It also handles starting/stopping the webcam stream.
- **Event Listeners**: Added to radio buttons to update input visibility and to the start button to initiate analysis.

Capturing Frames and Handling File Uploads

- **captureFrame()**: Captures a frame from the webcam, converts it to a blob, and sends it to the server for analysis.

Displaying Results and Speech Synthesis

- **displayResults(data):** Displays the detection results on the webpage. It also checks if all required items are detected and speaks a message accordingly.
- **speak(message):** Uses the Web Speech API to vocalize a message.

```
2  <html lang="en">
95 <body>
96   <h1>PPE Detection Interface</h1>
97   <div class="input-option">
98     <input type="radio" id="webcamOption" name="inputType" value="webcam" checked>
99     <label for="webcamOption">Use Webcam</label>
100    <input type="radio" id="fileOption" name="inputType" value="file">
101    <label for="fileOption">Upload Image</label>
102  </div>
103  <video id="video" autoplay></video>
104  <input type="file" id="fileInput" accept="image/*" style="display:none;">
105  <button id="start">Start Analysis</button>
106  <div id="results">
107    <div class="results-header">Detection Results</div>
108  </div>
109  <script>
110    const video = document.getElementById('video');
111    const startButton = document.getElementById('start');
112    const fileInput = document.getElementById('fileInput');
113    const resultsDiv = document.getElementById('results');
114    const webcamOption = document.getElementById('webcamOption');
115    const fileOption = document.getElementById('fileOption');
116    let timer;
117
118    function updateInputVisibility() {
119      if (webcamOption.checked) {
120        video.style.display = 'block';
121        fileInput.style.display = 'none';
122        navigator.mediaDevices.getUserMedia({ video: true })
123          .then(stream => video.srcObject = stream)
124          .catch(err => console.error("Error: ", err));
125      } else if (fileOption.checked) {
126        video.style.display = 'none';
127        fileInput.style.display = 'block';
128        if (video.srcObject) {
129          let tracks = video.srcObject.getTracks();
```

```

<body>
<script>
function displayResults(data) {
  console.log(data); // Here you can update the webpage to show the results
  resultsDiv.innerHTML = '<div class="results-header">Detection Results</div>';
  data.detections.forEach(detection => {
    const detectionDiv = document.createElement('div');
    detectionDiv.className = 'detection';
    detectionDiv.innerHTML = `
      <p>Class: ${detection.class}</p>
      <p class="confidence">Confidence: ${((detection.confidence * 100).toFixed(2))}%</p>
      <p class="bbox">Bounding Box: [${detection.bbox.join(', ')}]</p>
    `;
    resultsDiv.appendChild(detectionDiv);
  });

  const requiredItems = ['helmet'];
  const detectedItems = data.detections.map(d => d.class.toLowerCase());
  const allItemsDetected = requiredItems.every(item => detectedItems.includes(item));

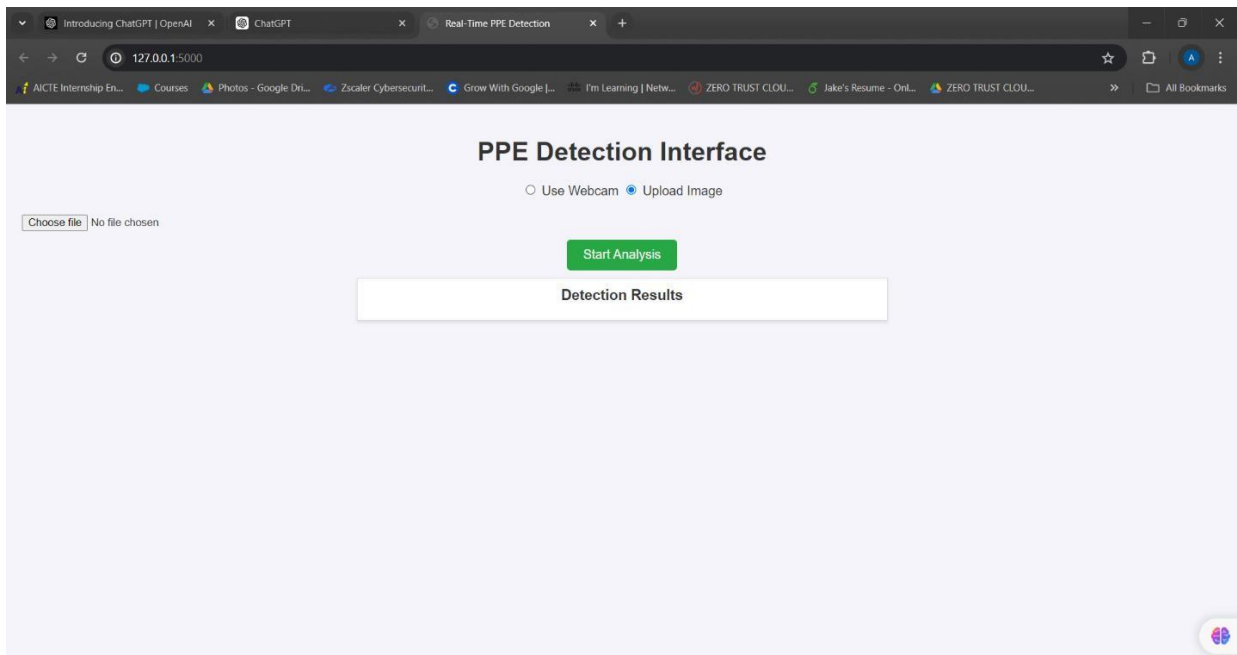
  if (allItemsDetected) {
    speak('Secure');
  } else {
    speak('Not Secure');
  }
}

function speak(message) {
  const speech = new SpeechSynthesisUtterance(message);
  speech.lang = 'en-US';
  window.speechSynthesis.speak(speech);
}

updateInputVisibility(); // Set initial visibility based on default selection
</script>
</body>

```

OUTPUT:



0: 480x640 (no detections), 1950.5ms
Speed: 8.4ms preprocess, 1950.5ms inference, 0.0ms postprocess per image at shape (1, 3, 480, 640)
127.0.0.1 - - [20/Jun/2024 21:13:36] "POST /predict HTTP/1.1" 200 -

0: 480x640 1 No-Vest, 1946.1ms
Speed: 9.4ms preprocess, 1946.1ms inference, 12.2ms postprocess per image at shape (1, 3, 480, 640)
127.0.0.1 - - [20/Jun/2024 21:13:38] "POST /predict HTTP/1.1" 200 -

0: 480x640 (no detections), 1940.5ms
Speed: 10.5ms preprocess, 1940.5ms inference, 0.0ms postprocess per image at shape (1, 3, 480, 640)
127.0.0.1 - - [20/Jun/2024 21:13:40] "POST /predict HTTP/1.1" 200 -

0: 480x640 1 No-Vest, 1955.8ms
Speed: 14.1ms preprocess, 1955.8ms inference, 0.0ms postprocess per image at shape (1, 3, 480, 640)
127.0.0.1 - - [20/Jun/2024 21:13:42] "POST /predict HTTP/1.1" 200 -

0: 480x640 (no detections), 1979.8ms
Speed: 14.5ms preprocess, 1979.8ms inference, 3.0ms postprocess per image at shape (1, 3, 480, 640)
127.0.0.1 - - [20/Jun/2024 21:13:44] "POST /predict HTTP/1.1" 200 -

0: 480x640 (no detections), 1972.0ms
Speed: 9.5ms preprocess, 1972.0ms inference, 4.1ms postprocess per image at shape (1, 3, 480, 640)
127.0.0.1 - - [20/Jun/2024 21:13:46] "POST /predict HTTP/1.1" 200 -



Backend for PPE Detection App

This Python code sets up a Flask web application that allows users to upload images, performs object detection using a YOLO model, and returns the results as JSON. It also

includes basic configurations and setup. Here's a detailed explanation of each part of the code:

1. Importing Libraries

- **Flask:** A lightweight WSGI web application framework for Python.
- **render_template:** Renders an HTML template.
- **request:** Handles incoming requests.
- **jsonify:** Converts Python dictionaries to JSON responses.
- **cv2:** OpenCV library for image processing.
- **numpy:** Library for numerical operations, particularly with arrays.
- **YOLO:** Ultralytics YOLO library for object detection.
- **CORS:** Enables Cross-Origin Resource Sharing (CORS) to allow requests from different origins.

2. Application Initialization

- **app = Flask(name):** Initializes the Flask application.
- **CORS(app):** Enables CORS for all routes, allowing the app to handle requests from different domains.

3. Loading the YOLO Model

- **YOLO('path_to_model.pt'):** Loads the trained YOLO model from the specified file path. This model is used for object detection on the uploaded images.

4. Route: /

- **@app.route('/'):** Defines a route for the root URL.
- **index():** Function that handles requests to the root URL.
- **return render_template('index2.html'):** Renders and returns the index2.html template as the response.

5. Route: /predict

- **@app.route('/predict', methods=['POST']):** Defines a route for handling POST requests to /predict.
- **predict():** Function that handles prediction requests.
- **if 'image' not in request.files:** Checks if the 'image' file is included in the request. If not, returns an error JSON response.
- **file = request.files['image']:** Retrieves the uploaded image file from the request.
- **img = np.frombuffer(file.read(), np.uint8):** Converts the file data to a NumPy array.
- **img = cv2.imdecode(img, cv2.IMREAD_COLOR):** Decodes the NumPy array into an image using OpenCV.
- **results = model(img):** Runs the YOLO model on the uploaded image to perform object detection.
- **detections = []:** Initializes an empty list to store detection results.
- **for result in results:** Iterates through the results returned by the model.

- **for box in result.bboxes:** Iterates through the detected boxes in each result.
- **detections.append(...):** Appends detection details (class, confidence, bounding box) to the detections list.
- **return jsonify({'detections': detections}):** Returns the detection results as a JSON response.

6. Main Execution

- **if name == 'main':** Ensures that the app runs only if the script is executed directly (not imported as a module).
- **app.run(debug=True):** Runs the Flask app with debug mode enabled, which provides detailed error messages and auto-reloads the server on code changes.

```

1  from flask import Flask, render_template, request, jsonify
2  import cv2
3  import numpy as np
4  from ultralytics import YOLO
5  from flask_cors import CORS
6
7  app = Flask(__name__)
8  CORS(app) # Enable CORS for all routes
9
10 # Load the trained YOLOv8 model
11 model = YOLO('C:\\Users\\Akhil\\steel plant\\ppe_detection_app\\best.pt')
12
13 @app.route('/')
14 def index():
15     return render_template('index2.html')
16
17 @app.route('/predict', methods=['POST'])
18 def predict():
19     if 'image' not in request.files:
20         return jsonify({'error': 'No image uploaded'}), 400
21
22     file = request.files['image']
23     img = np.frombuffer(file.read(), np.uint8)
24     img = cv2.imdecode(img, cv2.IMREAD_COLOR)
25
26     # Run inference on the image
27     results = model(img)
28
29     detections = []
30     for result in results:
31         for box in result.bboxes:
32             detections.append({
33                 'class': result.names[int(box.cls)],
34                 'confidence': box.conf.item(),
35                 'bbox': box.xyxy.tolist()
36             })
37
38     return jsonify({'detections': detections})
39
40 if __name__ == '__main__':
41     app.run(debug=True)
42

```

CONCLUSION-

Predicting co:co2 Ratio using Machine learning-Python

In this project, we aimed to predict the CO ratio using a Random Forest model by first converting datetime information into an hourly basis. The data was loaded from an Excel file and inspected to ensure all necessary columns were present. The 'Date' and 'Time' columns were combined into a single 'Datetime' column, which was then converted into a pandas-recognized datetime format. This datetime information was further transformed into milliseconds since epoch, and aggregated on an hourly basis by converting these milliseconds into hours.

For the predictive modeling, the 'HOUR' column was used as the feature variable and the 'CORatio' as the target variable. The dataset was split into training and testing sets, with 80% of the data used for training and 20% reserved for testing. A Random Forest Regressor was then trained on the training data to predict the CO ratio.

The performance of the model was evaluated using Mean Absolute Error (MAE), Mean Squared Error (MSE), and R-squared (R^2) metrics. These metrics helped in understanding how well the model was able to predict the CO ratio based on the provided hourly data.

After training the model, predictions were made on the testing data. The predicted CORatio values were printed to the console for further analysis. To visualize the performance, a plot was created comparing the original CO ratio values against the predicted values, using hours as the x-axis.

YOLO

The presented project harnesses the capabilities of the YOLO (You Only Look Once) object detection model to annotate and track objects within a video stream. By employing a pre-trained YOLO model, the code efficiently identifies and labels objects of interest, including heads, helmets, and persons, in real-time video footage. This project serves as a testament to the practical application of cutting-edge computer vision techniques in addressing complex challenges across various domains.

At its core, the YOLO model offers a streamlined approach to object detection by dividing the image into a grid and predicting bounding boxes and class probabilities for each grid cell. This enables rapid and accurate detection of multiple objects within a single pass through the neural network. By integrating YOLO with video processing techniques, the project showcases how advanced machine learning algorithms can be seamlessly integrated into real-world applications.

The significance of this project lies in its ability to automate and streamline tasks that would otherwise require manual intervention. In scenarios such as video surveillance, safety monitoring, and crowd analysis, the automated detection and tracking of objects can greatly enhance efficiency and accuracy. For instance, in surveillance applications, the ability to quickly identify and track individuals, including those wearing safety helmets, can improve security measures and response times.

Moreover, the project underscores the broader implications of computer vision technology in driving innovation and progress across various industries. From retail analytics to transportation safety systems, the ability to automatically analyze and interpret visual data opens up a myriad of opportunities for optimization and improvement. By leveraging deep learning-based object detection models like YOLO, organizations can gain valuable insights, make informed decisions, and enhance overall operational efficiency.

In conclusion, the project exemplifies the transformative potential of YOLO-based object detection in tackling real-world challenges. Through its seamless integration of advanced machine learning techniques with video processing capabilities, it demonstrates the power of computer vision in enhancing automation, efficiency, and decision-making across diverse applications and industries.