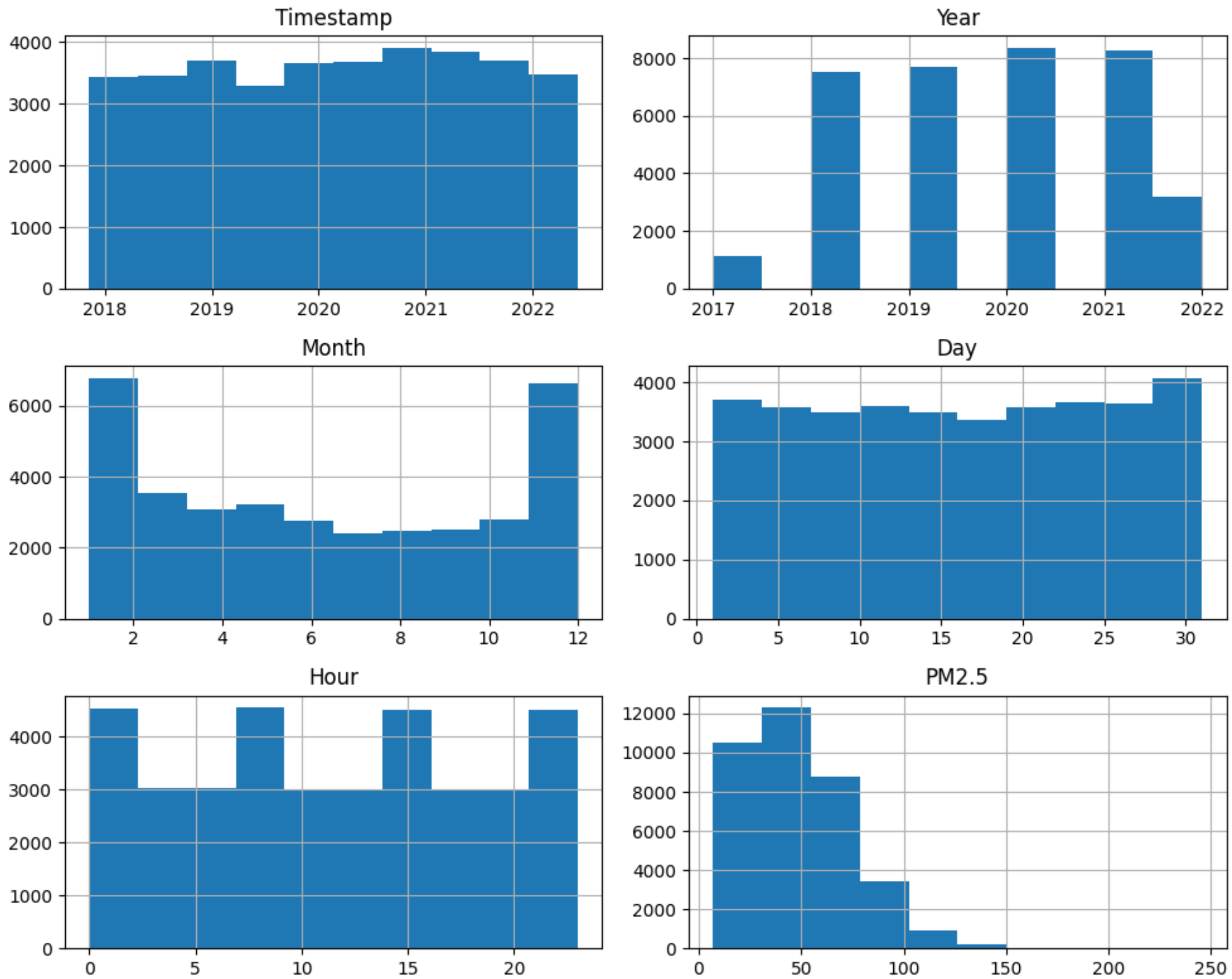
```python
import pandas as pd
data=pd.read_csv('/content/air-quality-india.csv')

import pandas as pd

# Load the dataset
data = pd.read_csv('/content/air-quality-india.csv')

# Handle missing values (example: filling with mean for numerical,
mode for categorical)
# Replace 'numerical_column' and 'categorical_column' with actual
column names
for col in data.columns:
    if data[col].dtype in ['int64', 'float64']:
        data[col].fillna(data[col].mean(), inplace=True)
    elif data[col].dtype == 'object':
        data[col].fillna(data[col].mode()[0], inplace=True)

# Handle duplicates
data.drop_duplicates(inplace=True)

# Handle inconsistent timestamp formats (assuming a column named
'Date')
# Replace 'Date' with the actual timestamp column name if different
# Changed 'Date' to 'Timestamp' based on available columns
data['Timestamp'] = pd.to_datetime(data['Timestamp'], errors='coerce')

# Display the first few rows and info after cleaning
print("Data after cleaning:")
print(data.head())
print("\nData Info after cleaning:")
data.info()
```

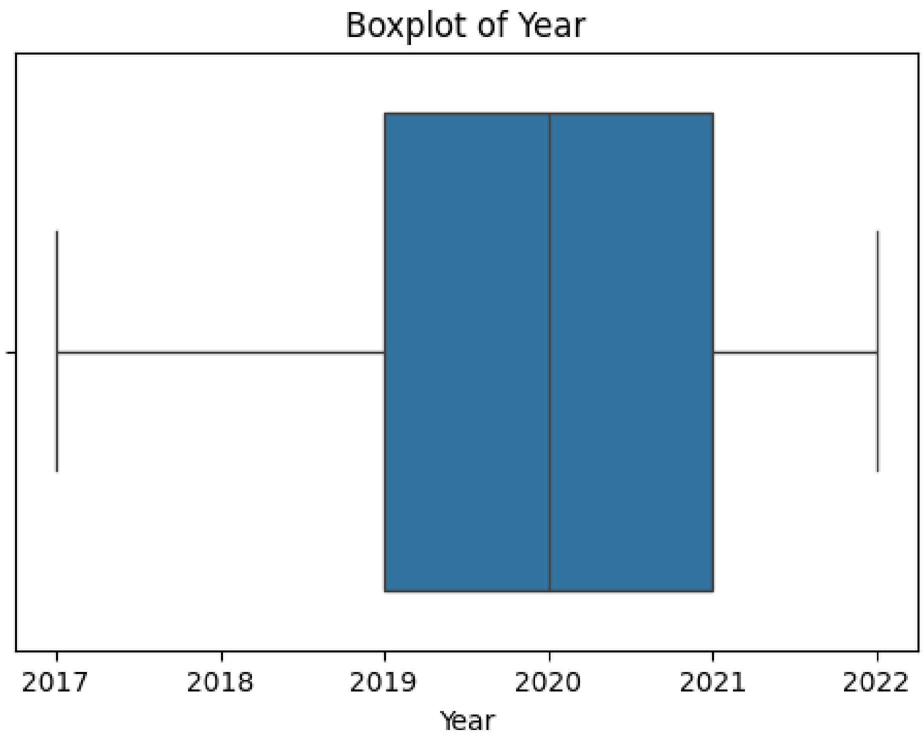
```
Data after cleaning:
            Timestamp  Year  Month  Day  Hour   PM2.5
0 2017-11-07 12:00:00  2017     11    7    12   64.51
1 2017-11-07 13:00:00  2017     11    7    13   69.95
2 2017-11-07 14:00:00  2017     11    7    14   92.79
3 2017-11-07 15:00:00  2017     11    7    15  109.66
4 2017-11-07 16:00:00  2017     11    7    16  116.50

Data Info after cleaning:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 36192 entries, 0 to 36191
Data columns (total 6 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   Timestamp  36192 non-null  datetime64[ns]
 1   Year       36192 non-null  int64
 2   Month      36192 non-null  int64
```

```
 3   Day         36192 non-null  int64
 4   Hour        36192 non-null  int64
 5   PM2.5       36192 non-null  float64
dtypes: datetime64[ns](1), float64(1), int64(4)
memory usage: 1.7 MB

<ipython-input-5-3cbb76a2987e>:12: FutureWarning: A value is trying to
be set on a copy of a DataFrame or Series through chained assignment
using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never
work because the intermediate object on which we are setting values
always behaves as a copy.

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


  data[col].fillna(data[col].mode()[0], inplace=True)
<ipython-input-5-3cbb76a2987e>:10: FutureWarning: A value is trying to
be set on a copy of a DataFrame or Series through chained assignment
using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never
work because the intermediate object on which we are setting values
always behaves as a copy.

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


  data[col].fillna(data[col].mean(), inplace=True)

# %% [markdown]
# Exploratory Data Analysis (EDA)
# - Use histograms, boxplots, heatmaps to understand distributions and
correlations

# %%
import matplotlib.pyplot as plt
import seaborn as sns

# Histograms for numerical columns
print("Histograms for numerical columns:")
data.hist(figsize=(10, 8))
plt.tight_layout()
plt.show()

# Boxplots for numerical columns to identify outliers
```

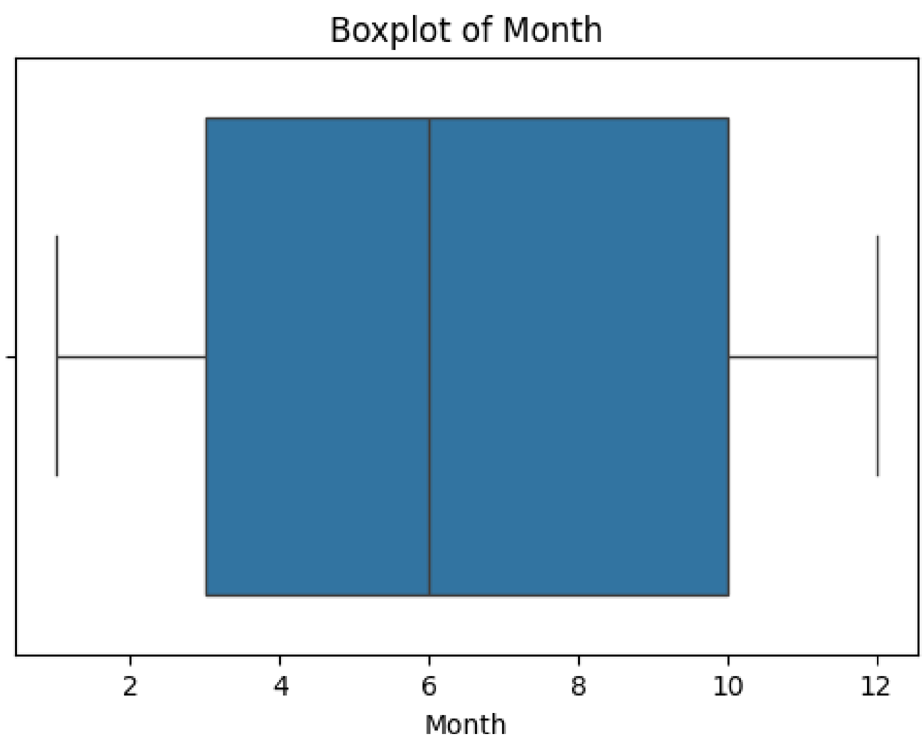
```
print("\nBoxplots for numerical columns:")
numerical_cols = data.select_dtypes(include=['int64',
'float64']).columns
for col in numerical_cols:
    plt.figure(figsize=(6, 4))
    sns.boxplot(x=data[col])
    plt.title(f'Boxplot of {col}')
    plt.show()

# Heatmap to show correlations between numerical columns
print("\nHeatmap of correlations:")
correlation_matrix = data.select_dtypes(include=['int64',
'float64']).corr()
plt.figure(figsize=(8, 6))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm',
fmt=".2f")
plt.title('Correlation Matrix')
plt.show()

# Example of a categorical variable analysis (if you have relevant
categorical columns)
# Replace 'categorical_column' with an actual categorical column name
if available
# For example, if 'City' or 'Location' was a column
# if 'City' in data.columns:
#     print("\nValue counts for City:")
#     print(data['City'].value_counts())
#
#     plt.figure(figsize=(12, 6))
#     sns.countplot(y='City', data=data,
order=data['City'].value_counts().index)
#     plt.title('Count of records per City')
#     plt.show()

Histograms for numerical columns:
```



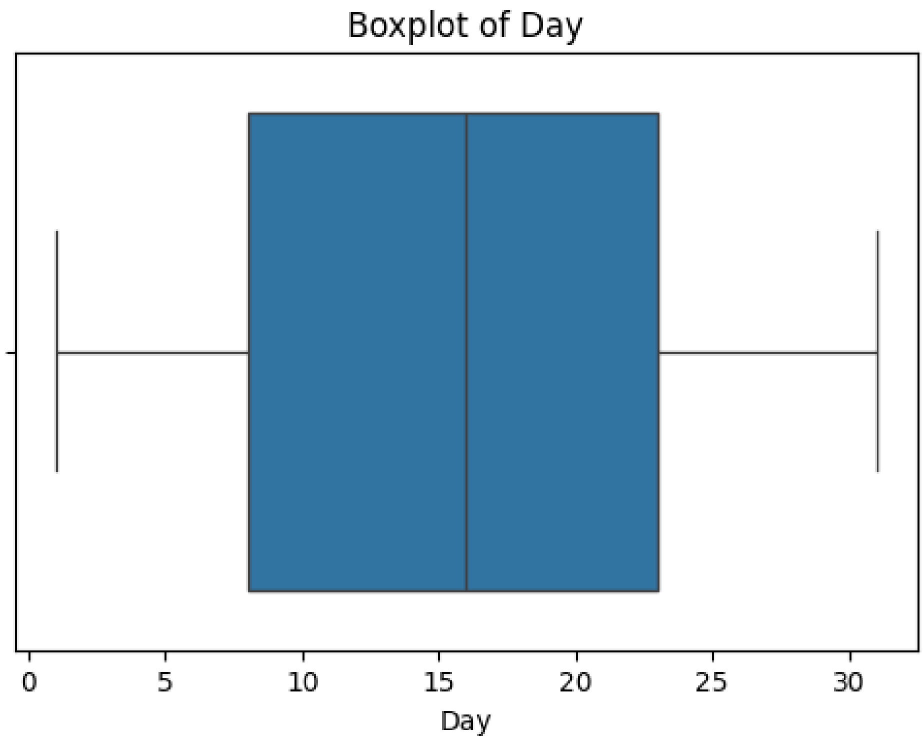

```
Boxplots for numerical columns:
```

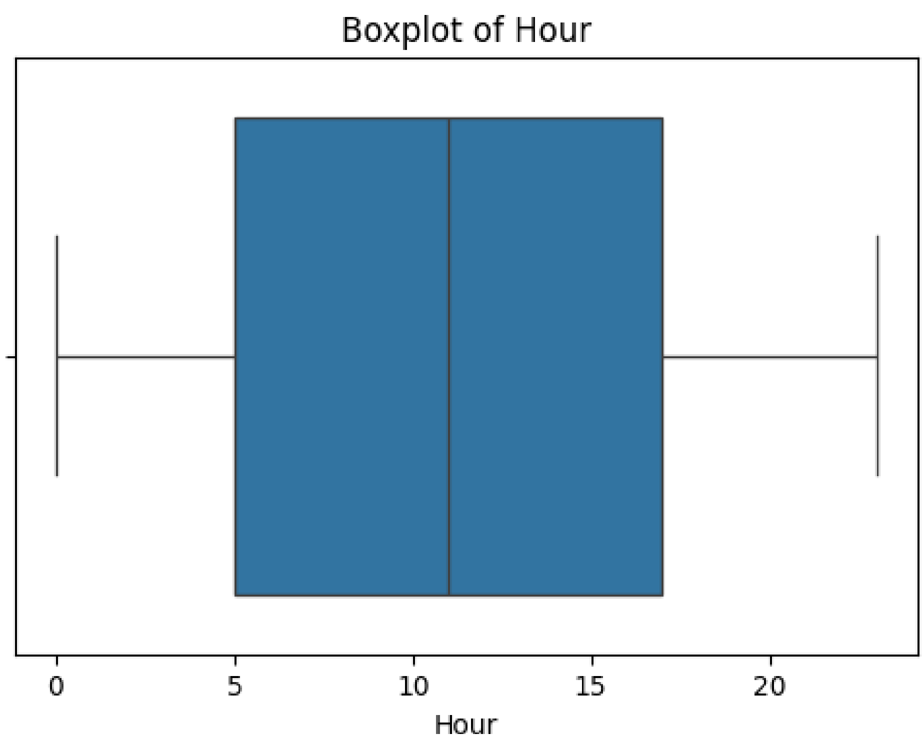


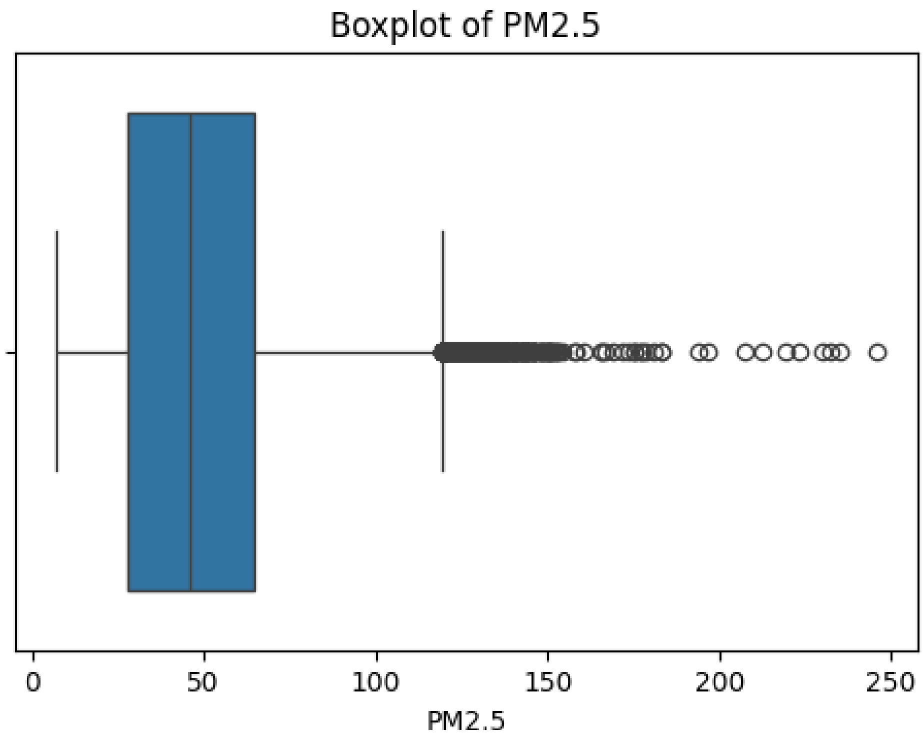
Boxplot of Year
2017
2018
2019
2020
2021
2022
Year



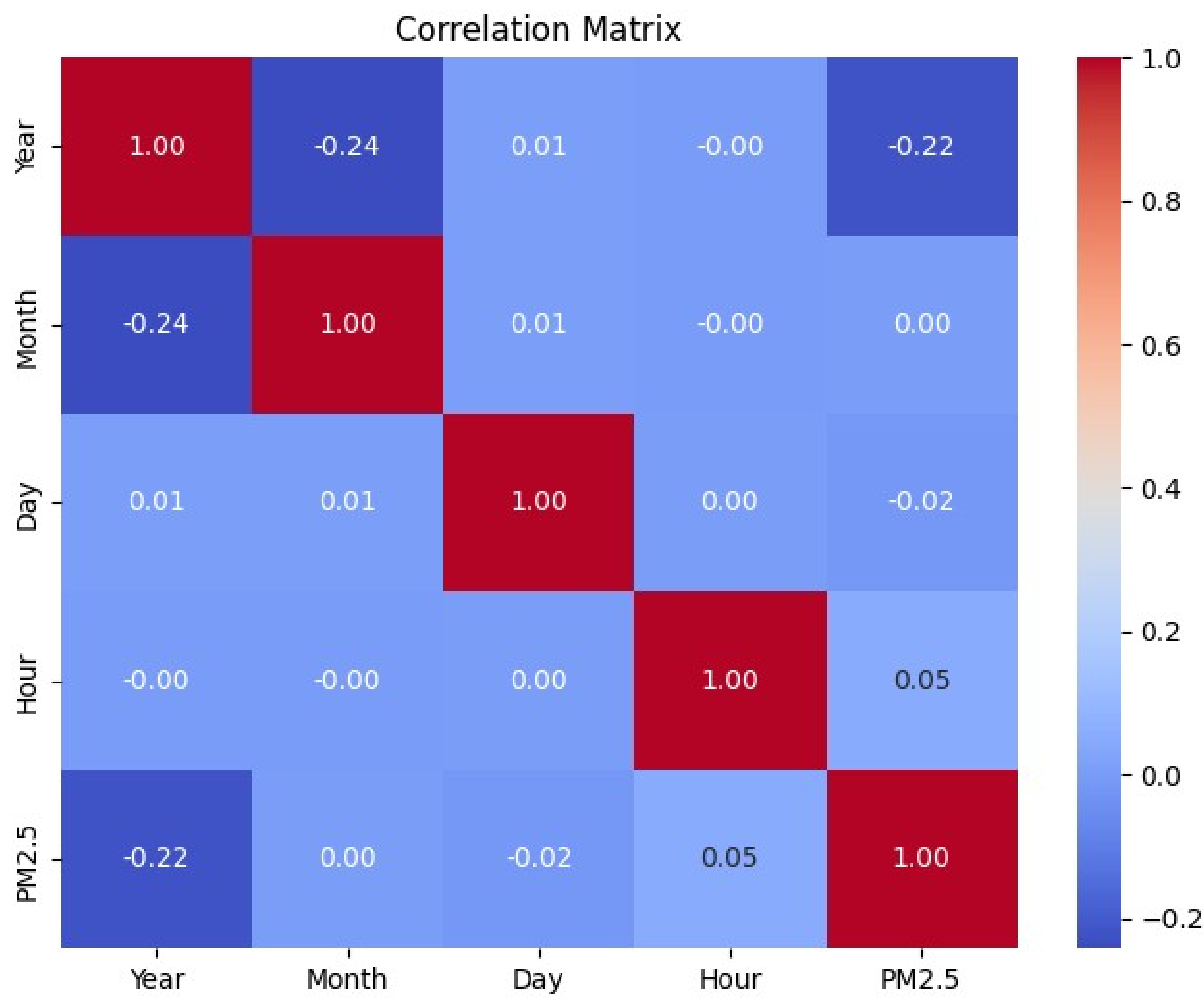
Boxplot of Month
2
4
6
8
10
12
Month


Boxplot of Day
0
5
10
15
20
25
30
Day


Boxplot of Hour
0
5
10
15
20
Hour




```
Heatmap of correlations:
```




```
# %% [markdown]
# Feature Engineering
# - Create lag features, rolling averages, time-based features

# %%
# Ensure the data is sorted by the timestamp column for time-series
features
# Replace 'Timestamp' with your actual timestamp column name if
different
data = data.sort_values(by='Timestamp')

# Create Lag Features
# Lag 1 feature for 'PM2.5' (example - replace with relevant numerical
columns)
# This will show the PM2.5 value from the previous time step
if 'PM2.5' in data.columns:
    data['PM2.5_lag1'] = data['PM2.5'].shift(1)

# Create Rolling Averages
```

```python
# Rolling 3-step average for 'PM2.5'
if 'PM2.5' in data.columns:
    data['PM2.5_rolling_avg_3'] =
data['PM2.5'].rolling(window=3).mean()

# Create Time-Based Features
# Extract components from the timestamp
data['Hour'] = data['Timestamp'].dt.hour
data['DayOfWeek'] = data['Timestamp'].dt.dayofweek
data['Month'] = data['Timestamp'].dt.month

# Display the first few rows with new features
print("Data after Feature Engineering:")
print(data.head())
print("\nData Info after Feature Engineering:")
data.info()
```

```
Data after Feature Engineering:
            Timestamp  Year  Month  Day  Hour   PM2.5  PM2.5_lag1  \
0 2017-11-07 12:00:00  2017     11    7    12   64.51         NaN
1 2017-11-07 13:00:00  2017     11    7    13   69.95       64.51
2 2017-11-07 14:00:00  2017     11    7    14   92.79       69.95
3 2017-11-07 15:00:00  2017     11    7    15  109.66       92.79
4 2017-11-07 16:00:00  2017     11    7    16  116.50      109.66

   PM2.5_rolling_avg_3  DayOfWeek
0                  NaN          1
1                  NaN          1
2            75.750000          1
3            90.800000          1
4           106.316667          1

Data Info after Feature Engineering:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 36192 entries, 0 to 36191
Data columns (total 9 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   Timestamp            36192 non-null  datetime64[ns]
 1   Year                 36192 non-null  int64
 2   Month                36192 non-null  int32
 3   Day                  36192 non-null  int64
 4   Hour                 36192 non-null  int32
 5   PM2.5                36192 non-null  float64
 6   PM2.5_lag1           36191 non-null  float64
 7   PM2.5_rolling_avg_3  36190 non-null  float64
 8   DayOfWeek            36192 non-null  int32
dtypes: datetime64[ns](1), float64(3), int32(3), int64(2)
memory usage: 2.1 MB
```

```
# %% [markdown]
# Model Building
# - Random Forest, Gradient Boosting, XGBoost, LSTM (if time-series).

# %%
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor,
GradientBoostingRegressor
import xgboost as xgb
from sklearn.metrics import mean_squared_error, r2_score
import numpy as np

# Define features (X) and target (y)
# Assuming 'PM2.5' is the target variable and others are features
# Drop the original 'PM2.5' column from features if it exists
features = ['Year', 'Month', 'DayOfWeek', 'Hour', 'PM2.5_lag1',
'PM2.5_rolling_avg_3'] # Include engineered features
target = 'PM2.5'

# Handle potential NaNs created by lag/rolling features by dropping
rows
data_model = data.dropna(subset=features + [target]).copy()

X = data_model[features]
y = data_model[target]

# Split 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, shuffle=False) # shuffle=False for
time-series

print("Training data shape:", X_train.shape, y_train.shape)
print("Testing data shape:", X_test.shape, y_test.shape)

# --- Model 1: Random Forest Regressor ---
print("\n--- Random Forest Regressor ---")
rf_model = RandomForestRegressor(n_estimators=100, random_state=42,
n_jobs=-1)
rf_model.fit(X_train, y_train)
rf_predictions = rf_model.predict(X_test)

rf_mse = mean_squared_error(y_test, rf_predictions)
rf_rmse = np.sqrt(rf_mse)
rf_r2 = r2_score(y_test, rf_predictions)

print(f"Random Forest - Mean Squared Error (MSE): {rf_mse:.2f}")
print(f"Random Forest - Root Mean Squared Error (RMSE):
{rf_rmse:.2f}")
print(f"Random Forest - R-squared (R2): {rf_r2:.2f}")

# --- Model 2: Gradient Boosting Regressor ---
```

```
print("\n--- Gradient Boosting Regressor ---")
gb_model = GradientBoostingRegressor(n_estimators=100,
learning_rate=0.1, max_depth=3, random_state=42)
gb_model.fit(X_train, y_train)
gb_predictions = gb_model.predict(X_test)

gb_mse = mean_squared_error(y_test, gb_predictions)
gb_rmse = np.sqrt(gb_mse)
gb_r2 = r2_score(y_test, gb_predictions)

print(f"Gradient Boosting - Mean Squared Error (MSE): {gb_mse:.2f}")
print(f"Gradient Boosting - Root Mean Squared Error (RMSE):
{gb_rmse:.2f}")
print(f"Gradient Boosting - R-squared (R2): {gb_r2:.2f}")

# --- Model 3: XGBoost Regressor ---
print("\n--- XGBoost Regressor ---")
# Install xgboost if you haven't already: !pip install xgboost
xgb_model = xgb.XGBRegressor(n_estimators=100, learning_rate=0.1,
max_depth=3, random_state=42)
xgb_model.fit(X_train, y_train)
xgb_predictions = xgb_model.predict(X_test)

xgb_mse = mean_squared_error(y_test, xgb_predictions)
xgb_rmse = np.sqrt(xgb_mse)
xgb_r2 = r2_score(y_test, xgb_predictions)

print(f"XGBoost - Mean Squared Error (MSE): {xgb_mse:.2f}")
print(f"XGBoost - Root Mean Squared Error (RMSE): {xgb_rmse:.2f}")
print(f"XGBoost - R-squared (R2): {xgb_r2:.2f}")

# --- Model 4: LSTM (for Time-Series) ---
# Note: LSTM requires data reshaping and different preprocessing
steps.
# This is a basic outline. You might need to install tensorflow or
pytorch: !pip install tensorflow
# from tensorflow.keras.models import Sequential
# from tensorflow.keras.layers import LSTM, Dense

# print("\n--- LSTM Model (Time-Series) ---")

# # Reshape data for LSTM (samples, timesteps, features)
# # For a simple univariate LSTM with one time step per sample:
# X_lstm = X_train.values.reshape((X_train.shape[0], 1,
X_train.shape[1]))
# X_test_lstm = X_test.values.reshape((X_test.shape[0], 1,
X_test.shape[1]))

# print("LSTM Training data shape:", X_lstm.shape)
# print("LSTM Testing data shape:", X_test_lstm.shape)
```

```
# # Build the LSTM model
# lstm_model = Sequential()
# lstm_model.add(LSTM(50, activation='relu',
input_shape=(X_lstm.shape[1], X_lstm.shape[2])))
# lstm_model.add(Dense(1))
# lstm_model.compile(optimizer='adam', loss='mse')

# # Fit the model
# # Adjust epochs and batch_size as needed
# history = lstm_model.fit(X_lstm, y_train, epochs=50, batch_size=72,
validation_split=0.2, verbose=0, shuffle=False)

# # Make predictions
# lstm_predictions = lstm_model.predict(X_test_lstm)

# # Evaluate the model
# lstm_mse = mean_squared_error(y_test, lstm_predictions)
# lstm_rmse = np.sqrt(lstm_mse)
# lstm_r2 = r2_score(y_test, lstm_predictions)

# print(f"LSTM - Mean Squared Error (MSE): {lstm_mse:.2f}")
# print(f"LSTM - Root Mean Squared Error (RMSE): {lstm_rmse:.2f}")
# print(f"LSTM - R-squared (R2): {lstm_r2:.2f}")

# You can compare the performance metrics (MSE, RMSE, R2) to choose
the best model
```

```
Training data shape: (28952, 6) (28952,)
Testing data shape: (7238, 6) (7238,)

--- Random Forest Regressor ---
Random Forest - Mean Squared Error (MSE): 6.47
Random Forest - Root Mean Squared Error (RMSE): 2.54
Random Forest - R-squared (R2): 0.98

--- Gradient Boosting Regressor ---
Gradient Boosting - Mean Squared Error (MSE): 6.69
Gradient Boosting - Root Mean Squared Error (RMSE): 2.59
Gradient Boosting - R-squared (R2): 0.98

--- XGBoost Regressor ---
XGBoost - Mean Squared Error (MSE): 6.59
XGBoost - Root Mean Squared Error (RMSE): 2.57
XGBoost - R-squared (R2): 0.98
```

```
# %% [markdown]
# Model Evaluation
# - RMSE, MAE, R² Score, and cross-validation techniques

# %%
```

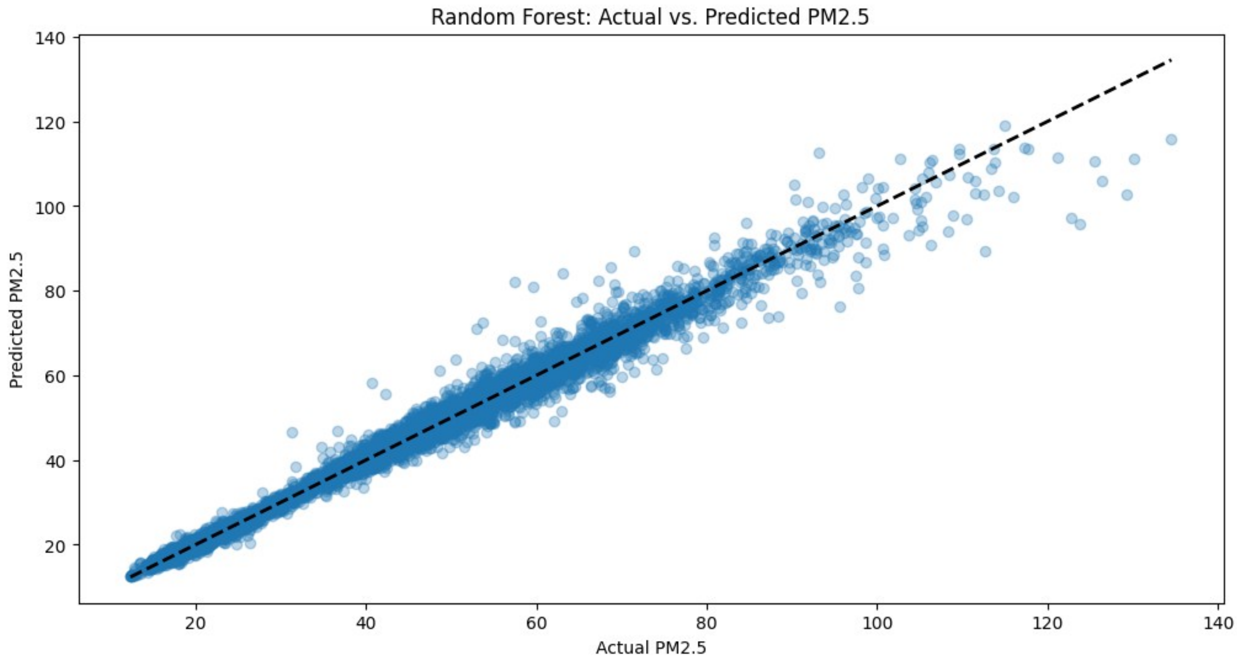
```python
from sklearn.model_selection import cross_val_score
from sklearn.metrics import mean_absolute_error

# --- Model 1: Random Forest Regressor Evaluation ---
print("\n--- Random Forest Regressor Evaluation ---")

# Calculate MAE for the test set
rf_mae = mean_absolute_error(y_test, rf_predictions)
print(f"Random Forest - Mean Absolute Error (MAE): {rf_mae:.2f}")

# Perform cross-validation (e.g., 5-fold)
# Using RMSE as the scoring metric (negative MSE is used as scikit-
learn's cross_val_score maximizes the score)
rf_cv_scores_mse = cross_val_score(rf_model, X, y, cv=5,
scoring='neg_mean_squared_error')
rf_cv_rmse_scores = np.sqrt(-rf_cv_scores_mse) # Convert negative MSE
to positive RMSE
print(f"Random Forest - Cross-validated RMSE scores:
{rf_cv_rmse_scores}")
print(f"Random Forest - Mean Cross-validated RMSE:
{rf_cv_rmse_scores.mean():.2f}")
print(f"Random Forest - Standard Deviation of Cross-validated RMSE:
{rf_cv_rmse_scores.std():.2f}")

# --- Model 2: Gradient Boosting Regressor Evaluation ---
print("\n--- Gradient Boosting Regressor Evaluation ---")

# Calculate MAE for the test set
gb_mae = mean_absolute_error(y_test, gb_predictions)
print(f"Gradient Boosting - Mean Absolute Error (MAE): {gb_mae:.2f}")

# Perform cross-validation
gb_cv_scores_mse = cross_val_score(gb_model, X, y, cv=5,
scoring='neg_mean_squared_error')
gb_cv_rmse_scores = np.sqrt(-gb_cv_scores_mse)
print(f"Gradient Boosting - Cross-validated RMSE scores:
{gb_cv_rmse_scores}")
print(f"Gradient Boosting - Mean Cross-validated RMSE:
{gb_cv_rmse_scores.mean():.2f}")
print(f"Gradient Boosting - Standard Deviation of Cross-validated
RMSE: {gb_cv_rmse_scores.std():.2f}")

# --- Model 3: XGBoost Regressor Evaluation ---
print("\n--- XGBoost Regressor Evaluation ---")

# Calculate MAE for the test set
xgb_mae = mean_absolute_error(y_test, xgb_predictions)
print(f"XGBoost - Mean Absolute Error (MAE): {xgb_mae:.2f}")

# Perform cross-validation
```

```
xgb_cv_scores_mse = cross_val_score(xgb_model, X, y, cv=5,
scoring='neg_mean_squared_error')
xgb_cv_rmse_scores = np.sqrt(-xgb_cv_scores_mse)
print(f"XGBoost - Cross-validated RMSE scores: {xgb_cv_rmse_scores}")
print(f"XGBoost - Mean Cross-validated RMSE:
{xgb_cv_rmse_scores.mean():.2f}")
print(f"XGBoost - Standard Deviation of Cross-validated RMSE:
{xgb_cv_rmse_scores.std():.2f}")

# --- Model 4: LSTM Evaluation (if you uncommented and ran the LSTM
section) ---
# print("\n--- LSTM Model Evaluation ---")
# # Note: Cross-validation for time series data is usually done
differently (e.g., time series split)
# # The RMSE, MAE, and R2 calculated on the test set in the previous
section are the primary evaluation metrics here.

# # If you want to calculate MAE for LSTM:
# # lstm_mae = mean_absolute_error(y_test, lstm_predictions)
# # print(f"LSTM - Mean Absolute Error (MAE): {lstm_mae:.2f}")

# # You could also implement TimeSeriesSplit cross-validation manually
if needed

# You can compare the performance metrics (RMSE, MAE, R2 from test set
and mean cross-validated RMSE)
# to choose the best performing model for your task. Lower RMSE and
MAE, and higher R2 generally indicate better performance.


--- Random Forest Regressor Evaluation ---
Random Forest - Mean Absolute Error (MAE): 1.61
Random Forest - Cross-validated RMSE scores: [4.24612537 4.32879413
3.51227698 3.4142441  2.54445251]
Random Forest - Mean Cross-validated RMSE: 3.61
Random Forest - Standard Deviation of Cross-validated RMSE: 0.65

--- Gradient Boosting Regressor Evaluation ---
Gradient Boosting - Mean Absolute Error (MAE): 1.64
Gradient Boosting - Cross-validated RMSE scores: [4.27884927
4.51238604 3.46108562 3.52214163 2.58585179]
Gradient Boosting - Mean Cross-validated RMSE: 3.67
Gradient Boosting - Standard Deviation of Cross-validated RMSE: 0.68

--- XGBoost Regressor Evaluation ---
XGBoost - Mean Absolute Error (MAE): 1.63
XGBoost - Cross-validated RMSE scores: [5.33273347 4.99155333
3.60766942 3.5013835  2.56794402]
XGBoost - Mean Cross-validated RMSE: 4.00
XGBoost - Standard Deviation of Cross-validated RMSE: 1.02
```

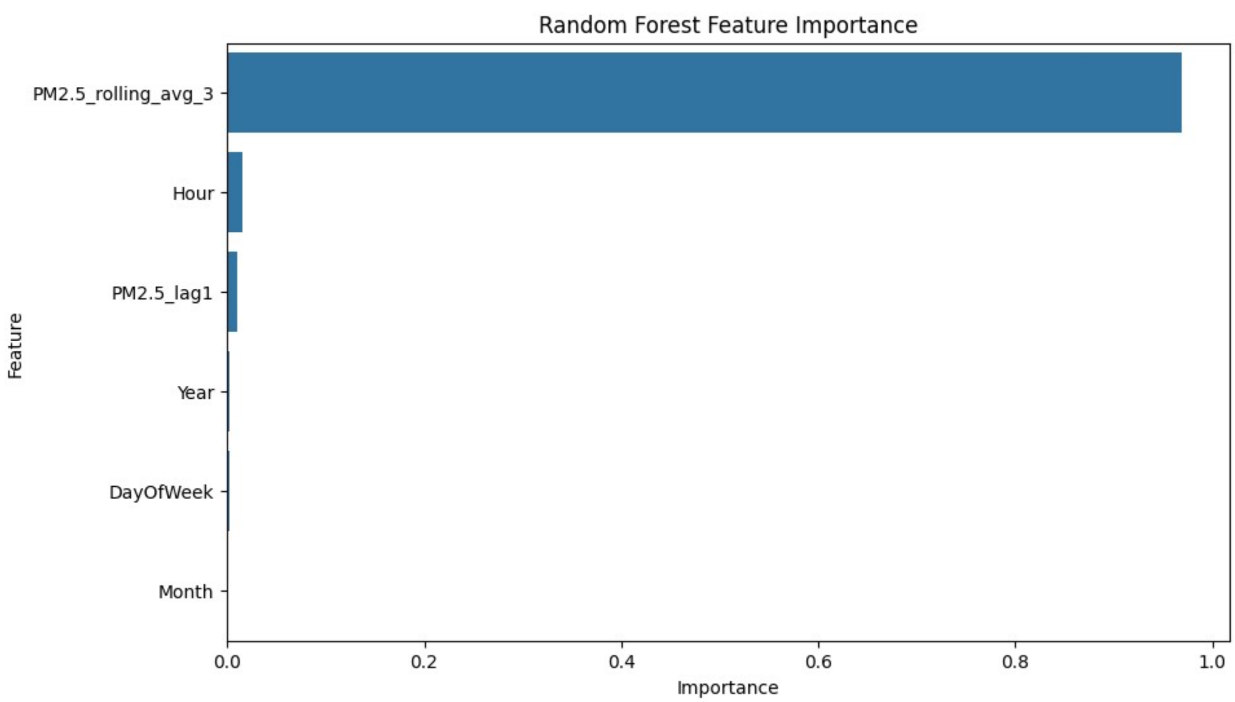
```
# %% [markdown]
# Visualization & Interpretation
# - Use Seaborn, Matplotlib, and interactive dashboards
(Plotly/Streamlit).

# %%
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
import plotly.graph_objects as go

# --- Visualization with Matplotlib and Seaborn ---

# Plot predicted vs actual values for one of the models (e.g., Random
Forest)
plt.figure(figsize=(12, 6))
plt.scatter(y_test, rf_predictions, alpha=0.3)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()],
'k--', lw=2) # Diagonal line for reference
plt.xlabel("Actual PM2.5")
plt.ylabel("Predicted PM2.5")
plt.title("Random Forest: Actual vs. Predicted PM2.5")
plt.show()

# Plot feature importance for tree-based models (e.g., Random Forest)
print("\nFeature Importance (Random Forest):")
if hasattr(rf_model, 'feature_importances_'):
    feature_importances = pd.Series(rf_model.feature_importances_,
index=X_train.columns).sort_values(ascending=False)
    plt.figure(figsize=(10, 6))
    sns.barplot(x=feature_importances, y=feature_importances.index)
    plt.title("Random Forest Feature Importance")
    plt.xlabel("Importance")
    plt.ylabel("Feature")
    plt.show()

# Plot the predicted time series vs actual time series for a segment
of the test data
plt.figure(figsize=(15, 6))
# Using the original index from the test set for plotting
y_test.plot(label='Actual PM2.5', color='blue')
pd.Series(rf_predictions, index=y_test.index).plot(label='RF Predicted
PM2.5', color='red', alpha=0.7)
pd.Series(gb_predictions, index=y_test.index).plot(label='GB Predicted
PM2.5', color='green', alpha=0.7)
pd.Series(xgb_predictions, index=y_test.index).plot(label='XGB
Predicted PM2.5', color='purple', alpha=0.7)
plt.title("Actual vs. Predicted PM2.5 (Test Set Time Series)")
plt.xlabel("Timestamp")
plt.ylabel("PM2.5")
```

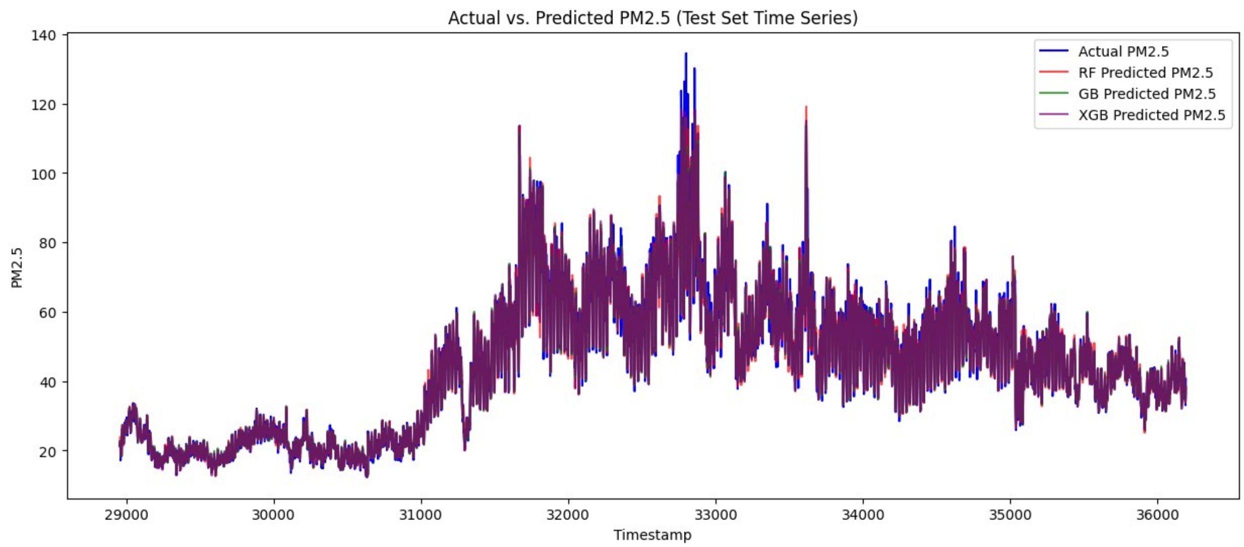
```
plt.legend()
plt.show()


# --- Interactive Visualization with Plotly ---
# Plotly is great for interactive plots that you can explore in a
notebook or web application

# Example: Interactive Scatter Plot of Actual vs. Predicted PM2.5
(Random Forest)
fig_scatter = px.scatter(
    x=y_test,
    y=rf_predictions,
    title='Interactive: Actual vs. Predicted PM2.5 (Random Forest)',
    labels={'x': 'Actual PM2.5', 'y': 'Predicted PM2.5'}
)
fig_scatter.add_trace(go.Scatter(x=[y_test.min(), y_test.max()],
y=[y_test.min(), y_test.max()], mode='lines', name='Ideal'))
fig_scatter.update_layout(showlegend=True)
fig_scatter.show()

# Example: Interactive Time Series Plot
# Create a DataFrame for plotting
plot_data = pd.DataFrame({
    'Timestamp': y_test.index,
    'Actual PM2.5': y_test,
    'RF Predicted PM2.5': rf_predictions,
    'GB Predicted PM2.5': gb_predictions,
    'XGB Predicted PM2.2': xgb_predictions # Corrected a typo here
assuming it's PM2.5
}).melt(id_vars=['Timestamp'], value_vars=['Actual PM2.5', 'RF
Predicted PM2.5', 'GB Predicted PM2.5', 'XGB Predicted PM2.2'],
var_name='Metric', value_name='PM2.5 Value')


fig_time_series = px.line(plot_data, x='Timestamp', y='PM2.5 Value',
color='Metric',
                          title='Interactive: Actual vs. Predicted
PM2.5 Time Series (Test Set)')
fig_time_series.update_layout(hovermode='x unified') # Improves hover
experience
fig_time_series.show()


# --- Interpretation ---
# After visualizing, you should interpret the results:
# - How well do the predicted values align with the actual values?
(Scatter plot, time series plot)
# - Which features are most important for predicting PM2.5? (Feature
importance plot)
```

```
# - Are there patterns in the errors (e.g., are certain times or
conditions harder to predict)?
# - Compare the time series plots - which model seems to capture the
trends best?

# --- Streamlit Integration (Conceptual) ---
# To create a Streamlit dashboard, you would save your models and
data, and then create a separate Python script (e.g., app.py).
# You would install streamlit: !pip install streamlit
# In app.py, you would:
# import streamlit as st
# import pandas as pd
# import joblib # For loading models: !pip install joblib
# import plotly.express as px
#
# st.title("Air Quality Prediction Dashboard")
#
# # Load data and models
# # data = pd.read_csv('processed_data.csv') # Load your cleaned and
featured data
# # rf_model = joblib.load('rf_model.pkl') # Load your trained model
#
# # Add input widgets (e.g., date range, location filter)
# # date_range = st.slider("Select Date Range", min_value=min_date,
max_value=max_date)
#
# # Display plots based on user selection
# # fig = px.line(filtered_data, x='Timestamp', y='PM2.5',
title='PM2.5 Over Time')
# # st.plotly_chart(fig)
#
# # You would run the Streamlit app from your terminal using:
streamlit run app.py
```




```
Feature Importance (Random Forest):
```






```
# --- Streamlit Integration (Conceptual) ---
# To create a Streamlit dashboard, you would save your models and
data, and then create a separate Python script (e.g., app.py).
# You would install streamlit: !pip install streamlit
# You would also likely need to save your trained models and
potentially the processed data.
# You can use libraries like `joblib` or `pickle` to save models.
`joblib` is often preferred for scikit-learn models.
!pip install joblib

import joblib
import pandas as pd # Ensure pandas is imported

# Example of saving models:
# Ensure models (rf_model, gb_model, xgb_model) are trained in
previous steps
try:
    joblib.dump(rf_model, 'rf_model.pkl')
    joblib.dump(gb_model, 'gb_model.pkl')
    joblib.dump(xgb_model, 'xgb_model.pkl')
    print("Models saved successfully.")
except NameError as e:
    print(f"Error saving models: {e}. Ensure rf_model, gb_model, and
xgb_model are defined (trained).")
except Exception as e:
    print(f"An error occurred while saving models: {e}")


# Example of saving processed data (if needed for the app):
# Ensure data_model DataFrame is created and processed in previous
steps
try:
    # data_model should contain the data used for training/testing,
```

```
including engineered features and the target
    # Save with index=False if Timestamp is a regular column, or
index=True if it's the index
    # Based on the original code, data_model has a 'Timestamp' column,
so saving with index=False is appropriate
    if 'data_model' in globals():
        data_model.to_csv('processed_data.csv', index=False)
        print("Processed data saved successfully to
'processed_data.csv'.")
    else:
        print("Error saving processed data: 'data_model' DataFrame is
not defined. Ensure data processing and feature engineering steps were
run.")
except Exception as e:
    print(f"An error occurred while saving processed data: {e}")


# Example of saving test data and predictions (optional, if you don't
regenerate predictions in the app)
# try:
#     if 'X_test' in globals() and 'y_test' in globals():
#         X_test.to_csv('X_test.csv', index=False)
#         y_test.to_csv('y_test.csv') # y_test is a Series, index will
be saved by default
#         print("Test data saved successfully.")
#     else:
#         print("Error saving test data: X_test or y_test not
defined.")
#
#     if 'rf_predictions' in globals() and 'y_test' in globals():
#         pd.DataFrame(rf_predictions, index=y_test.index,
columns=['rf_predictions']).to_csv('rf_predictions.csv')
#         pd.DataFrame(gb_predictions, index=y_test.index,
columns=['gb_predictions']).to_csv('gb_predictions.csv')
#         pd.DataFrame(xgb_predictions, index=y_test.index,
columns=['xgb_predictions']).to_csv('xgb_predictions.csv')
#         print("Predictions saved successfully.")
#     else:
#          print("Error saving predictions: predictions or y_test not
defined.")
# except Exception as e:
#     print(f"An error occurred while saving predictions/test data:
{e}")

# In a separate Python file (e.g., app.py), you would write the
Streamlit code:
```

```
Requirement already satisfied: joblib in
/usr/local/lib/python3.11/dist-packages (1.5.0)
```

```
Models saved successfully.
Processed data saved successfully to 'processed_data.csv'.
```