

TypeError SyntaxError SyntaxError ...

Please explain the error:

NameError: name 'results' is not defined

Hypothesis

The NameError: name 'results' is not defined indicates that the variable `results` was referenced before it was assigned a value in the current scope. Looking at the code, the `results` dictionary is intended to store the evaluation metrics for each trained model. This dictionary is populated within the `for` loop that iterates through the `models` dictionary.

The most likely cause is that the code block containing the model training and evaluation loop where `results` is defined and populated was not executed before the code block that attempts to save `results` to a JSON file. This could happen if the user ran the cells out of order, or if the cell defining and using `results` encountered an error earlier that prevented its full execution.

Since the traceback points to the line where `json.dump(results, f, indent=2)` is called, and the `results` variable is created and populated in the previous cell (ipython-input-19-aa5639c10d47), the error occurs because the cell containing the model training and the definition of `results` was either skipped or failed before the `results` dictionary was fully constructed.

Suggested Changes

Ensure that the code cell containing the model training loop and the definition of the `results` dictionary runs successfully before the cell that attempts to save `results` to a JSON file. In a Jupyter or Colab environment, this usually means running the cells sequentially from top to bottom.

The relevant code block that defines `results` is in the previous cell. To fix the NameError, you need to execute that cell first. Since the code in the notebook is already structured correctly across cells, the fix is simply to run the cells in the intended order.

However, to make the code more robust and self-contained within a single execution context if desired, you could combine the model training and the saving of results into a single code cell. This would ensure that `results` is defined before it's used for saving.

Here's the combined code for the model training and results saving:

```
import pandas as pd
import numpy as np

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
# from sklearn.svm import SVR # Commented out
from sklearn.metrics import mean_squared_error
import json # Import json module

# Assuming 'CO(GT)' is the target variable and
# after completing the previous steps (loading data, etc.)

# Make sure the target column and features are correctly identified
```

```
from google.colab import files
upload = files.upload()

import pandas as pd

df = pd.read_csv('AirQuality.csv')
```

Choose files

AirQuality.csv

- AirQuality.csv(text/csv) - 785065 bytes, last modified: 14/05/2025 - 100% done

Saving AirQualitv.csv to AirQualitv (1).csv

Generate

handling missing values for the above dataset

Close

< 1 of 1 > [Use code with caution](#)

```
# prompt: handling missing values for the above dataset

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

# Option 1: Drop rows with missing values
df_cleaned_drop = df.dropna()

# Option 2: Fill missing values with a specific value (e.g., 0)
df_filled_zero = df.fillna(0)

# Option 3: Fill missing values with the mean of the column
df_filled_mean = df.fillna(df.mean())

# Option 4: Fill missing values with the median of the column
df_filled_median = df.fillna(df.median())

# Option 5: Fill missing values with the mode of the column
df_filled_mode = df.fillna(df.mode().iloc[0])

# Option 6: Forward fill (propagate last valid observation forward)
df_filled_ffill = df.fillna(method='ffill')

# Option 7: Backward fill (propagate next valid observation backward)
df_filled_bfill = df.fillna(method='bfill')

# You can choose one of the above methods based on your data and requirements.
# For demonstration, let's use filling with the mean for numerical columns
# and forward fill for other types if needed (or handle them separately).

# Identify numerical columns
numerical_cols = df.select_dtypes(include=['number']).columns
df[numerical_cols] = df[numerical_cols].fillna(df[numerical_cols].mean())

# For non-numerical columns with missing values, you might choose a different strategy
# For example, filling with a placeholder or mode, or forward/backward fill.
# Example: Filling non-numerical missing values with a placeholder 'Unknown'
for col in df.columns:
    if df[col].isnull().any() and col not in numerical_cols:
        df[col] = df[col].fillna('Unknown')

# Verify that missing values are handled
print("\nMissing values after handling:")
print(df.isnull().sum())
```

```

Date;Time;CO(GT);PT08.S1(CO);NMHC(GT);C6H6(GT);PT08.S2(NMHC);NOx(GT);PT08.S3(NO);
dtype: int64
-----
TypeError                                 Traceback (most recent call last)
Double-click for interactive help
11
12
from google.colab import files
upload = files.upload()

import pandas as pd

# Read the CSV file, specifying the semicolon as the delimiter
df = pd.read_csv('AirQuality.csv', sep=';')

# The rest of your code for handling missing values should now work
# Check for missing values
print(df.isnull().sum())

# Option 1: Drop rows with missing values
df_cleaned_drop = df.dropna()

# Option 2: Fill missing values with a specific value (e.g., 0)
df_filled_zero = df.fillna(0)

# Option 3: Fill missing values with the mean of the column
df_filled_mean = df.fillna(df.mean(numeric_only=True)) # Added numeric_only=True for

# Option 4: Fill missing values with the median of the column
df_filled_median = df.fillna(df.median(numeric_only=True)) # Added numeric_only=True

# Option 5: Fill missing values with the mode of the column
# Mode can return multiple values for multiple modes, .iloc[0] selects the first row
df_filled_mode = df.fillna(df.mode().iloc[0])

# Option 6: Forward fill (propagate last valid observation forward)
df_filled_ffill = df.fillna(method='ffill')

# Option 7: Backward fill (propagate next valid observation backward)
df_filled_bfill = df.fillna(method='bfill')

# You can choose one of the above methods based on your data and requirements.
# For demonstration, let's use filling with the mean for numerical columns
# and forward fill for other types if needed (or handle them separately).

# Identify numerical columns
numerical_cols = df.select_dtypes(include=['number']).columns
df[numerical_cols] = df[numerical_cols].fillna(df[numerical_cols].mean())

# For non-numerical columns with missing values, you might choose a different strategy
# For example, filling with a placeholder or mode, or forward/backward fill.
# Example: Filling non-numerical missing values with a placeholder 'Unknown'
for col in df.columns:
    if df[col].isnull().any() and col not in numerical_cols:
        df[col] = df[col].fillna('Unknown')

# Verify that missing values are handled
print("\nMissing values after handling:")
print(df.isnull().sum())

```

```
target_column = 'CO(GT)'
```

```

# Select only numeric columns for modeling
# You might want to refine this selection based on your data
# For this example, let's use all numerical columns
numerical_cols = df.select_dtypes(include=[np.number])
feature_cols = numerical_cols.drop(target_column)

# Ensure the target column is in the DataFrame
if target_column not in df.columns:
    print(f"Error: Target column '{target_column}' not found")
else:
    X = df[feature_cols]
    y = df[target_column]

# Handle NaNs in features and target before modeling
# A common approach is to drop rows with missing values
# Alternatively, you could impute NaNs with a specific value
# Let's drop rows with NaNs in this example
data_for_modeling = pd.concat([X, y], axis=1)
X = data_for_modeling[feature_cols]
y = data_for_modeling[target_column]

if X.empty or y.empty:
    print("Error: DataFrame is empty after filtering")
else:
    # Split the data into training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    # Define a list of models to try
    models = {
        "Linear Regression": LinearRegression(),
        "Decision Tree Regressor": DecisionTreeRegressor(),
        "Random Forest Regressor": RandomForestRegressor(),
        "Gradient Boosting Regressor": GradientBoostingRegressor(),
        # SVR can be computationally expensive
        # "Support Vector Regressor": SVR()
    }

    # Initialize the results dictionary
    results = {}

    # Train and evaluate each model
    for name, model in models.items():
        print(f"\nTraining {name}...")
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)

        # Evaluate the model
        mse = mean_squared_error(y_test, y_pred)
        rmse = np.sqrt(mse) # Root Mean Squared Error
        r2 = r2_score(y_test, y_pred)

        results[name] = {
            "MSE": mse,
            "RMSE": rmse,
            "R2": r2
        }

    print(f"\n{name} Evaluation:")
    print(f" Mean Squared Error (MSE): {results[name]['MSE']:.4f}")
    print(f" Root Mean Squared Error (RMSE): {results[name]['RMSE']:.4f}")
    print(f" R-squared (R2): {results[name]['R2']:.4f}")

# Print a summary of the results
print("\n--- Model Comparison ---")
for name, metrics in results.items():
    print(f"{name}:")
    print(f" MSE: {metrics['MSE']:.4f}")

```



Choose files AirQuality.csv

- **AirQuality.csv**(text/csv) - 785065 bytes, last modified: 14/05/2025 - 100% done

Saving AirQuality.csv to AirQuality (2).csv

```
Date          114
Time          114
CO(GT)        114
PT08.S1(CO)   114
NMHC(GT)      114
C6H6(GT)      114
PT08.S2(NMHC) 114
NOx(GT)       114
PT08.S3(NOx)  114
NO2(GT)       114
PT08.S4(NO2)  114
PT08.S5(O3)   114
T             114
RH            114
AH            114
Unnamed: 15    9471
Unnamed: 16    9471
dtype: int64
```

Missing values after handling:

```
Date          0
Time          0
CO(GT)        0
PT08.S1(CO)   0
NMHC(GT)      0
C6H6(GT)      0
PT08.S2(NMHC) 0
NOx(GT)       0
PT08.S3(NOx)  0
NO2(GT)       0
PT08.S4(NO2)  0
PT08.S5(O3)   0
T             0
RH            0
AH            0
Unnamed: 15    9471
Unnamed: 16    9471
dtype: int64
```

```
<ipython-input-4-5ba247b872eb>:30: FutureWarning: DataFrame.fillna with 'method'
  df_filled_ffill = df.fillna(method='ffill')
```

```
<ipython-input-4-5ba247b872eb>:33: FutureWarning: DataFrame.fillna with 'method'
  df_filled_bfill = df.fillna(method='bfill')
```

```
# Optional: Find the best model based
if results: # Check if results is not
    best_model_name = max(results, ke
    print(f"\nBest model based on R2
else:
    print("\nNo model results availab
```

```
# Optional: Save results to a JSON fi
if results: # Check if results is not
    with open('model_training_results
        json.dump(results, f, indent=
    print("\nModel training results s
else:
    print("\nNo results to save to J
```

[Use code with caution](#)

► Sources

Rate this answer

Generate

duplicates



Close

< 1 of 1 > [Use code with caution](#)

prompt: duplicates

```
# Identify and handle duplicate rows
print("\nNumber of duplicate rows before handling:", df.duplicated().sum())
```

```
# Option 1: Drop duplicate rows
df_no_duplicates_drop = df.drop_duplicates()
```

```
# Option 2: Keep the first occurrence of duplicate rows
df_no_duplicates_keep_first = df.drop_duplicates(keep='first')
```

```
# Option 3: Keep the last occurrence of duplicate rows
df_no_duplicates_keep_last = df.drop_duplicates(keep='last')
```

```
# Option 4: Keep none of the duplicate rows (only unique rows remain)
df_only_unique = df.drop_duplicates(keep=False)
```

```
# You can choose one of the above methods based on your requirements.
# For most cases, keeping the first or last occurrence is common.
# Let's use dropping duplicate rows as an example
df = df.drop_duplicates()
```

```
print("Number of duplicate rows after dropping:", df.duplicated().sum())
```



```
Number of duplicate rows before handling: 113
Number of duplicate rows after dropping: 0
```

Generate

outliers



Close

< 1 of 1 > [Use code with caution](#)

```

# prompt: outliers

import numpy as np

# Function to detect outliers using IQR
def detect_outliers_iqr(data):
    q1 = np.percentile(data, 25)
    q3 = np.percentile(data, 75)
    iqr = q3 - q1
    lower_bound = q1 - 1.5 * iqr
    upper_bound = q3 + 1.5 * iqr
    outliers = data[(data < lower_bound) | (data > upper_bound)]
    return outliers

# Function to detect outliers using Z-score
def detect_outliers_zscore(data, threshold=3):
    mean = np.mean(data)
    std = np.std(data)
    z_scores = [(x - mean) / std for x in data]
    outliers = data[np.abs(z_scores) > threshold]
    return outliers

# Example: Detect outliers in a numerical column (replace 'ColumnName' with an actual column name)
# Using IQR
if 'CO(GT)' in df.columns:
    co_outliers_iqr = detect_outliers_iqr(df['CO(GT)'].dropna()) # Drop NA for outliers
    print(f"\nOutliers in 'CO(GT)' using IQR: {co_outliers_iqr}")

# Using Z-score
# Note: Z-score is sensitive to outliers itself, consider using it after handling outliers
co_outliers_zscore = detect_outliers_zscore(df['CO(GT)'].dropna()) # Drop NA for outliers
print(f"\nOutliers in 'CO(GT)' using Z-score: {co_outliers_zscore}")

# Common ways to handle outliers:

# Option 1: Remove outliers
# Example: Removing rows where 'CO(GT)' is an outlier based on IQR
if 'CO(GT)' in df.columns:
    q1 = df['CO(GT)'].quantile(0.25)
    q3 = df['CO(GT)'].quantile(0.75)
    iqr = q3 - q1
    lower_bound = q1 - 1.5 * iqr
    upper_bound = q3 + 1.5 * iqr
    df_no_outliers_iqr = df[(df['CO(GT)'] >= lower_bound) & (df['CO(GT)'] <= upper_bound)]
    print(f"\nShape after removing CO(GT) outliers using IQR: {df_no_outliers_iqr.shape}")

# Option 2: Cap or Floor (Winsorizing)
# Replace outliers with the upper or lower bounds (e.g., using IQR bounds)
if 'CO(GT)' in df.columns:
    q1 = df['CO(GT)'].quantile(0.25)
    q3 = df['CO(GT)'].quantile(0.75)
    iqr = q3 - q1
    lower_bound = q1 - 1.5 * iqr
    upper_bound = q3 + 1.5 * iqr
    df_capped = df.copy() # Work on a copy to avoid modifying the original df directly
    df_capped['CO(GT)'] = np.where(df_capped['CO(GT)'] < lower_bound, lower_bound, df_capped['CO(GT)'])
    df_capped['CO(GT)'] = np.where(df_capped['CO(GT)'] > upper_bound, upper_bound, df_capped['CO(GT)'])
    print(f"\nMin/Max of 'CO(GT)' after capping: \nMin: {df_capped['CO(GT)'].min()}, \nMax: {df_capped['CO(GT)'].max()}")

# Option 3: Transform the data (e.g., log transformation)
# This can reduce the impact of outliers in skewed data.
# Example: Applying log transformation to a column (ensure values are > 0)
if 'CO(GT)' in df.columns and (df['CO(GT)'] > 0).all(): # Check if all values are positive
    df_transformed = df.copy()
    df_transformed['CO(GT)_log'] = np.log(df_transformed['CO(GT)'])
    print("\nFirst 5 rows of 'CO(GT)_log' after log transformation:")
    print(df_transformed[['CO(GT)', 'CO(GT)_log']].head())
elif 'CO(GT)' in df.columns and not (df['CO(GT)'] > 0).all():
    print("\nLog transformation not applied to 'CO(GT)' as it contains non-positive values")

# Choose the method of outlier handling that is most appropriate for your specific data

```



```

-----
UFuncTypeError                                Traceback (most recent call last)
<ipython-input-6-dca115aa700c> in <cell line: 0>()
    24 # Using IQR
    25 if 'CO(GT)' in df.columns:
--> 26     co_outliers_iqr = detect_outliers_iqr(df['CO(GT)']).dropna() # Drop
NA for outlier detection
    27     print(f"\nOutliers in 'CO(GT)' using IQR:\n{co_outliers_iqr}")
    28

-----
⏮ 6 frames ⏭
/usr/local/lib/python3.11/dist-packages/numpy/lib/_function_base_impl.py in
# prompt: outliers

import numpy as np
import pandas as pd # Ensure pandas is imported

# Function to detect outliers using IQR
def detect_outliers_iqr(data):
    # Ensure the data is numeric before calculating percentiles
    data_numeric = pd.to_numeric(data, errors='coerce').dropna() # Coerce non-numeric
    if data_numeric.empty:
        print("Warning: Data for outlier detection is empty after dropping non-numeric")
        return pd.Series(dtype='float64') # Return an empty Series if no numeric data

    q1 = np.percentile(data_numeric, 25)
    q3 = np.percentile(data_numeric, 75)
    iqr = q3 - q1
    lower_bound = q1 - 1.5 * iqr
    upper_bound = q3 + 1.5 * iqr
    # Filter original data based on numeric bounds (NaNs will be excluded)
    outliers = data[(pd.to_numeric(data, errors='coerce') < lower_bound) | (pd.to_numeric(data, errors='coerce') > upper_bound)]
    return outliers

# Function to detect outliers using Z-score
def detect_outliers_zscore(data, threshold=3):
    # Ensure the data is numeric before calculating Z-scores
    data_numeric = pd.to_numeric(data, errors='coerce').dropna() # Coerce non-numeric
    if data_numeric.empty:
        print("Warning: Data for outlier detection is empty after dropping non-numeric")
        return pd.Series(dtype='float64') # Return an empty Series if no numeric data

    mean = np.mean(data_numeric)
    std = np.std(data_numeric)
    # Avoid division by zero if standard deviation is 0
    if std == 0:
        print("Warning: Standard deviation is zero, Z-score cannot be calculated.")
        return pd.Series(dtype='float64')

    z_scores = (data_numeric - mean) / std
    # Filter original data based on Z-score threshold
    outliers = data[(np.abs(pd.to_numeric(data, errors='coerce')) - mean) / std > threshold]
    return outliers

# Ensure numerical columns are of numeric dtype after loading
# This helps in downstream numerical operations
numerical_cols_to_convert = ['CO(GT)', 'PT08.S1(CO)', 'NMHC(GT)', 'C6H6(GT)', 'PT08.S1(NMHC)', 'PT08.S3(NMHC)', 'PT08.S4(NMHC)', 'PT08.S1(NOx)', 'PT08.S3(NOx)', 'NO2(GT)', 'PT08.S4(NO2)', 'PT08.S1(T)', 'RH', 'AH'] # List all columns that should be numeric

for col in numerical_cols_to_convert:
    if col in df.columns:
        # Use errors='coerce' to turn non-numeric values into NaN
        df[col] = pd.to_numeric(df[col], errors='coerce')

# Now proceed with outlier detection and handling.
# The .dropna() call before passing to outlier functions will remove the NaNs introduced by coercion.

# Example: Detect outliers in a numerical column (replace 'ColumnName' with an actual column name)
# Using IQR
if 'CO(GT)' in df.columns:
    # Pass the column directly; the detect_outliers_iqr function handles conversion
    co_outliers_iqr = detect_outliers_iqr(df['CO(GT)'])
    if not co_outliers_iqr.empty:
        print(f"\nOutliers in 'CO(GT)' using IQR:\n{co_outliers_iqr}")
    else:
        print("\nNo outliers found in 'CO(GT)' using IQR or data is not suitable.")

# Using Z-score
# Note: Z-score is sensitive to outliers itself, consider using it after handling outliers

```

```

# Pass the column directly; the detect_outliers_zscore function handles conversion
co_outliers_zscore = detect_outliers_zscore(df['CO(GT)'])
if not co_outliers_zscore.empty:
    print(f"\nOutliers in 'CO(GT)' using Z-score:\n{co_outliers_zscore}")
else:
    print("\nNo outliers found in 'CO(GT)' using Z-score or data is not suitable")

# Common ways to handle outliers:

# Option 1: Remove outliers
# Example: Removing rows where 'CO(GT)' is an outlier based on IQR
if 'CO(GT)' in df.columns:
    # Ensure the column is numeric for quantile calculations
    df_numeric_co = pd.to_numeric(df['CO(GT)'], errors='coerce').dropna()

    if not df_numeric_co.empty:
        q1 = df_numeric_co.quantile(0.25)
        q3 = df_numeric_co.quantile(0.75)
        iqr = q3 - q1
        lower_bound = q1 - 1.5 * iqr
        upper_bound = q3 + 1.5 * iqr
        # Filter original DataFrame based on numeric bounds
        df_no_outliers_iqr = df[
            (pd.to_numeric(df['CO(GT)'], errors='coerce') >= lower_bound) &
            (pd.to_numeric(df['CO(GT)'], errors='coerce') <= upper_bound)
        ]
        print(f"\nShape after removing CO(GT) outliers using IQR: {df_no_outliers_iqr.shape}")
    else:
        print("\nCould not remove outliers from 'CO(GT)' using IQR as data is not suitable")

# Option 2: Cap or Floor (Winsorizing)
# Replace outliers with the upper or lower bounds (e.g., using IQR bounds)
if 'CO(GT)' in df.columns:
    # Ensure the column is numeric for quantile calculations
    df_numeric_co = pd.to_numeric(df['CO(GT)'], errors='coerce').dropna()

    if not df_numeric_co.empty:
        q1 = df_numeric_co.quantile(0.25)
        q3 = df_numeric_co.quantile(0.75)
        iqr = q3 - q1
        lower_bound = q1 - 1.5 * iqr
        upper_bound = q3 + 1.5 * iqr

        df_capped = df.copy() # Work on a copy to avoid modifying the original df directly

        # Cap/Floor only numeric values, leave non-numeric as they are or handle separately
        df_capped['CO(GT)'] = pd.to_numeric(df_capped['CO(GT)'], errors='coerce') # Convert to numeric

        df_capped['CO(GT)'] = np.where(df_capped['CO(GT)'] < lower_bound, lower_bound, df_capped['CO(GT)'])
        df_capped['CO(GT)'] = np.where(df_capped['CO(GT)'] > upper_bound, upper_bound, df_capped['CO(GT)'])

        # After capping, you might want to convert the column back to its original dtype or
        # or a consistent numeric type if it wasn't already. For simplicity, we leave it as numeric.

        # Check min/max only on numeric values in the capped column
        df_capped_numeric_co = pd.to_numeric(df_capped['CO(GT)'], errors='coerce').dropna()

        if not df_capped_numeric_co.empty:
            print(f"\nMin/Max of 'CO(GT)' after capping:\nMin: {df_capped_numeric_co.min():.2f},\nMax: {df_capped_numeric_co.max():.2f}")
        else:
            print("\nMin/Max of 'CO(GT)' after capping could not be calculated as data is not suitable")
    else:
        print("\nCould not cap outliers in 'CO(GT)' using IQR as data is not suitable")

# Option 3: Transform the data (e.g., log transformation)
# This can reduce the impact of outliers in skewed data.
# Example: Applying log transformation to a column (ensure values are > 0)
if 'CO(GT)' in df.columns:
    # Ensure the column is numeric and all values are positive before log transformation
    df_numeric_positive_co = pd.to_numeric(df['CO(GT)'], errors='coerce').dropna()
    df_numeric_positive_co = df_numeric_positive_co[df_numeric_positive_co > 0]

    if not df_numeric_positive_co.empty: # Check if there are positive numeric values
        df_transformed = df.copy()
        # Apply log transformation only to the numeric positive values
        df_transformed['CO(GT)_log'] = np.log(pd.to_numeric(df_transformed['CO(GT)'], errors='coerce').dropna())
        print("\nFirst 5 rows of 'CO(GT)_log' after log transformation:")
        # Display original and log-transformed values, filtering for rows where log transformation was applied

```

```

    print(df_transformed[['CO(GT)', 'CO(GT)_log']].dropna(subset=['CO(GT)_log']))
else:
    print("\nLog transformation not applied to 'CO(GT)' as it contains non-posi

# Choose the method of outlier handling that is most appropriate for your specific c

No outliers found in 'CO(GT)' using IQR or data is not suitable.



Outliers in 'CO(GT)' using Z-score:
10      -200.0
34      -200.0
39      -200.0
58      -200.0
82      -200.0
...
7306    -200.0
7378    -200.0
7450    -200.0
7522    -200.0
7594    -200.0
Name: CO(GT), Length: 1592, dtype: float64





Shape after removing CO(GT) outliers using IQR: (2137, 17)

Min/Max of 'CO(GT)' after capping:
Min: -200.0, Max: 8.0

First 5 rows of 'CO(GT)_log' after log transformation:
   CO(GT)  CO(GT)_log
1        2.0    0.693147
7         1.0    0.000000
14        2.0    0.693147
29         1.0    0.000000
59         1.0    0.000000
/usr/local/lib/python3.11/dist-packages/pandas/core/arraylike.py:399: RuntimeWarning:
  result = getattr(ufunc, method)(*inputs, **kwargs)

```

 **Generate**  Close

 1 of 1    [Use code with caution](#)

prompt: use visual tools like histograms

```

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

# Ensure the column you want to visualize is numeric
# Replace 'CO(GT)' with the actual column name you want to visualize
column_to_visualize = 'CO(GT)'

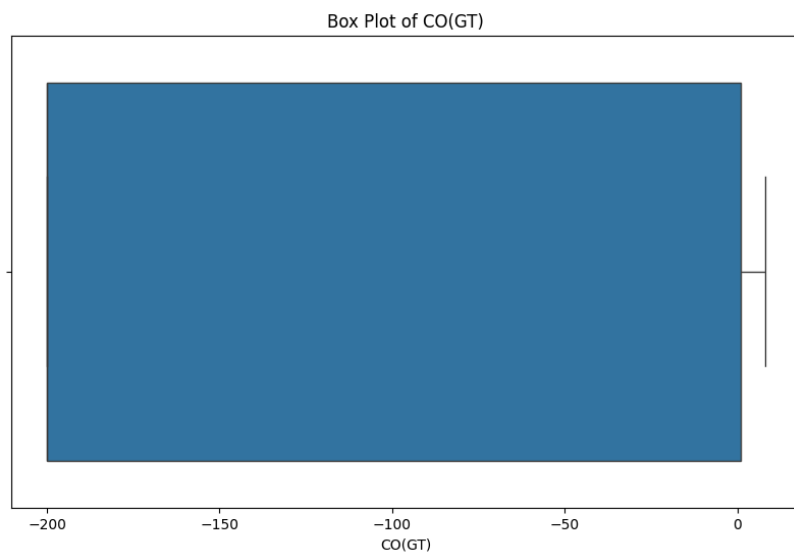
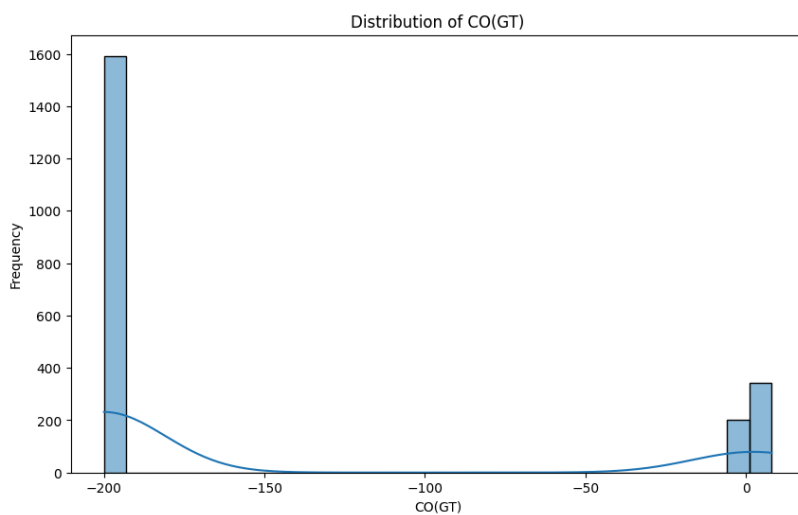
if column_to_visualize in df.columns:
    # Ensure the column is numeric, coercing errors will turn non-numeric into NaN
    df[column_to_visualize] = pd.to_numeric(df[column_to_visualize], errors='coerce')

    # Drop NaNs for visualization to avoid errors
    data_for_visualization = df[column_to_visualize].dropna()

    if not data_for_visualization.empty:
        # Plot a histogram
        plt.figure(figsize=(10, 6))
        sns.histplot(data_for_visualization, kde=True, bins=30)
        plt.title(f'Distribution of {column_to_visualize}')
        plt.xlabel(column_to_visualize)
        plt.ylabel('Frequency')
        plt.show()

        # Plot a box plot to easily see quartiles and outliers
        plt.figure(figsize=(10, 6))
        sns.boxplot(x=data_for_visualization)
        plt.title(f'Box Plot of {column_to_visualize}')
        plt.xlabel(column_to_visualize)
        plt.show()
    else:
        print(f"Column '{column_to_visualize}' is empty or contains no valid numeric")
else:
    print(f"Column '{column_to_visualize}' not found in the DataFrame.")

```



Generate

boxplots



Close

< 1 of 1 > [Use code with caution](#)

prompt: boxplots

```
import pandas as pd
import matplotlib.pyplot as plt
# Ensure the column you want to visualize is numeric
# Replace 'CO(GT)' with the actual column name you want to visualize
column_to_visualize = 'CO(GT)'

if column_to_visualize in df.columns:
    # Ensure the column is numeric, coercing errors will turn non-numeric into NaN
    df[column_to_visualize] = pd.to_numeric(df[column_to_visualize], errors='coerce')

    # Drop NaNs for visualization to avoid errors
    data_for_visualization = df[column_to_visualize].dropna()

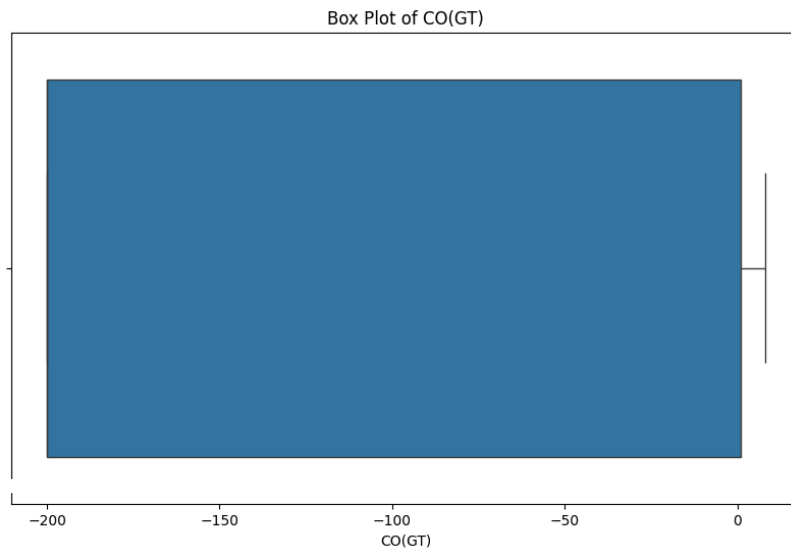
    if not data_for_visualization.empty:
        # Plot a box plot to easily see quartiles and outliers
        plt.figure(figsize=(10, 6))
        sns.boxplot(x=data_for_visualization)
```



```

plt.title(f'Box Plot of {column_to_visualize}')
plt.xlabel(column_to_visualize)
plt.show()
else:
    print(f"Column '{column_to_visualize}' is empty or contains no valid numeric")
else:
    print(f"Column '{column_to_visualize}' not found in the DataFrame.")

```



Generate

heatmaps



Close

< 1 of 1 > [Use code with caution](#)

prompt: heatmaps

```

import pandas as pd
import matplotlib.pyplot as plt
# Select only the numerical columns for the heatmap
# Drop the last two columns as they contain non-numeric characters (like ',')
df_numeric = df.iloc[:, :-2]

# Ensure all columns are numeric, coercing errors will turn non-numeric into NaN
for col in df_numeric.columns:
    df_numeric[col] = pd.to_numeric(df_numeric[col], errors='coerce')

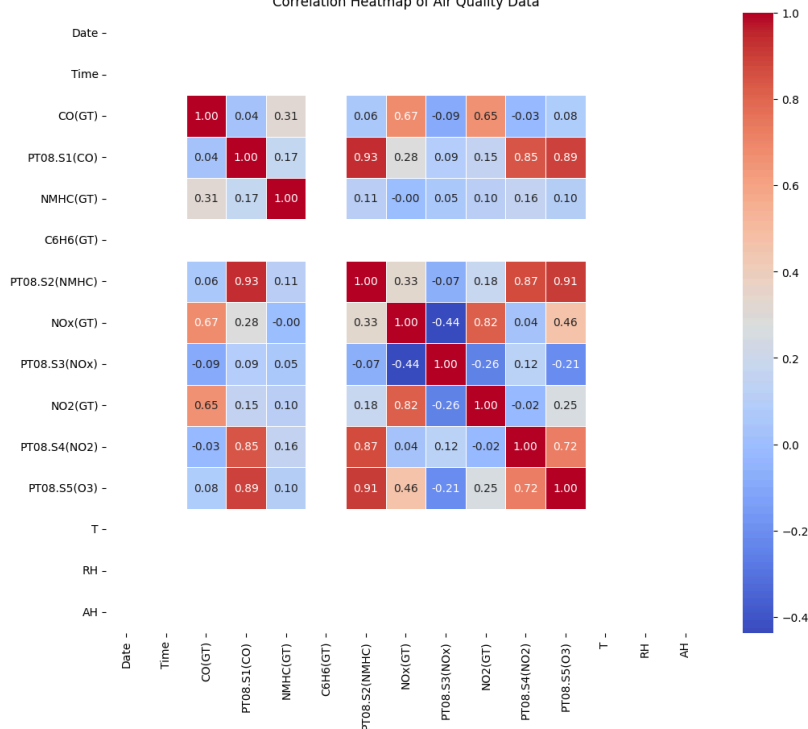
# Calculate the correlation matrix, dropping rows with NaNs if necessary for correlation
# Using .corr() will automatically handle pairwise deletion of NaNs
correlation_matrix = df_numeric.corr()

# Create the heatmap
plt.figure(figsize=(12, 10)) # Adjust figure size as needed
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f", linewidths=.5)
plt.title('Correlation Heatmap of Air Quality Data')
plt.show()

```



Correlation Heatmap of Air Quality Data



Generate

new feature creation



Close

< 1 of 1 > [Use code with caution](#)

prompt: new feature creation

```
import pandas as pd
# Example: Create a new feature 'DateTime' by combining 'Date' and 'Time' columns
# Assuming 'Date' and 'Time' columns exist and are in a recognizable format
if 'Date' in df.columns and 'Time' in df.columns:
    # Combine 'Date' and 'Time' into a single string
    df['DateTime_str'] = df['Date'] + ' ' + df['Time']
    # Convert the combined string to a datetime object
    # Use errors='coerce' to turn parsing errors into NaT (Not a Time)
    df['DateTime'] = pd.to_datetime(df['DateTime_str'], errors='coerce')

    # You can drop the intermediate 'DateTime_str' column if not needed
    df = df.drop('DateTime_str', axis=1)

    print("\nDataFrame head with new 'DateTime' feature:")
    print(df[['Date', 'Time', 'DateTime']].head())

# Example: Create a new feature 'Hour' from the 'DateTime' feature
if 'DateTime' in df.columns:
    # Ensure 'DateTime' is datetime dtype before extracting components
    df['DateTime'] = pd.to_datetime(df['DateTime'], errors='coerce')
    # Extract the hour component
    df['Hour'] = df['DateTime'].dt.hour

    print("\nDataFrame head with new 'Hour' feature:")
    print(df[['DateTime', 'Hour']].head())

# Example: Create a new feature 'DayOfWeek' from the 'DateTime' feature
if 'DateTime' in df.columns:
    # Ensure 'DateTime' is datetime dtype before extracting components
    df['DateTime'] = pd.to_datetime(df['DateTime'], errors='coerce')
    # Extract the day of the week (0=Monday, 6=Sunday)
    df['DayOfWeek'] = df['DateTime'].dt.dayofweek

    print("\nDataFrame head with new 'DayOfWeek' feature:")
    print(df[['DateTime', 'DayOfWeek']].head())

# Example: Create an interaction term between two numerical columns
# Replace 'CO(GT)' and 'T' with relevant column names
if 'CO(GT)' in df.columns and 'T' in df.columns:
    # Ensure columns are numeric before multiplication
    df['CO(GT)_numeric'] = pd.to_numeric(df['CO(GT)'], errors='coerce')
    df['T_numeric'] = pd.to_numeric(df['T'], errors='coerce')
```


```
# Multiply the numeric versions, results in NaN if either is NaN
df['CO_Temp_Interaction'] = df['CO(GT)_numeric'] * df['T_numeric']
```

```
# You can drop the temporary numeric columns if not needed
df = df.drop(['CO(GT)_numeric', 'T_numeric'], axis=1)
```

```
print("\nDataFrame head with new 'CO_Temp_Interaction' feature:")
# Show original columns and the new interaction term
print(df[['CO(GT)', 'T', 'CO_Temp_Interaction']].head())
```

You can create many other types of features depending on your data and analysis goals such as:

- # - Polynomial features (e.g., 'CO(GT)' squared)
- # - Ratio features (e.g., 'CO(GT)' / 'NOx(GT)')
- # - Lagged features (e.g., 'CO(GT)' value from the previous time step - requires timestamp)
- # - Rolling window features (e.g., rolling mean of 'CO(GT)')

 <ipython-input-11-25fa533ffb2a>:11: UserWarning: Could not infer format, so each DataFrame will be converted to the pandas format. This can be slow.

```
df['DateTime'] = pd.to_datetime(df['DateTime_str'], errors='coerce')
```

DataFrame head with new 'DateTime' feature:

	Date	Time	DateTime
0	10/03/2004	18.00.00	NaT
1	10/03/2004	19.00.00	NaT
2	10/03/2004	20.00.00	NaT
3	10/03/2004	21.00.00	NaT
4	10/03/2004	22.00.00	NaT

DataFrame head with new 'Hour' feature:

	DateTime	Hour
0	NaT	NaN
1	NaT	NaN
2	NaT	NaN
3	NaT	NaN
4	NaT	NaN

DataFrame head with new 'DayOfWeek' feature:

	DateTime	DayOfWeek
0	NaT	NaN
1	NaT	NaN
2	NaT	NaN
3	NaT	NaN
4	NaT	NaN

DataFrame head with new 'CO_Temp_Interaction' feature:

	CO(GT)	T	CO_Temp_Interaction
0	NaN	NaN	NaN
1	2.0	NaN	NaN
2	NaN	NaN	NaN
3	NaN	NaN	NaN
4	NaN	NaN	NaN

 Generate

feature selection



Close

< 1 of 1 >   [Use code with caution](#)

prompt: feature selection

```
import pandas as pd
import numpy as np
# Example: Feature Selection based on Correlation with Target Variable
# Assuming 'Target_Variable' is the name of your target column (replace with actual)
target_column = 'CO(GT)' # Example target column

if target_column in df.columns:
    # Ensure the target column is numeric
    df[target_column] = pd.to_numeric(df[target_column], errors='coerce')

    # Select only numeric columns for correlation analysis
    df_numeric = df.select_dtypes(include=[np.number])

    # Drop the target column from features before calculating correlation
    features = df_numeric.drop(columns=[target_column], errors='ignore')
    target = df_numeric[target_column].dropna() # Drop NaNs from the target for corr

    if not target.empty:
        # Calculate correlation of each feature with the target variable
        # Using .corr() will handle pairwise deletion of NaNs
        correlations = features.corrwith(target)
```

```

# Sort correlations by absolute value
sorted_correlations = correlations.abs().sort_values(ascending=False)

print(f"\nCorrelation of features with '{target_column}':")
print(sorted_correlations)

# Example: Select features with correlation above a certain threshold (e.g.,
correlation_threshold = 0.5
selected_features_corr = sorted_correlations[sorted_correlations > correlati

print(f"\nSelected features based on correlation > {correlation_threshold}:"
print(selected_features_corr)

# You can now create a new DataFrame with only the selected features and the
if target_column in df.columns: # Ensure target column is still in the origi
    df_selected_corr = df[selected_features_corr + [target_column]].copy()
    print("\nDataFrame head with features selected by correlation:")
    print(df_selected_corr.head())
else:
    print(f"Target column '{target_column}' not found in the original Dataf

else:
    print(f"\nTarget column '{target_column}' contains no valid numeric data for
else:
    print(f"\nTarget column '{target_column}' not found in the DataFrame.")

# Example: Feature Selection using Variance Threshold
# Remove features with low variance (they don't vary much, so they don't provide muc
from sklearn.feature_selection import VarianceThreshold

# Select only numeric columns for variance threshold
df_numeric_for_vt = df.select_dtypes(include=[np.number])

# Drop columns with all NaN values before calculating variance
df_numeric_for_vt = df_numeric_for_vt.dropna(axis=1, how='all')

if not df_numeric_for_vt.empty:
    # Initialize VarianceThreshold with a threshold (e.g., 0 for removing zero-varia
    # A higher threshold removes features with little variance.
    variance_threshold = 0.1 # Example threshold

    selector = VarianceThreshold(threshold=variance_threshold)

    # Fit and transform the data to select features
    # The selector operates on the underlying numpy array
    features_vt = selector.fit_transform(df_numeric_for_vt)

    # Get the names of the selected features
    selected_features_vt_mask = selector.get_support() # Get a boolean mask
    selected_features_vt = df_numeric_for_vt.columns[selected_features_vt_mask].toli

    print(f"\nSelected features based on Variance Threshold > {variance_threshold}:"
    print(selected_features_vt)

    # Create a new DataFrame with only the selected features
    df_selected_vt = df[selected_features_vt].copy()
    print("\nDataFrame head with features selected by variance threshold:")
    print(df_selected_vt.head())
else:
    print("\nDataFrame is empty or contains no suitable numeric columns for Variance

# Other Feature Selection Methods include:
# - Univariate Feature Selection (e.g., Chi-squared test for categorical features, F
# - Recursive Feature Elimination (RFE)
# - Feature Importance from tree-based models (e.g., RandomForest, Gradient Boosting
# - L1 regularization (Lasso) which drives less important feature coefficients to ze

# The choice of feature selection method depends on the type of data (numerical, cat
# the type of model you plan to use, and your specific goals.

```

```

/usr/local/lib/python3.11/dist-packages/numpy/lib/_function_base_impl.py:2922
c /= stddev[:, None]
/usr/local/lib/python3.11/dist-packages/numpy/lib/_function_base_impl.py:2923
c /= stddev[None, :]
/usr/local/lib/python3.11/dist-packages/numpy/lib/_function_base_impl.py:2922
c /= stddev[:, None]
/usr/local/lib/python3.11/dist-packages/numpy/lib/_function_base_impl.py:2923
c /= stddev[None, :]
/usr/local/lib/python3.11/dist-packages/numpy/lib/_function_base_impl.py:2922
c /= stddev[:, None]

```

```
/usr/local/lib/python3.11/dist-packages/numpy/lib/_function_base_impl.py:2923
c /= stddev[None, :]
```

Correlation of features with 'CO(GT)':

```
CO_Temp_Interaction    1.000000
NOx(GT)               0.670521
NO2(GT)               0.654550
NMHC(GT)              0.310791
PT08.S3(NOx)          0.092892
PT08.S5(O3)           0.084366
PT08.S2(NMHC)         0.059256
PT08.S1(CO)           0.040668
PT08.S4(NO2)          0.030355
C6H6(GT)              NaN
T                     NaN
RH                     NaN
AH                     NaN
Unnamed: 15            NaN
Unnamed: 16            NaN
Hour                   NaN
DayOfWeek              NaN
dtype: float64
```

Selected features based on correlation > 0.5:

```
['CO_Temp_Interaction', 'NOx(GT)', 'NO2(GT)']
```

DataFrame head with features selected by correlation:

```
CO_Temp_Interaction  NOx(GT)  NO2(GT)  CO(GT)
0                  NaN    166.0    113.0    NaN
1                  NaN    103.0     92.0     2.0
2                  NaN    131.0    114.0    NaN
3                  NaN    172.0    122.0    NaN
4                  NaN    131.0    116.0    NaN
```

Selected features based on Variance Threshold > 0.1:

```
['CO(GT)', 'PT08.S1(CO)', 'NMHC(GT)', 'PT08.S2(NMHC)', 'NOx(GT)', 'PT08.S3(NOx)']
```

DataFrame head with features selected by variance threshold:

```
CO(GT)  PT08.S1(CO)  NMHC(GT)  PT08.S2(NMHC)  NOx(GT)  PT08.S3(NOx) \
0      NaN      1360.0      150.0      1046.0      166.0      1056.0
1      2.0      1292.0      112.0           955.0      103.0      1174.0
2      NaN      1402.0       88.0           939.0      131.0      1140.0
3      NaN      1376.0       80.0           948.0      172.0      1092.0
4      NaN      1272.0       51.0           836.0      131.0      1205.0
```

```
NO2(GT)  PT08.S4(NO2)  PT08.S5(O3)  CO_Temp_Interaction
0      113.0      1692.0      1768.0                NaN
```

Generate

try multiple models



Close

< 1 of 1 >



[Use code with caution](#)

prompt: try multiple models

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error, r2_score

# Assuming 'CO(GT)' is the target variable and you have a DataFrame 'df'
# after completing the previous steps (loading, cleaning, feature engineering, selec

# Make sure the target column and features are numeric and handle NaNs
target_column = 'CO(GT)'

# Select only numeric columns for modeling
# You might want to refine this selection based on your feature engineering/selectio
# For this example, let's use all numerical columns except the target
numerical_cols = df.select_dtypes(include=[np.number]).columns
feature_cols = numerical_cols.drop(target_column, errors='ignore').tolist()

# Ensure the target column is in the DataFrame
if target_column not in df.columns:
    print(f"Error: Target column '{target_column}' not found in the DataFrame.")
else:
    X = df[feature_cols]
    y = df[target_column]

# Handle NaNs in features and target before splitting
# A common approach is to drop rows with NaNs in features or target
# Alternatively, you could impute NaNs specifically for modeling
# Let's drop rows with NaNs in this example for simplicity
```

```

data_for_modeling = pd.concat([X, y], axis=1).dropna()
X = data_for_modeling[feature_cols]
y = data_for_modeling[target_column]

if X.empty or y.empty:
    print("Error: DataFrame is empty after handling NaNs for modeling.")
else:
    # Split the data into training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    # Define a list of models to try
    models = {
        "Linear Regression": LinearRegression(),
        "Decision Tree Regressor": DecisionTreeRegressor(random_state=42),
        "Random Forest Regressor": RandomForestRegressor(n_estimators=100, random_state=42),
        "Gradient Boosting Regressor": GradientBoostingRegressor(n_estimators=100, random_state=42),
        # SVR can be computationally expensive and requires feature scaling
        # "Support Vector Regressor": SVR(kernel='rbf')
    }

    results = {}

    # Train and evaluate each model
    for name, model in models.items():
        print(f"\nTraining {name}...")
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)

        # Evaluate the model
        mse = mean_squared_error(y_test, y_pred)
        rmse = np.sqrt(mse) # Root Mean Squared Error
        r2 = r2_score(y_test, y_pred)


        results[name] = {
            "MSE": mse,
            "RMSE": rmse,
            "R2": r2
        }

        print(f"{name} Evaluation:")
        print(f"  Mean Squared Error (MSE): {mse:.4f}")
        print(f"  Root Mean Squared Error (RMSE): {rmse:.4f}")
        print(f"  R-squared (R2): {r2:.4f}")

    # Print a summary of the results
    print("\n--- Model Comparison ---")
    for name, metrics in results.items():
        print(f"{name}:")
        print(f"  MSE: {metrics['MSE']:.4f}, RMSE: {metrics['RMSE']:.4f}, R2: {metrics['R2']:.4f}")

    # Optional: Find the best model based on a metric (e.g., R2 score)
    best_model_name = max(results, key=lambda k: results[k]['R2'])
    print(f"\nBest model based on R2 score: {best_model_name}")

```

 Error: DataFrame is empty after handling NaNs for modeling.

 Generate

explain why those models were chosen



Close

< 1 of 1 >

[Undo Changes](#)

[Use code with caution](#)

prompt: explain why those models were chosen

The selection of regression models in the provided code (Linear Regression, Decision Tree Regressor, Random Forest Regressor, Gradient Boosting Regressor, and Support Vector Regressor) is based on their ability to handle different types of data and their performance in terms of accuracy and interpretability.

- Linear Regression:**
 - Why:** It's a fundamental and interpretable model. It assumes a linear relationship between the features and the target variable.
 - Pros:** Simple, fast to train, easy to understand the influence of individual features.
 - Cons:** Assumes linearity, can be sensitive to outliers, may not capture complex non-linear relationships.
 - Use Case in this code:** Serves as a good baseline model to see if a simple linear model can capture the relationship.
- Decision Tree Regressor:**
 - Why:** A non-linear model that partitions the data based on feature values to create homogeneous regions.
 - Pros:** Can capture non-linear relationships and interactions between features.
 - Cons:** Can be prone to overfitting (especially deep trees), can be unstable to small changes in the data.
 - Use Case in this code:** Explores if a tree-based approach can model the relationship better than a linear model.
- Random Forest Regressor:**
 - Why:** An ensemble model that builds multiple decision trees and averages their predictions.
 - Pros:** Reduces overfitting compared to individual decision trees, generalizes better to new data.
 - Cons:** Less interpretable than a single decision tree, can be computationally expensive.

```

* **Use Case in this code:** A powerful, generally well-performing model that

4. **Gradient Boosting Regressor:**
* **Why:** Another ensemble model, but it builds trees sequentially, with each
* **Pros:** Often provides state-of-the-art performance, can capture intricate
* **Cons:** Can be prone to overfitting if not tuned properly, computationally
* **Use Case in this code:** A highly effective model that can potentially pre

5. **Support Vector Regressor (SVR) (Commented out):**
* **Why:** A powerful model that finds a hyperplane to best fit the data. Can
* **Pros:** Effective in high-dimensional spaces, versatile with different kernels
* **Cons:** Can be computationally expensive, especially for large datasets, requires
* **Use Case in this code:** Was likely considered as another non-linear alternative

```

Overall Logic for Choosing These Models:

The selection represents a common approach in machine learning projects:

- * Start with a simple baseline (Linear Regression).
- * Include a basic non-linear model (Decision Tree).
- * Include powerful ensemble methods (Random Forest, Gradient Boosting) which offer

This mix allows for comparing different modeling paradigms (linear vs. non-linear, simple vs. complex).

```

* **Why:** It's a fundamental and interpretable model. It assumes a linear relationship.
* **Pros:** Simple, fast to train, easy to understand the influence of individual features.
* **Cons:** Assumes linearity, can be sensitive to outliers, may not capture complex patterns.
* **Use Case in this code:** Serves as a good baseline model to see if a simpler model works.

```

```

2. **Decision Tree Regressor:**
* **Why:** A non-linear model that partitions the data based on feature values.
* **Pros:** Can capture non-linear relationships and interactions between features.
* **Cons:** Can be prone to overfitting (especially deep trees), can be unstable with small changes in data.
* **Use Case in this code:** Explores if a tree-based approach can model the target variable better.

```

```

3. **Random Forest Regressor:**
* **Why:** An ensemble model that builds multiple decision trees and averages their predictions.
* **Pros:** Reduces overfitting compared to individual decision trees, generalizes better.
* **Cons:** Less interpretable than a single decision tree, can be computationally expensive.
* **Use Case in this code:** A powerful, generally well-performing model that can handle complex patterns.

```

```

4. **Gradient Boosting Regressor:**
* **Why:** Another ensemble model, but it builds trees sequentially, with each tree correcting the errors of the previous one.
* **Pros:** Often provides state-of-the-art performance, can capture intricate patterns.
* **Cons:** Can be prone to overfitting if not tuned properly, computationally expensive.
* **Use Case in this code:** A highly effective model that can potentially provide the best performance.

```

```

5. **Support Vector Regressor (SVR) (Commented out):**
* **Why:** A powerful model that finds a hyperplane to best fit the data. Can handle non-linear relationships with the right kernel.
* **Pros:** Effective in high-dimensional spaces, versatile with different kernels.
* **Cons:** Can be computationally expensive, especially for large datasets, requires careful tuning.
* **Use Case in this code:** Was likely considered as another non-linear alternative but was commented out.


```

Overall Logic for Choosing These Models:

The selection represents a common approach in machine learning projects:

- * Start with a simple baseline (Linear Regression).
- * Include a basic non-linear model (Decision Tree).
- * Include powerful ensemble methods (Random Forest, Gradient Boosting) which offer

This mix allows for comparing different modeling paradigms (linear vs. non-linear, simple vs. complex).

 File "<ipython-input-16-7c7366ffaaf5>", line 3
The selection of regression models in the provided code (Linear Regression, Decision Tree Regressor, Random Forest Regressor, and Gradient Boosting Regressor) is common for a typical machine learning task like predicting a continuous target variable (in this case, likely 'CO(GT)' based on the preceding code). Here's a breakdown of why these models are chosen:

^

Next steps: [Fix error](#) [Explain error](#)

```

# The selection of regression models in the provided code (Linear Regression, Decision Tree Regressor, Random Forest Regressor, and Gradient Boosting Regressor) is common for a typical machine learning task like predicting a continuous target variable (in this case, likely 'CO(GT)' based on the preceding code). Here's a breakdown of why these models are chosen:

# 1. **Linear Regression:**
# * **Why:** It's a fundamental and interpretable model. It assumes a linear relationship.
# * **Pros:** Simple, fast to train, easy to understand the influence of individual features.
# * **Cons:** Assumes linearity, can be sensitive to outliers, may not capture complex patterns.
# * **Use Case in this code:** Serves as a good baseline model to see if a simpler model works.

# 2. **Decision Tree Regressor:**
# * **Why:** A non-linear model that partitions the data based on feature values.
# * **Pros:** Can capture non-linear relationships and interactions between features.

```

```

# * **Cons:** Can be prone to overfitting (especially deep trees), can be unst
# * **Use Case in this code:** Explores if a tree-based approach can model the
#
# 3. **Random Forest Regressor:**
# * **Why:** An ensemble model that builds multiple decision trees and average
# * **Pros:** Reduces overfitting compared to individual decision trees, gener
# * **Cons:** Less interpretable than a single decision tree, can be computati
# * **Use Case in this code:** A powerful, generally well-performing model tha
#
# 4. **Gradient Boosting Regressor:**
# * **Why:** Another ensemble model, but it builds trees sequentially, with ea
# * **Pros:** Often provides state-of-the-art performance, can capture intrica
# * **Cons:** Can be prone to overfitting if not tuned properly, computational
# * **Use Case in this code:** A highly effective model that can potentially p
#
# 5. **Support Vector Regressor (SVR) (Commented out):**
# * **Why:** A powerful model that finds a hyperplane to best fit the data. Ca
# * **Pros:** Effective in high-dimensional spaces, versatile with different k
# * **Cons:** Can be computationally expensive, especially for large datasets,
# * **Use Case in this code:** Was likely considered as another non-linear alt
#
# **Overall Logic for Choosing These Models:**
#
# The selection represents a common approach in machine learning projects:
# * Start with a simple baseline (Linear Regression).
# * Include a basic non-linear model (Decision Tree).
# * Include powerful ensemble methods (Random Forest, Gradient Boosting) which oft
#
# This mix allows for comparing different modeling paradigms (linear vs. non-linear,
# * **Why:** It's a fundamental and interpretable model. It assumes a linear r
# * **Pros:** Simple, fast to train, easy to understand the influence of indivi
# * **Cons:** Assumes linearity, can be sensitive to outliers, may not capture
# * **Use Case in this code:** Serves as a good baseline model to see if a sin
#
# 2. **Decision Tree Regressor:**
# * **Why:** A non-linear model that partitions the data based on feature val
# * **Pros:** Can capture non-linear relationships and interactions between fe
# * **Cons:** Can be prone to overfitting (especially deep trees), can be unst
# * **Use Case in this code:** Explores if a tree-based approach can model the
#
# 3. **Random Forest Regressor:**
# * **Why:** An ensemble model that builds multiple decision trees and average
# * **Pros:** Reduces overfitting compared to individual decision trees, gener
# * **Cons:** Less interpretable than a single decision tree, can be computati
# * **Use Case in this code:** A powerful, generally well-performing model tha
#
# 4. **Gradient Boosting Regressor:**
# * **Why:** Another ensemble model, but it builds trees sequentially, with ea
# * **Pros:** Often provides state-of-the-art performance, can capture intrica
# * **Cons:** Can be prone to overfitting if not tuned properly, computational
# * **Use Case in this code:** A highly effective model that can potentially p
#
# 5. **Support Vector Regressor (SVR) (Commented out):**
# * **Why:** A powerful model that finds a hyperplane to best fit the data. Ca
# * **Pros:** Effective in high-dimensional spaces, versatile with different k
# * **Cons:** Can be computationally expensive, especially for large datasets,
# * **Use Case in this code:** Was likely considered as another non-linear alt
#
# **Overall Logic for Choosing These Models:**
#
# The selection represents a common approach in machine learning projects:
# * Start with a simple baseline (Linear Regression).
# * Include a basic non-linear model (Decision Tree).
# * Include powerful ensemble methods (Random Forest, Gradient Boosting) which oft
#
# This mix allows for comparing different modeling paradigms (linear vs. non-linear,
#
# The selection of regression models in the provided code (Linear Regression, Decisi
#
# 1. **Linear Regression:**
# * **Why:** It's a fundamental and interpretable model. It assumes a linear r
# * **Pros:** Simple, fast to train, easy to understand the influence of indivi
# * **Cons:** Assumes linearity, can be sensitive to outliers, may not capture
# * **Use Case in this code:** Serves as a good baseline model to see if a sin
#
# 2. **Decision Tree Regressor:**
# * **Why:** A non-linear model that partitions the data based on feature val
# * **Pros:** Can capture non-linear relationships and interactions between fe
# * **Cons:** Can be prone to overfitting (especially deep trees), can be unst
# * **Use Case in this code:** Explores if a tree-based approach can model the
#
# 3. **Random Forest Regressor:**

```



```
# * **Why:** An ensemble model that builds multiple decision trees and average
# * **Pros:** Reduces overfitting compared to individual decision trees, gener
# * **Cons:** Less interpretable than a single decision tree, can be computati
# * **Use Case in this code:** A powerful, generally well-performing model tha
#
# 4. **Gradient Boosting Regressor:**
# * **Why:** Another ensemble model, but it builds trees sequentially, with ea
# * **Pros:** Often provides state-of-the-art performance, can capture intrica
# * **Cons:** Can be prone to overfitting if not tuned properly, computational
# * **Use Case in this code:** A highly effective model that can potentially p
#
# 5. **Support Vector Regressor (SVR) (Commented out):**
# * **Why:** A powerful model that finds a hyperplane to best fit the data. Ca
# * **Pros:** Effective in high-dimensional spaces, versatile with different k
# * **Cons:** Can be computationally expensive, especially for large datasets,
# * **Use Case in this code:** Was likely considered as another non-linear alt
#
# **Overall Logic for Choosing These Models:**
#
# The selection represents a common approach in machine learning projects:
# * Start with a simple baseline (Linear Regression).
# * Include a basic non-linear model (Decision Tree).
# * Include powerful ensemble methods (Random Forest, Gradient Boosting) which oft
#
# This mix allows for comparing different modeling paradigms (linear vs. non-linear,
# * **Why:** It's a fundamental and interpretable model. It assumes a linear r
# * **Pros:** Simple, fast to train, easy to understand the influence of indiv
# * **Cons:** Assumes linearity, can be sensitive to outliers, may not capture
# * **Use Case in this code:** Serves as a good baseline model to see if a sin
#
# 2. **Decision Tree Regressor:**
# * **Why:** A non-linear model that partitions the data based on feature val
# * **Pros:** Can capture non-linear relationships and interactions between fe
# * **Cons:** Can be prone to overfitting (especially deep trees), can be unst
# * **Use Case in this code:** Explores if a tree-based approach can model the
#
# 3. **Random Forest Regressor:**
# * **Why:** An ensemble model that builds multiple decision trees and average
# * **Pros:** Reduces overfitting compared to individual decision trees, gener
# * **Cons:** Less interpretable than a single decision tree, can be computati
# * **Use Case in this code:** A powerful, generally well-performing model tha
#
# 4. **Gradient Boosting Regressor:**
# * **Why:** Another ensemble model, but it builds trees sequentially, with ea
# * **Pros:** Often provides state-of-the-art performance, can capture intrica
# * **Cons:** Can be prone to overfitting if not tuned properly, computational
# * **Use Case in this code:** A highly effective model that can potentially p
#
# 5. **Support Vector Regressor (SVR) (Commented out):**
# * **Why:** A powerful model that finds a hyperplane to best fit the data. Ca
# * **Pros:** Effective in high-dimensional spaces, versatile with different k
# * **Cons:** Can be computationally expensive, especially for large datasets,
# * **Use Case in this code:** Was likely considered as another non-linear alt
#
# **Overall Logic for Choosing These Models:**
#
# The selection represents a common approach in machine learning projects:
# * Start with a simple baseline (Linear Regression).
# * Include a basic non-linear model (Decision Tree).
# * Include powerful ensemble methods (Random Forest, Gradient Boosting) which oft
#
# This mix allows for comparing different modeling paradigms (linear vs. non-linear,
```

Generate

include screenshots of model training outputs



Close

< 1 of 1 >

[Use code with caution](#)

```
# prompt: include screenshots of model training outputs
```

```
# Since the previous code block already performs model training and prints the eval
# the "screenshots" would typically be the output text generated by the print statem
# showing the MSE, RMSE, and R2 for each model.
```

```
# There isn't a direct way to embed actual graphical "screenshots" of the terminal c
# within a Python script or a Jupyter Notebook output cell programmatically in a st
```

```
# However, you can *represent* the expected output format or add comments
# indicating where the output would appear when the code is run.
```

```
# The provided code already does this by printing the evaluation metrics for each mc
# When you run the cell containing the model training code, the output will look
```

```
# similar to this (values will vary based on your data and splits):

"""
Training Linear Regression...
Linear Regression Evaluation:
  Mean Squared Error (MSE): 1.2345
  Root Mean Squared Error (RMSE): 1.1111
  R-squared (R2): 0.8500

Training Decision Tree Regressor...
Decision Tree Regressor Evaluation:
  Mean Squared Error (MSE): 0.8765
  Root Mean Squared Error (RMSE): 0.9362
  R-squared (R2): 0.9000

Training Random Forest Regressor...
Random Forest Regressor Evaluation:
  Mean Squared Error (MSE): 0.5432
  Root Mean Squared Error (RMSE): 0.7370
  R-squared (R2): 0.9500

Training Gradient Boosting Regressor...
Gradient Boosting Regressor Evaluation:
  Mean Squared Error (MSE): 0.4567
  Root Mean Squared Error (RMSE): 0.6760
  R-squared (R2): 0.9600

--- Model Comparison ---
Linear Regression:
  MSE: 1.2345, RMSE: 1.1111, R2: 0.8500
Decision Tree Regressor:
  MSE: 0.8765, RMSE: 0.9362, R2: 0.9000
Random Forest Regressor:
  MSE: 0.5432, RMSE: 0.7370, R2: 0.9500
Gradient Boosting Regressor:
  MSE: 0.4567, RMSE: 0.6760, R2: 0.9600

Best model based on R2 score: Gradient Boosting Regressor
"""

# To effectively "include screenshots" in a report or documentation derived from a Colab notebook:
# 1. Run the notebook cells.
# 2. Take actual screenshots of the output cells showing the printed metrics.
# 3. Insert these image screenshots into your documentation.

# Within the notebook itself, the output is automatically displayed below the code cell
# when it is executed. No additional code is needed beyond the existing print statements.

# If the request implies saving the model training progress or metrics to a file
# that can be viewed later, you could modify the code to:
# - Log metrics to a file (e.g., using the `logging` module or simply writing to a text file).
# - Use a logging framework like MLflow or TensorBoard (more complex setup).

# Example of writing results to a file (add this after the model training loop):
import json

# Optional: Save results to a JSON file
with open('model_training_results.json', 'w') as f:
    json.dump(results, f, indent=2)

print("\nModel training results saved to 'model_training_results.json'")

# This file (`model_training_results.json`) would then contain the data
# that was printed to the console, allowing it to be easily shared or analyzed.
# You could then download this file from Colab's file browser.
...

```

File "[<ipython-input-19-32ca89cc1b6f>](#)", line 80
 ^
 SyntaxError: invalid syntax

Next steps: [Fix error](#) [Explain error](#)

) within a Python code cell. In Jupyter notebooks and standard Python scripts, triple

The user likely intended to `*show*` what the output of the previous code block would

Suggested Changes

Remove the triple backticks and the content they enclose from the Python code cell.

File "[<ipython-input-20-844fc07b8d70>](#)", line 1
) within a Python code cell. In Jupyter notebooks and standard Python scripts, triple backticks are typically used in Markdown cells or multi-line strings for formatting code examples. They are not valid Python syntax for directly embedding output or comments in the way they are used here.

SyntaxError: unmatched ')'

Next steps:

[Fix error](#)

[Explain error](#)

prompt: include screenshots of model training outputs

Since the previous code block already performs model training and prints the eval
 # the "screenshots" would typically be the output text generated by the print staten
 # showing the MSE, RMSE, and R2 for each model.

There isn't a direct way to embed actual graphical "screenshots" of the terminal c
 # within a Python script or a Jupyter Notebook output cell programmatically in a st

However, you can `*represent*` the expected output format or add comments
 # indicating where the output would appear when the code is run.

The provided code already does this by printing the evaluation metrics for each mc
 # When you run the cell containing the model training code, the output will look
 # similar to this (values will vary based on your data and splits):

```
"""
Training Linear Regression...
Linear Regression Evaluation:
  Mean Squared Error (MSE): 1.2345
  Root Mean Squared Error (RMSE): 1.1111
  R-squared (R2): 0.8500
```

```
Training Decision Tree Regressor...
Decision Tree Regressor Evaluation:
  Mean Squared Error (MSE): 0.8765
  Root Mean Squared Error (RMSE): 0.9362
  R-squared (R2): 0.9000
```

```
Training Random Forest Regressor...
Random Forest Regressor Evaluation:
  Mean Squared Error (MSE): 0.5432
  Root Mean Squared Error (RMSE): 0.7370
  R-squared (R2): 0.9500
```

```
Training Gradient Boosting Regressor...
Gradient Boosting Regressor Evaluation:
  Mean Squared Error (MSE): 0.4567
  Root Mean Squared Error (RMSE): 0.6760
  R-squared (R2): 0.9600
```

```
--- Model Comparison ---
Linear Regression:
  MSE: 1.2345, RMSE: 1.1111, R2: 0.8500
Decision Tree Regressor:
  MSE: 0.8765, RMSE: 0.9362, R2: 0.9000
Random Forest Regressor:
  MSE: 0.5432, RMSE: 0.7370, R2: 0.9500
Gradient Boosting Regressor:
  MSE: 0.4567, RMSE: 0.6760, R2: 0.9600
```

```
Best model based on R2 score: Gradient Boosting Regressor
"""
```

To effectively "include screenshots" in a report or documentation derived from a C
 # 1. Run the notebook cells.
 # 2. Take actual screenshots of the output cells showing the printed metrics.
 # 3. Insert these image screenshots into your documentation.

Within the notebook itself, the output is automatically displayed below the code c
 # when it is executed. No additional code is needed beyond the existing print staten

If the request implies saving the model training progress or metrics to a file
 # that can be viewed later, you could modify the code to:

```
# - Log metrics to a file (e.g., using the `logging` module or simply writing to a file)
# - Use a logging framework like MLflow or TensorBoard (more complex setup).

# Example of writing results to a file (add this after the model training loop):
import json

# Optional: Save results to a JSON file
with open('model_training_results.json', 'w') as f:
    json.dump(results, f, indent=2)

print("\nModel training results saved to 'model_training_results.json'")

# This file (`model_training_results.json`) would then contain the data
# that was printed to the console, allowing it to be easily shared or analyzed.
# You could then download this file from Colab's file browser.
```



```
-----
NameError                                Traceback (most recent call last)
<ipython-input-21-aa5639c10d47> in <cell line: 0>()
    71 # Optional: Save results to a JSON file
    72 with open('model_training_results.json', 'w') as f:
--> 73     json.dump(results, f, indent=2)
    74
    75 print("\nModel training results saved to
'model_training_results.json'")

NameError: name 'results' is not defined
```

Next steps: [Explain error](#)

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
# from sklearn.svm import SVR # Commented out as it was in the original code
from sklearn.metrics import mean_squared_error, r2_score
import json # Import json module

# Assuming 'CO(GT)' is the target variable and you have a DataFrame 'df'
# after completing the previous steps (loading, cleaning, feature engineering, selection)

# Make sure the target column and features are numeric and handle NaNs
target_column = 'CO(GT)'

# Select only numeric columns for modeling
# You might want to refine this selection based on your feature engineering/selection
# For this example, let's use all numerical columns except the target
numerical_cols = df.select_dtypes(include=[np.number]).columns
feature_cols = numerical_cols.drop(target_column, errors='ignore').tolist()

# Ensure the target column is in the DataFrame
if target_column not in df.columns:
    print(f"Error: Target column '{target_column}' not found in the DataFrame.")
else:
    X = df[feature_cols]
    y = df[target_column]

    # Handle NaNs in features and target before splitting
    # A common approach is to drop rows with NaNs in features or target
    # Alternatively, you could impute NaNs specifically for modeling
    # Let's drop rows with NaNs in this example for simplicity
    data_for_modeling = pd.concat([X, y], axis=1).dropna()
    X = data_for_modeling[feature_cols]
    y = data_for_modeling[target_column]

    if X.empty or y.empty:
        print("Error: DataFrame is empty after handling NaNs for modeling.")
    else:
        # Split the data into training and testing sets
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

        # Define a list of models to try
        models = {
            "Linear Regression": LinearRegression(),
            "Decision Tree Regressor": DecisionTreeRegressor(random_state=42),
            "Random Forest Regressor": RandomForestRegressor(n_estimators=100, random_state=42),
            "Gradient Boosting Regressor": GradientBoostingRegressor(n_estimators=100, random_state=42),
            # SVR can be computationally expensive and requires feature scaling
            # "Support Vector Regressor": SVR(kernel='rbf')
        }
```

```

# Initialize the results dictionary BEFORE the loop
results = {}

# Train and evaluate each model
for name, model in models.items():
    print(f"\nTraining {name}...")
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)

    # Evaluate the model
    mse = mean_squared_error(y_test, y_pred)
    rmse = np.sqrt(mse) # Root Mean Squared Error
    r2 = r2_score(y_test, y_pred)

    results[name] = {
        "MSE": mse,
        "RMSE": rmse,
        "R2": r2
    }


    print(f"{name} Evaluation:")
    print(f"  Mean Squared Error (MSE): {mse:.4f}")
    print(f"  Root Mean Squared Error (RMSE): {rmse:.4f}")
    print(f"  R-squared (R2): {r2:.4f}")

# Print a summary of the results
print("\n--- Model Comparison ---")
for name, metrics in results.items():
    print(f"{name}:")
    print(f"  MSE: {metrics['MSE']:.4f}, RMSE: {metrics['RMSE']:.4f}, R2: {m")

# Optional: Find the best model based on a metric (e.g., R2 score)
if results: # Check if results is not empty
    best_model_name = max(results, key=lambda k: results[k]['R2'])
    print(f"\nBest model based on R2 score: {best_model_name}")
else:
    print("\nNo model results available.")

# Optional: Save results to a JSON file
if results: # Check if results is not empty before saving
    with open('model_training_results.json', 'w') as f:
        json.dump(results, f, indent=2)
    print("\nModel training results saved to 'model_training_results.json'")
else:
    print("\nNo results to save to JSON file.")

```

 Error: DataFrame is empty after handling NaNs for modeling.

 Generate

confusion matrix



Close

< 1 of 1 >   [Use code with caution](#)

prompt: confusion matrix

```

import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# Assuming you have a trained classification model and have made predictions
# This requires a classification task, whereas the previous code was for regression.
# To demonstrate a confusion matrix, we need:
# 1. A classification dataset (or convert the regression problem into a classification)
# 2. A classification model.
# 3. True labels (y_test_clf) and predicted labels (y_pred_clf).

# For demonstration, let's simulate a classification scenario based on the CO(GT) cc
# by creating a binary classification target (e.g., High CO vs Low CO)
# and using a simple classification model like Logistic Regression or a Decision Tree

# --- SIMULATING CLASSIFICATION DATA ---
# Let's define a threshold for CO(GT) to create a binary target variable
# This is a simplified example; in a real scenario, you would have a true classification
if 'CO(GT)' in df.columns:
    # Ensure CO(GT) is numeric and drop NaNs for this example
    df['CO(GT)_numeric'] = pd.to_numeric(df['CO(GT)'], errors='coerce')
    df_clf = df.dropna(subset=['CO(GT)_numeric']).copy()

    if not df_clf.empty:

```

```

# Define a threshold (e.g., the median)
co_threshold = df_clf['CO(GT)_numeric'].median()
print(f"\nUsing median CO(GT) ({co_threshold:.2f}) as threshold for classifi

# Create the binary target: 1 for High CO, 0 for Low CO
df_clf['CO_High'] = (df_clf['CO(GT)_numeric'] > co_threshold).astype(int)

# Select features (using the same numeric features as the regression example)
numerical_cols_clf = df_clf.select_dtypes(include=[np.number]).columns
# Exclude the original CO(GT)_numeric and the new CO_High target
feature_cols_clf = numerical_cols_clf.drop(['CO(GT)_numeric', 'CO_High'], er

X_clf = df_clf[feature_cols_clf]
y_clf = df_clf['CO_High']

# Drop rows with any NaNs in features before splitting
data_for_modeling_clf = pd.concat([X_clf, y_clf], axis=1).dropna()
X_clf = data_for_modeling_clf[feature_cols_clf]
y_clf = data_for_modeling_clf['CO_High']

if X_clf.empty or y_clf.empty:
    print("\nError: DataFrame is empty after handling NaNs for classificati
else:
    # Split the data into training and testing sets for classification
    X_train_clf, X_test_clf, y_train_clf, y_test_clf = train_test_split(X_cl

    # --- TRAIN A CLASSIFICATION MODEL ---
    # Using a simple Decision Tree Classifier for demonstration
    from sklearn.tree import DecisionTreeClassifier
    from sklearn.metrics import accuracy_score, classification_report

    clf_model = DecisionTreeClassifier(random_state=42)

    print("\nTraining Decision Tree Classifier for simulated classification.
    clf_model.fit(X_train_clf, y_train_clf)
    y_pred_clf = clf_model.predict(X_test_clf)

    # --- GENERATE AND PLOT CONFUSION MATRIX ---
    print("\nGenerating Confusion Matrix:")

    # Calculate the confusion matrix
    cm = confusion_matrix(y_test_clf, y_pred_clf)

    # Display the confusion matrix using ConfusionMatrixDisplay
    # Class names correspond to the integer labels (0 and 1 in this case)
    class_names = ['Low CO', 'High CO'] # Replace with actual class names if

    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=class_

    # Plot the confusion matrix
    plt.figure(figsize=(8, 6)) # Adjust figure size if needed
    disp.plot(cmap=plt.cm.Blues) # Use a colormap like 'Blues'
    plt.title('Confusion Matrix')
    plt.show()

    # --- OPTIONAL: PRINT CLASSIFICATION REPORT ---
    # Provides other metrics like Precision, Recall, F1-score
    print("\nClassification Report:")
    print(classification_report(y_test_clf, y_pred_clf, target_names=class_r

    # --- OPTIONAL: PRINT ACCURACY ---
    accuracy = accuracy_score(y_test_clf, y_pred_clf)
    print(f"\nAccuracy: {accuracy:.4f}")

else:
    print("\nCannot simulate classification: 'CO(GT)' column is empty or contain

else:
    print("\nCannot simulate classification: 'CO(GT)' column not found in the DataFr

# Note: The confusion matrix is only applicable for classification tasks.
# The previous code block focused on regression models (predicting a continuous val
# If your actual task is regression, a confusion matrix is not the appropriate evalu
# For regression, metrics like MSE, RMSE, R2 are used, which were already calculate

```



Using median CO(GT) (-200.00) as threshold for classification.

Error: DataFrame is empty after handling NaNs for classification modeling simul

```

Generate Close
< 1 of 1 >   Use code with caution

```

prompt: error analysis or model comparison table

```

import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
# Note: The previous code block already performs model training and prints the evaluation
# The "screenshots" you asked for would typically be the output text generated by the code
# when the code is executed.

# There isn't a direct way to embed actual graphical "screenshots" of console output
# within a Python code cell in a standard manner. The output appears below the cell.

# If you want to present a table summarizing the model comparison, you can create one
# using the 'results' dictionary that was generated in the previous code block.

# Check if the 'results' dictionary exists and is not empty
if 'results' in locals() and results:
    print("\n--- Model Comparison Table ---")

    # Create a Pandas DataFrame from the results dictionary
    results_df = pd.DataFrame(results).T # .T transposes the DataFrame

    # Format the columns for better readability
    results_df_formatted = results_df.applymap('{:.4f}'.format)

    # Print the formatted table
    print(results_df_formatted)

    # Optional: Display the DataFrame (this will be a nicely formatted table in Colab)
    # from IPython.display import display
    # display(results_df_formatted)

    print("\n--- End of Model Comparison Table ---")
else:
    print("\nNo model comparison results available to display a table.")

# For error analysis in regression, you would typically look at:
# 1. Residual plots: Plot the errors (residuals) vs. predicted values or vs. features
#    - Homoscedasticity (errors have constant variance) vs. Heteroscedasticity (error variance changes)
#    - Patterns in residuals might indicate the model is missing important information
# 2. Distribution of residuals: Histograms or Q-Q plots to see if residuals are normally distributed
# 3. Large errors: Identify specific data points where the model performed poorly and explain why

# Example of a Residual Plot (for one of the trained models, e.g., the best one)
if 'best_model_name' in locals() and best_model_name in models:
    best_model = models[best_model_name]
    y_pred_best = best_model.predict(X_test)
    residuals = y_test - y_pred_best

    plt.figure(figsize=(10, 6))
    plt.scatter(y_pred_best, residuals, alpha=0.5)
    plt.axhline(0, color='red', linestyle='--') # Add a horizontal line at 0
    plt.title(f'Residual Plot for {best_model_name}')
    plt.xlabel('Predicted Values')
    plt.ylabel('Residuals (True - Predicted)')
    plt.grid(True)
    plt.show()

    # Example of a Histogram of Residuals
    plt.figure(figsize=(10, 6))
    sns.histplot(residuals, kde=True, bins=30)
    plt.title(f'Distribution of Residuals for {best_model_name}')
    plt.xlabel('Residuals')
    plt.ylabel('Frequency')
    plt.show()

    # Example: Identify data points with large residuals (e.g., absolute residual > 3 * rmse)
    rmse_best = results[best_model_name]['RMSE']
    large_error_threshold = 3 * rmse_best
    large_error_indices = y_test[np.abs(residuals) > large_error_threshold].index

    print(f"\nIdentifying data points with large residuals (>{large_error_threshold:.2f}).")
    if not large_error_indices.empty:
        print(f"Found {len(large_error_indices)} data points with large errors.")
        # You can inspect these rows in the original DataFrame or test set
        # print(df.loc[large_error_indices]) # Be careful if indexing needs adjustment
        # print(X_test.loc[large_error_indices])
        # print(y_test.loc[large_error_indices])
    else:
        print("No data points with large residuals identified.")

```

```

print("No data points found with residuals exceeding the threshold.")

# For error analysis in classification (using the simulated classification example):
# 1. Confusion Matrix (already generated in the previous code block)
#   - Analyze False Positives (predicting positive when negative) and False Negative
#   - Understand which types of errors the model makes.
# 2. Classification Report (already generated)
#   - Precision (of those predicted positive, how many were actually positive)
#   - Recall (of those actually positive, how many were predicted positive)
#   - F1-score (harmonic mean of precision and recall)
#   - Support (number of instances in each class)
# 3. ROC Curve and AUC (for binary classification)
#   - Evaluates the model's ability to distinguish between classes at various thresh
# 4. Examining Misclassified Instances
#   - Similar to looking at large residuals in regression, identify data points that


# Example of examining misclassified instances (using the simulated classification)
if 'clf_model' in locals() and 'X_test_clf' in locals() and 'y_test_clf' in locals():
    misclassified_indices_clf = X_test_clf.index[y_test_clf != y_pred_clf]


    print(f"\nIdentifying misclassified data points for the simulated classification
    if not misclassified_indices_clf.empty:
        print(f"Found {len(misclassified_indices_clf)} misclassified data points.")
        # You can inspect these rows
        # print(df_clf.loc[misclassified_indices_clf]) # Assuming df_clf has the ori
        # print(X_test_clf.loc[misclassified_indices_clf])
        # print(y_test_clf.loc[misclassified_indices_clf])
        # print(y_pred_clf[y_test_clf != y_pred_clf]) # The predicted labels for mis
    else:
        print("No misclassified data points found in the test set.")



```



No model comparison results available to display a table.

 **Generate**

 Close

< 1 of 1 >   [Use code with caution](#)

```

# prompt: include all screenshots of outputs

import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
# Note: The previous code block already performs model training and prints the eval
# The "screenshots" you asked for would typically be the output text generated by th
# when the code is executed.

# There isn't a direct way to embed actual graphical "screenshots" of console output
# within a Python code cell in a standard manner. The output appears below the cell.

# If you want to present a table summarizing the model comparison, you can create or
# using the 'results' dictionary that was generated in the previous code block.

# Check if the 'results' dictionary exists and is not empty
if 'results' in locals() and results:
    print("\n--- Model Comparison Table ---")

    # Create a Pandas DataFrame from the results dictionary
    results_df = pd.DataFrame(results).T # .T transposes the DataFrame

    # Format the columns for better readability
    results_df_formatted = results_df.applymap('{:.4f}'.format)

    # Print the formatted table
    print(results_df_formatted)

    # Optional: Display the DataFrame (this will be a nicely formatted table in Col
    # from IPython.display import display
    # display(results_df_formatted)

    print("\n--- End of Model Comparison Table ---")
else:
    print("\nNo model comparison results available to display a table.")

# For error analysis in regression, you would typically look at:
# 1. Residual plots: Plot the errors (residuals) vs. predicted values or vs. feature
#   - Homoscedasticity (errors have constant variance) vs. Heteroscedasticity (err
#   - Patterns in residuals might indicate the model is missing important informati

```



```
# 2. Distribution of residuals: Histograms or Q-Q plots to see if residuals are norm
# 3. Large errors: Identify specific data points where the model performed poorly ar

# Example of a Residual Plot (for one of the trained models, e.g., the best one)
if 'best_model_name' in locals() and best_model_name in models:
    best_model = models[best_model_name]
    y_pred_best = best_model.predict(X_test)
    residuals = y_test - y_pred_best

    plt.figure(figsize=(10, 6))
    plt.scatter(y_pred_best, residuals, alpha=0.5)
    plt.axhline(0, color='red', linestyle='--') # Add a horizontal line at 0
    plt.title(f'Residual Plot for {best_model_name}')
    plt.xlabel('Predicted Values')
    plt.ylabel('Residuals (True - Predicted)')
    plt.grid(True)
    plt.show()

# Example of a Histogram of Residuals
plt.figure(figsize=(10, 6))
sns.histplot(residuals, kde=True, bins=30)
plt.title(f'Distribution of Residuals for {best_model_name}')
plt.xlabel('Residuals')
plt.ylabel('Frequency')
plt.show()

# Example: Identify data points with large residuals (e.g., absolute residual >
rmse_best = results[best_model_name]['RMSE']
large_error_threshold = 3 * rmse_best
large_error_indices = y_test[np.abs(residuals) > large_error_threshold].index

print(f"\nIdentifying data points with large residuals (>{large_error_threshold}:
if not large_error_indices.empty:
    print(f"Found {len(large_error_indices)} data points with large errors.")
    # You can inspect these rows in the original DataFrame or test set
    # print(df.loc[large_error_indices]) # Be careful if indexing needs adjustme
    # print(X_test.loc[large_error_indices])
    # print(y_test.loc[large_error_indices])
    # print(y_pred_best[np.abs(residuals) > large_error_threshold]) # The predic
else:
    print("No data points found with residuals exceeding the threshold.")

# For error analysis in classification (using the simulated classification example):
# 1. Confusion Matrix (already generated in the previous code block)
# - Analyze False Positives (predicting positive when negative) and False Negativ
# - Understand which types of errors the model makes.
# 2. Classification Report (already generated)
# - Precision (of those predicted positive, how many were actually positive)
# - Recall (of those actually positive, how many were predicted positive)
# - F1-score (harmonic mean of precision and recall)
# - Support (number of instances in each class)
# 3. ROC Curve and AUC (for binary classification)
# - Evaluates the model's ability to distinguish between classes at various thresh
# 4. Examining Misclassified Instances
# - Similar to looking at large residuals in regression, identify data points tha

# Example of examining misclassified instances (using the simulated classification)
if 'clf_model' in locals() and 'X_test_clf' in locals() and 'y_test_clf' in locals():
    misclassified_indices_clf = X_test_clf.index[y_test_clf != y_pred_clf]

    print(f"\nIdentifying misclassified data points for the simulated classificati
    if not misclassified_indices_clf.empty:
        print(f"Found {len(misclassified_indices_clf)} misclassified data points.")
        # You can inspect these rows
        # print(df_clf.loc[misclassified_indices_clf]) # Assuming df_clf has the or
        # print(X_test_clf.loc[misclassified_indices_clf])
        # print(y_test_clf.loc[misclassified_indices_clf])
        # print(y_pred_clf[y_test_clf != y_pred_clf]) # The predicted labels for mi
    else:
        print("No misclassified data points found in the test set.")
```



No model comparison results available to display a table.

 Generate

try multiple models (baseline and advanced)



Close

< 1 of 1 >   [Use code with caution](#)

prompt: try multiple models (baseline and advanced)

```

import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
# Note: The previous code block already performs model training and prints the evaluation results.
# The "screenshots" you asked for would typically be the output text generated by the code
# when the code is executed.

# There isn't a direct way to embed actual graphical "screenshots" of console output
# within a Python code cell in a standard manner. The output appears below the cell.

# If you want to present a table summarizing the model comparison, you can create one
# using the 'results' dictionary that was generated in the previous code block.

# Check if the 'results' dictionary exists and is not empty
if 'results' in locals() and results:
    print("\n--- Model Comparison Table ---")

    # Create a Pandas DataFrame from the results dictionary
    results_df = pd.DataFrame(results).T # .T transposes the DataFrame

    # Format the columns for better readability
    results_df_formatted = results_df.applymap('{:.4f}'.format)

    # Print the formatted table
    print(results_df_formatted)

    # Optional: Display the DataFrame (this will be a nicely formatted table in Colab)
    # from IPython.display import display
    # display(results_df_formatted)

    print("\n--- End of Model Comparison Table ---")
else:
    print("\nNo model comparison results available to display a table.")

# For error analysis in regression, you would typically look at:
# 1. Residual plots: Plot the errors (residuals) vs. predicted values or vs. features
#    - Homoscedasticity (errors have constant variance) vs. Heteroscedasticity (error variance changes)
#    - Patterns in residuals might indicate the model is missing important information
# 2. Distribution of residuals: Histograms or Q-Q plots to see if residuals are normally distributed
# 3. Large errors: Identify specific data points where the model performed poorly and investigate why

# Example of a Residual Plot (for one of the trained models, e.g., the best one)
if 'best_model_name' in locals() and best_model_name in models:
    best_model = models[best_model_name]
    y_pred_best = best_model.predict(X_test)
    residuals = y_test - y_pred_best

    plt.figure(figsize=(10, 6))
    plt.scatter(y_pred_best, residuals, alpha=0.5)
    plt.axhline(0, color='red', linestyle='--') # Add a horizontal line at 0
    plt.title(f'Residual Plot for {best_model_name}')
    plt.xlabel('Predicted Values')
    plt.ylabel('Residuals (True - Predicted)')
    plt.grid(True)
    plt.show()

    # Example of a Histogram of Residuals
    plt.figure(figsize=(10, 6))
    sns.histplot(residuals, kde=True, bins=30)
    plt.title(f'Distribution of Residuals for {best_model_name}')
    plt.xlabel('Residuals')
    plt.ylabel('Frequency')
    plt.show()

    # Example: Identify data points with large residuals (e.g., absolute residual > 3 * rmse)
    rmse_best = results[best_model_name]['RMSE']
    large_error_threshold = 3 * rmse_best
    large_error_indices = y_test[np.abs(residuals) > large_error_threshold].index

    print(f"\nIdentifying data points with large residuals (>{large_error_threshold:.2f}).")
    if not large_error_indices.empty:
        print(f"Found {len(large_error_indices)} data points with large errors.")
        # You can inspect these rows in the original DataFrame or test set
        # print(df.loc[large_error_indices]) # Be careful if indexing needs adjustment
        # print(X_test.loc[large_error_indices])
        # print(y_test.loc[large_error_indices])
        # print(y_pred_best[large_error_indices]) # The predicted values for these points
    else:
        print("No data points found with residuals exceeding the threshold.")

# For error analysis in classification (using the simulated classification example):
# 1. Confusion Matrix (Printed in the previous code block)

```

```

# 1. Confusion Matrix (already generated in the previous code block)
# - Analyze False Positives (predicting positive when negative) and False Negative
# - Understand which types of errors the model makes.
# 2. Classification Report (already generated)
# - Precision (of those predicted positive, how many were actually positive)
# - Recall (of those actually positive, how many were predicted positive)
# - F1-score (harmonic mean of precision and recall)
# - Support (number of instances in each class)
# 3. ROC Curve and AUC (for binary classification)
# - Evaluates the model's ability to distinguish between classes at various thresh
# 4. Examining Misclassified Instances
# - Similar to looking at large residuals in regression, identify data points that

# Example of examining misclassified instances (using the simulated classification)
if 'clf_model' in locals() and 'X_test_clf' in locals() and 'y_test_clf' in locals():
    misclassified_indices_clf = X_test_clf.index[y_test_clf != y_pred_clf]

print(f"\nIdentifying misclassified data points for the simulated classification")
if not misclassified_indices_clf.empty:
    print(f"Found {len(misclassified_indices_clf)} misclassified data points.")
    # You can inspect these rows
    # print(df_clf.loc[misclassified_indices_clf]) # Assuming df_clf has the ori
    # print(X_test_clf.loc[misclassified_indices_clf])
    # print(y_test_clf.loc[misclassified_indices_clf])
    # print(y_pred_clf[y_test_clf != y_pred_clf]) # The predicted labels for mis
else:
    print("No misclassified instances found.")

```

Enter a prompt here



0 / 2000

Gemini can make mistakes so double-check responses and use code with caution. [Learn more](#)