Air Pollution Analysis in Seoul, South Korea

Frantz Alexander

Case Study:

> Analyze and predict the air quality in the Gangnam downtown shopping district for the first of November 2019.

Libraries

In [1]:
```
import time

import warnings

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px

from sklearn.metrics import mean_absolute_error
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.arima.model import ARIMA

%matplotlib inline
```

```python
warnings.filterwarnings("ignore")
```

Prepare Dataset

Import Dataset

In [2]:
```python
def wrangle(filepath, resample_rule = "1H"):
    # Read data into DataFrame
    dataset = pd.read_csv(filepath)

    # Subset station in Gangnam
    station_mask = dataset["Station code"] == 123

    # Subset readings to pm2.5
    pm_mask = dataset["Item code"] == 9

    # subset
    dataset = dataset[station_mask & pm_mask]

    # Drop station code, item code, instument status
    dataset = (
        dataset.drop(
            columns = ["Station code", "Item code", "Instrument status"]
        )
    )

    # Rename Values to PM2.5 readings
    dataset = dataset.rename(columns = {"Average value": "PM2.5 Readings"})

    # Setting the index to the measurement date
    dataset = dataset.set_index("Measurement date")

    # Setting index to datetime
    dataset.index = pd.to_datetime(dataset.index)

    # Converting index to local time in Seoul
```

```python
    dataset.index = dataset.index.tz_localize("UTC").tz_convert("Asia/Seoul")


    # Remove outliers
    outlier_mask_high = dataset["PM2.5 Readings"] < 500

    outlier_mask_low = dataset["PM2.5 Readings"] > 0

    dataset = dataset[outlier_mask_high & outlier_mask_low]

    # Resample and forward-fill
    y = dataset["PM2.5 Readings"].resample(resample_rule).mean().fillna(method = "ffill")

    return y
```

In [3]:
```python
y = wrangle("Measurement_info.csv")
```

In [4]:
```python
y.info()
y.tail(30)
```

```
<class 'pandas.core.series.Series'>
DatetimeIndex: 26280 entries, 2017-01-01 09:00:00+09:00 to 2020-01-01 08:00:00+09:00
Freq: H
Series name: PM2.5 Readings
Non-Null Count  Dtype
--------------  -----
26280 non-null  float64
dtypes: float64(1)
memory usage: 410.6 KB
```

Out[4]:
```
Measurement date
2019-12-31 03:00:00+09:00    31.0
2019-12-31 04:00:00+09:00    23.0
2019-12-31 05:00:00+09:00    22.0
2019-12-31 06:00:00+09:00    16.0
2019-12-31 07:00:00+09:00    14.0
2019-12-31 08:00:00+09:00    14.0
2019-12-31 09:00:00+09:00    20.0
2019-12-31 10:00:00+09:00    18.0
2019-12-31 11:00:00+09:00    19.0
2019-12-31 12:00:00+09:00    23.0
2019-12-31 13:00:00+09:00    23.0
2019-12-31 14:00:00+09:00    18.0
2019-12-31 15:00:00+09:00    14.0
2019-12-31 16:00:00+09:00    12.0
2019-12-31 17:00:00+09:00    12.0
2019-12-31 18:00:00+09:00    10.0
2019-12-31 19:00:00+09:00     9.0
2019-12-31 20:00:00+09:00     7.0
2019-12-31 21:00:00+09:00    11.0
2019-12-31 22:00:00+09:00    12.0
2019-12-31 23:00:00+09:00     9.0
2020-01-01 00:00:00+09:00    13.0
2020-01-01 01:00:00+09:00    16.0
2020-01-01 02:00:00+09:00    14.0
2020-01-01 03:00:00+09:00    15.0
2020-01-01 04:00:00+09:00    15.0
2020-01-01 05:00:00+09:00    16.0
2020-01-01 06:00:00+09:00    14.0
2020-01-01 07:00:00+09:00    15.0
2020-01-01 08:00:00+09:00    13.0
Freq: H, Name: PM2.5 Readings, dtype: float64
```

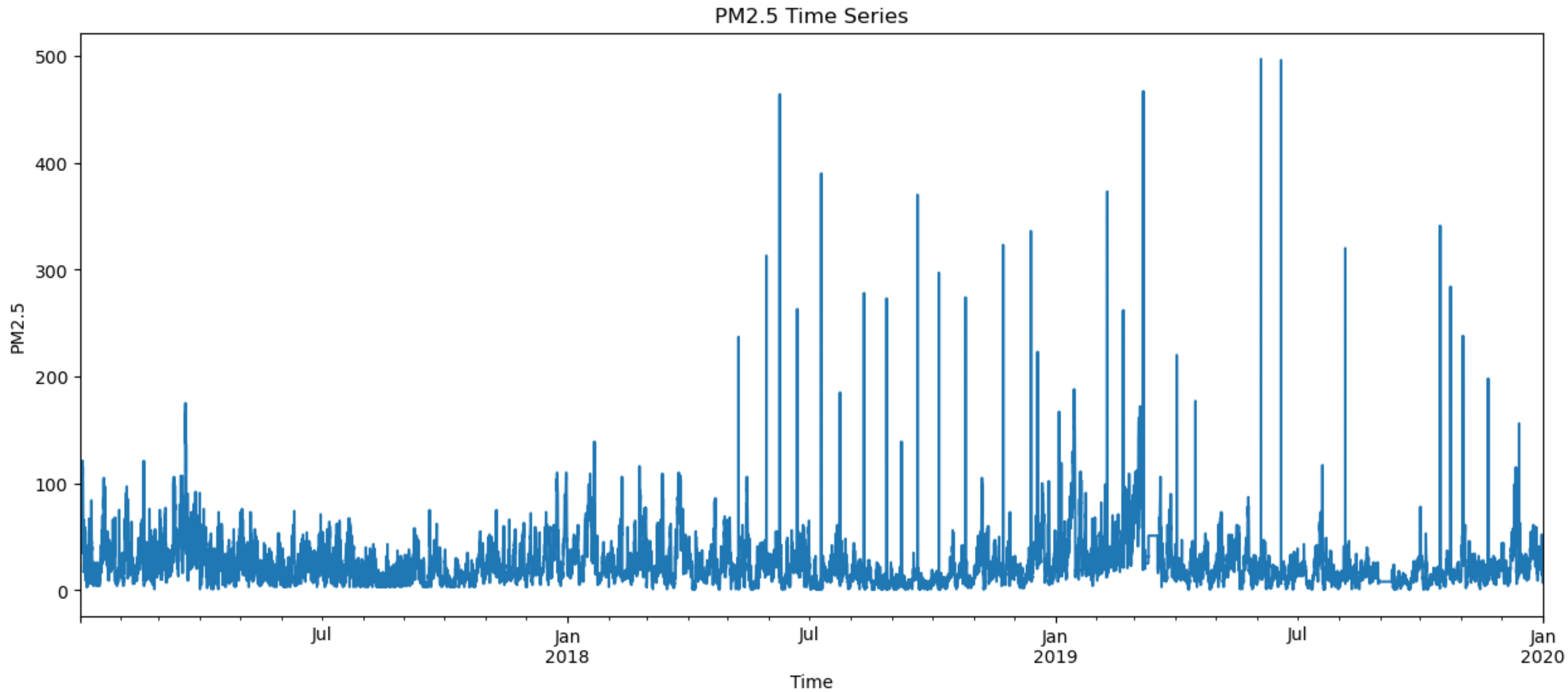
Explore Data

In [5]:
```
y.describe()
```

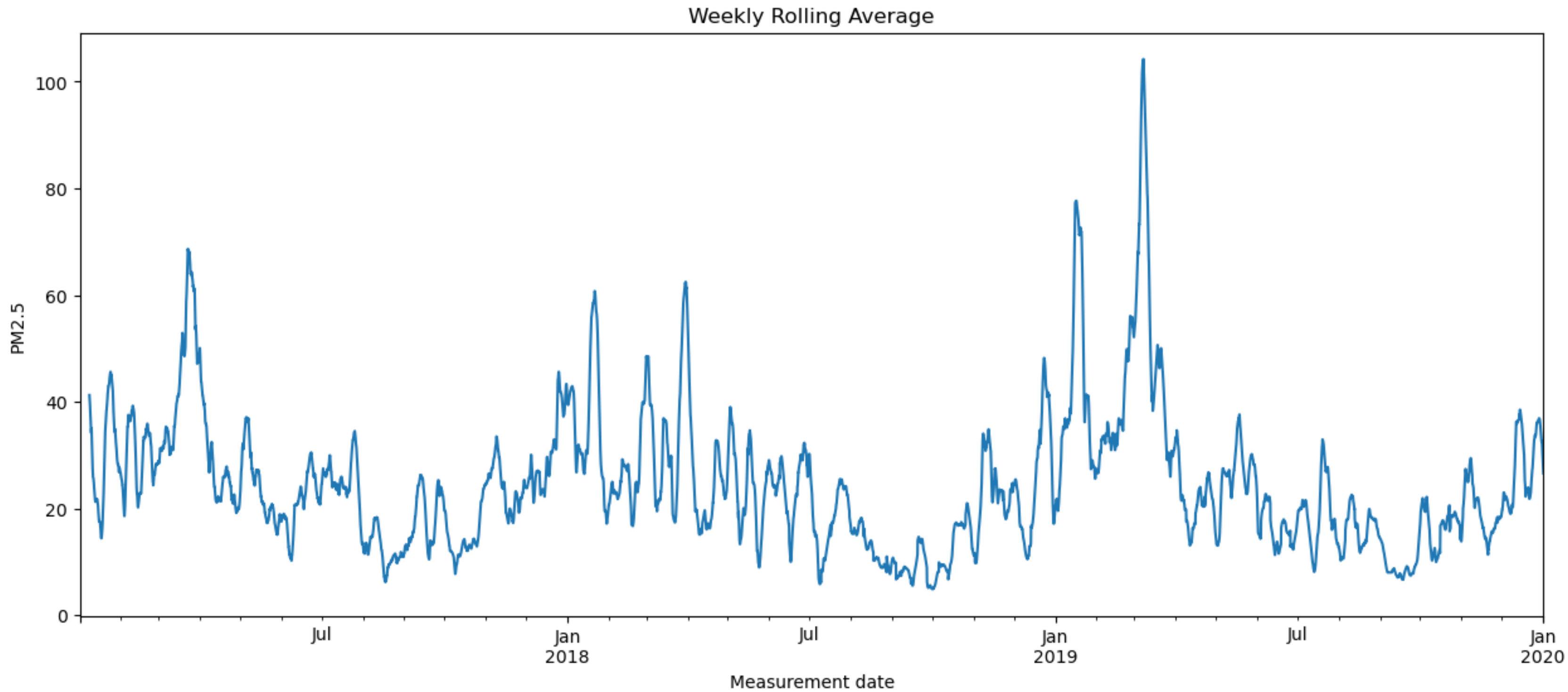
Out[5]:
```
count    26280.000000
mean        24.465715
std         22.249360
min          1.000000
25%         11.000000
50%         19.000000
75%         31.000000
max        497.000000
Name: PM2.5 Readings, dtype: float64
```

> We removed readings that were above 500 as they were highly likely to be measurement errors. Readings that were below 0 were also removed.

In [6]:
```
fig, ax = plt.subplots(figsize = (15, 6))
y.plot(
    xlabel = "Time",
    ylabel = "PM2.5",
    title = "PM2.5 Time Series",
    ax = ax
);
```



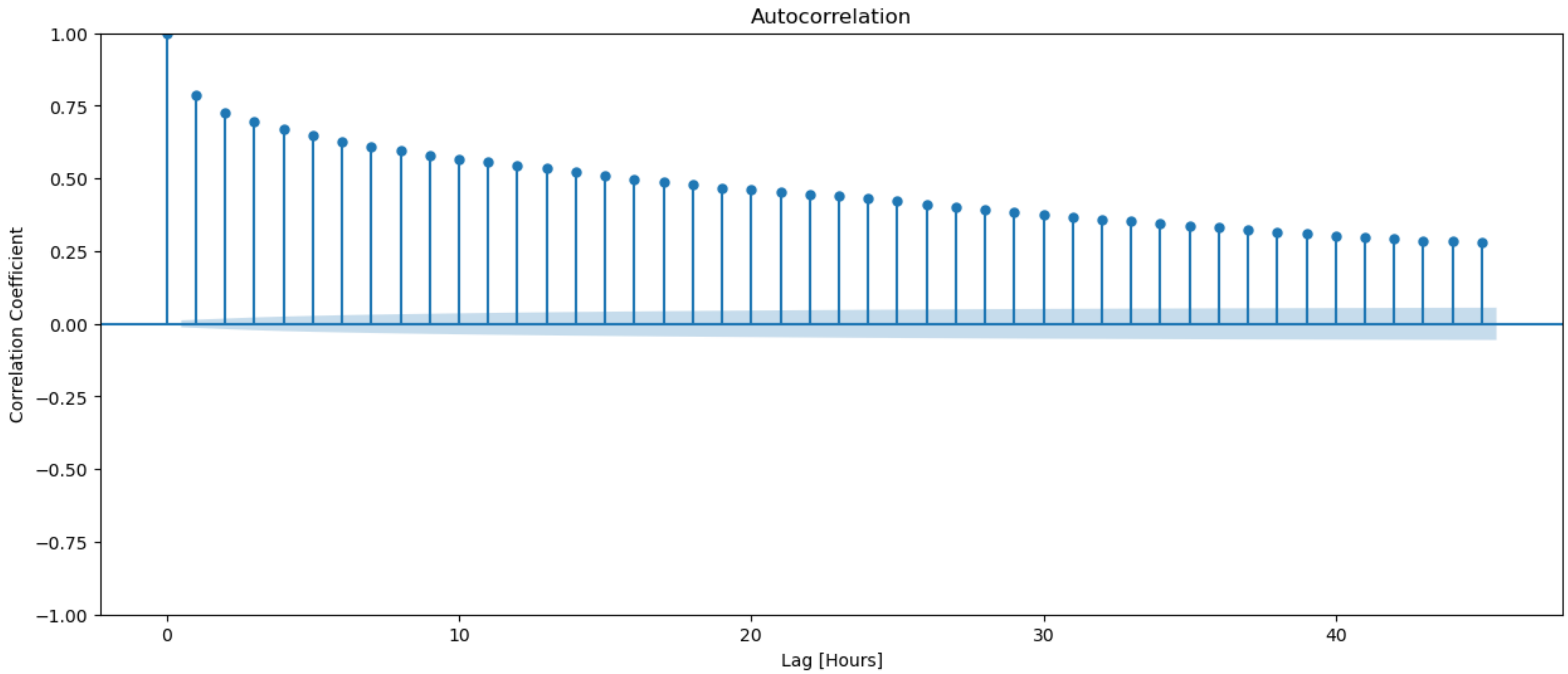

```
In [7]: fig, ax = plt.subplots(figsize = (15, 6))
        y.rolling(168).mean().plot(
            ax = ax,
            ylabel ="PM2.5",
            title = "Weekly Rolling Average"
        );
```



> Using a rolling average smooths out the data and aids in identifying general trends that would be important to model.

Create an ACF Plot for the data in y.

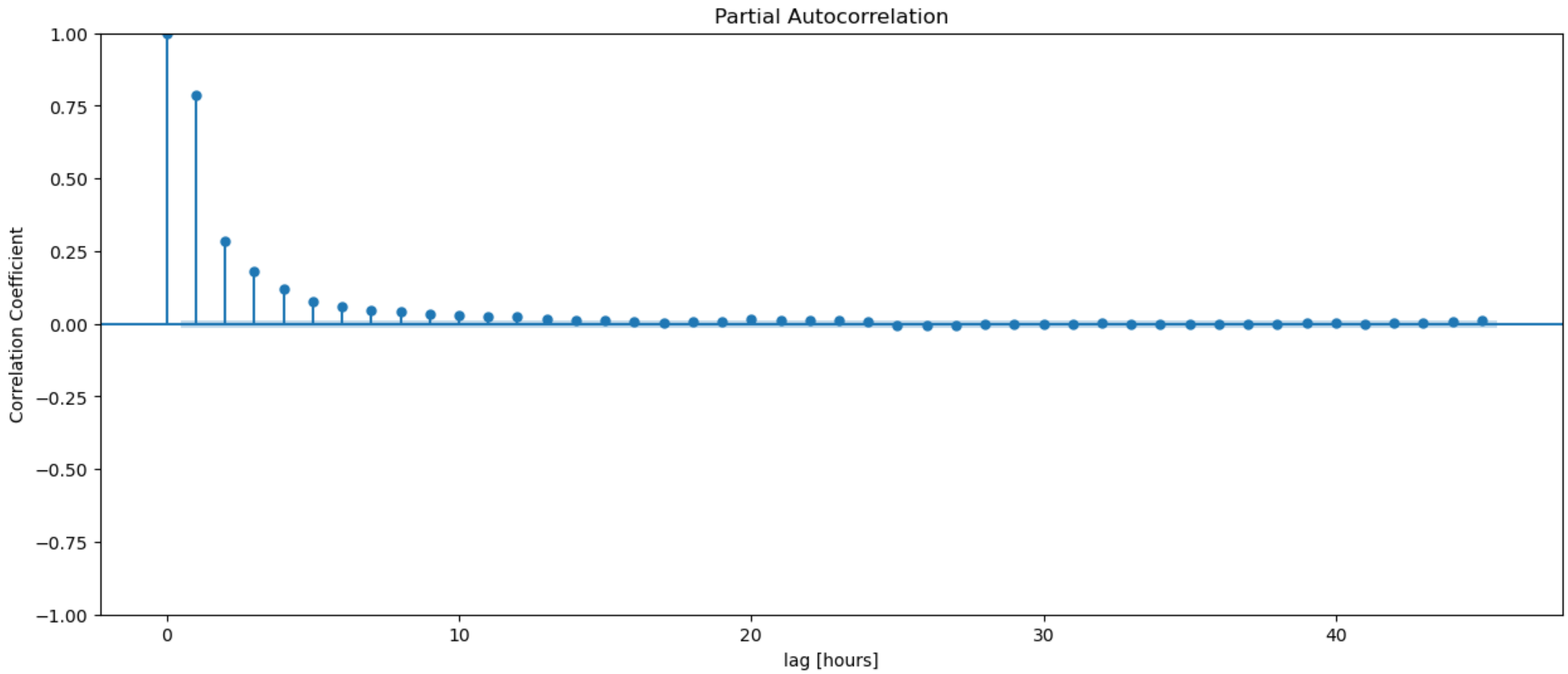
In [8]:
```
fig, ax = plt.subplots(figsize = (15, 6))
plot_acf(y, ax = ax)
plt.xlabel("Lag [Hours]")
plt.ylabel("Correlation Coefficient");
```



> The ACF plot illustrates how the Correlation Coefficient changes as the time lag increases.
>
> The blue band at the bottom of the graph shows that the measurements within it are statistically insignificant.
>
> That is because they are very close to a Correlation Coefficient of 0.
>
> We are only interested in measurements that are far above or below the band.
>
> We use the Partial AutoCorrelation Function plot(PACF) to determine how many lag terms we want in our model.

Create PACF Plot

In [9]:
```
fig, ax = plt.subplots(figsize = (15, 6))
plot_pacf(y, ax = ax)
plt.xlabel("lag [hours]")
plt.ylabel("Correlation Coefficient");
```



> Once you pull out the lag at 1 hour, the predictive power drastically drops as time passes.
> The predictive power continues to drop until a time lag of 12. Then again at lags 22, and 23 hours.
>
> At lag 26, the correlation coefficent becomes slightly negative until a lag of 45.
>
> Adding more time lags beyond 45 would not benefit our model.

Split Data

Training Set

In [10]:
```
y["2019-10-01":"2019-10-31"]
```

Out[10]:
```
Measurement date
2019-10-01 00:00:00+09:00    25.0
2019-10-01 01:00:00+09:00    22.0
2019-10-01 02:00:00+09:00    39.0
2019-10-01 03:00:00+09:00    57.0
2019-10-01 04:00:00+09:00    57.0
                             ...
2019-10-31 19:00:00+09:00    12.0
2019-10-31 20:00:00+09:00    12.0
2019-10-31 21:00:00+09:00    16.0
2019-10-31 22:00:00+09:00    24.0
2019-10-31 23:00:00+09:00    25.0
Freq: H, Name: PM2.5 Readings, Length: 744, dtype: float64
```

Test set

In [11]:
```
y["2019-11-01":"2019-11-30"]
```

Out[11]:
```
Measurement date
2019-11-01 00:00:00+09:00    30.0
2019-11-01 01:00:00+09:00    37.0
2019-11-01 02:00:00+09:00    37.0
2019-11-01 03:00:00+09:00    40.0
2019-11-01 04:00:00+09:00    40.0
                             ...
2019-11-30 19:00:00+09:00    28.0
2019-11-30 20:00:00+09:00    24.0
2019-11-30 21:00:00+09:00    25.0
2019-11-30 22:00:00+09:00    26.0
2019-11-30 23:00:00+09:00    26.0
Freq: H, Name: PM2.5 Readings, Length: 720, dtype: float64
```

Create Training Variable and Test Variable

In [12]:
```
y_train = y["2019-10-01":"2019-10-31"]
y_test = y["2019-11-01"]
```

In [13]:
```
print(len(y_train))
print(len(y_test))
```

```
744
24
```

Model Building

Baseline

Calculate the baseline Mean Absolute Error

In [14]:
```
y_train_mean = y_train.mean()
y_pred_baseline = [y_train_mean]*len(y_train)
mae_baseline = mean_absolute_error(y_train, y_pred_baseline)

print("Mean P2 Reading:", round(y_train_mean, 2))
print("Baseline MAE", round(mae_baseline, 2))
```

```
Mean P2 Reading: 15.98
Baseline MAE 8.08
```

Iterate

Hyper Parameters

In [15]:
```
p_params = range(0, 46, 5)
q_params = range(0, 4, 1)
```

In [16]:
```
list(p_params)
```

Out[16]:
```
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

In [17]:
```
list(q_params)
```

Out[17]:
```
[0, 1, 2, 3]
```

In [18]:
```
# Create an empty dictionary for MAE values
mae_grid = {}
for p in p_params:
    # Create new key in dict with empty list
    mae_grid[p] = []
    for q in q_params:
        order = (p, 0, q)
        # Start Timing
        start_time = time.time()
        model = ARIMA(y_train, order = order).fit()
        # Calculate the Elapsed Time
        elapsed_time = round(time.time() - start_time, 2)
        print(f"Trained ARIMA model {order} in {elapsed_time} seconds.")
        # Generate in-sample predictions
        y_pred = model.predict()
        # Calculate training MAE
        mae = mean_absolute_error(y_train, y_pred)
        print(mae)
        # Add MAE to Dictionary
        mae_grid[p].append(mae)
```

```
Trained ARIMA model (0, 0, 0) in 0.16 seconds.
8.081945672169077
Trained ARIMA model (0, 0, 1) in 0.12 seconds.
5.469754371426196
Trained ARIMA model (0, 0, 2) in 0.17 seconds.
4.545075159100064
Trained ARIMA model (0, 0, 3) in 0.27 seconds.
4.4256982832994
Trained ARIMA model (5, 0, 0) in 0.18 seconds.
4.226508439164455
Trained ARIMA model (5, 0, 1) in 0.71 seconds.
4.231962358179267
Trained ARIMA model (5, 0, 2) in 1.33 seconds.
4.175676546497824
Trained ARIMA model (5, 0, 3) in 1.4 seconds.
4.145377676127871
Trained ARIMA model (10, 0, 0) in 0.67 seconds.
4.165848676340261
Trained ARIMA model (10, 0, 1) in 2.52 seconds.
4.186803316668656
Trained ARIMA model (10, 0, 2) in 2.2 seconds.
4.142727910159227
Trained ARIMA model (10, 0, 3) in 1.42 seconds.
4.137354466464098
Trained ARIMA model (15, 0, 0) in 1.15 seconds.
4.1406348006994005
Trained ARIMA model (15, 0, 1) in 1.02 seconds.
4.140808825169648
Trained ARIMA model (15, 0, 2) in 1.23 seconds.
4.141042480804938
Trained ARIMA model (15, 0, 3) in 2.0 seconds.
4.155261991090604
Trained ARIMA model (20, 0, 0) in 1.76 seconds.
4.137845619147744
Trained ARIMA model (20, 0, 1) in 1.58 seconds.
4.139377925758959
Trained ARIMA model (20, 0, 2) in 1.85 seconds.
4.139552909198084
Trained ARIMA model (20, 0, 3) in 1.65 seconds.
4.139419369988037
Trained ARIMA model (25, 0, 0) in 3.1 seconds.
4.1289408575348
```

```
Trained ARIMA model (25, 0, 1) in 2.38 seconds.
4.12886561602995
Trained ARIMA model (25, 0, 2) in 5.38 seconds.
4.162416165626995
Trained ARIMA model (25, 0, 3) in 2.85 seconds.
4.1295606686006225
Trained ARIMA model (30, 0, 0) in 5.23 seconds.
4.127449487856752
Trained ARIMA model (30, 0, 1) in 3.49 seconds.
4.127925492781879
Trained ARIMA model (30, 0, 2) in 4.27 seconds.
4.127990474157672
Trained ARIMA model (30, 0, 3) in 10.65 seconds.
4.1524915306381
Trained ARIMA model (35, 0, 0) in 7.59 seconds.
4.134496001416412
Trained ARIMA model (35, 0, 1) in 6.46 seconds.
4.135085438395935
Trained ARIMA model (35, 0, 2) in 7.0 seconds.
4.135119956234608
Trained ARIMA model (35, 0, 3) in 21.15 seconds.
4.1604595502463235
Trained ARIMA model (40, 0, 0) in 10.46 seconds.
4.139770108980159
Trained ARIMA model (40, 0, 1) in 7.36 seconds.
4.13874002223658
Trained ARIMA model (40, 0, 2) in 9.66 seconds.
4.138679637108064
Trained ARIMA model (40, 0, 3) in 30.43 seconds.
4.187551368060375
Trained ARIMA model (45, 0, 0) in 16.05 seconds.
4.129178974160771
Trained ARIMA model (45, 0, 1) in 11.87 seconds.
4.128304612996378
Trained ARIMA model (45, 0, 2) in 40.63 seconds.
4.1580243638611
Trained ARIMA model (45, 0, 3) in 40.4 seconds.
4.122860154601666
```

In [19]:

```
print(mae_grid)
```

```
{0: [8.081945672169077, 5.469754371426196, 4.545075159100064, 4.4256982832994], 5: [4.226508439164455, 4.23196235817
9267, 4.175676546497824, 4.145377676127871], 10: [4.165848676340261, 4.186803316668656, 4.142727910159227, 4.1373544
66464098], 15: [4.1406348006994005, 4.140808825169648, 4.141042480804938, 4.155261991090604], 20: [4.13784561914774
4, 4.139377925758959, 4.139552909198084, 4.139419369988037], 25: [4.1289408575348, 4.12886561602995, 4.1624161656269
95, 4.1295606686006225], 30: [4.127449487856752, 4.127925492781879, 4.127990474157672, 4.1524915306381], 35: [4.1344
96001416412, 4.135085438395935, 4.135119956234608, 4.1604595502463235], 40: [4.139770108980159, 4.13874002223658, 4.
138679637108064, 4.187551368060375], 45: [4.129178974160771, 4.128304612996378, 4.1580243638611, 4.122860154601666]}
```


In [20]:

```
def gold_min(xs):
    m = xs.to_numpy().min()
    color = {True: "background-color: #c78f2e", False: ""}
    is_min = (xs == m).replace(color)
    return is_min

mae_df = pd.DataFrame(mae_grid)
mae_df.round(4).style.apply(gold_min, axis = None)
```

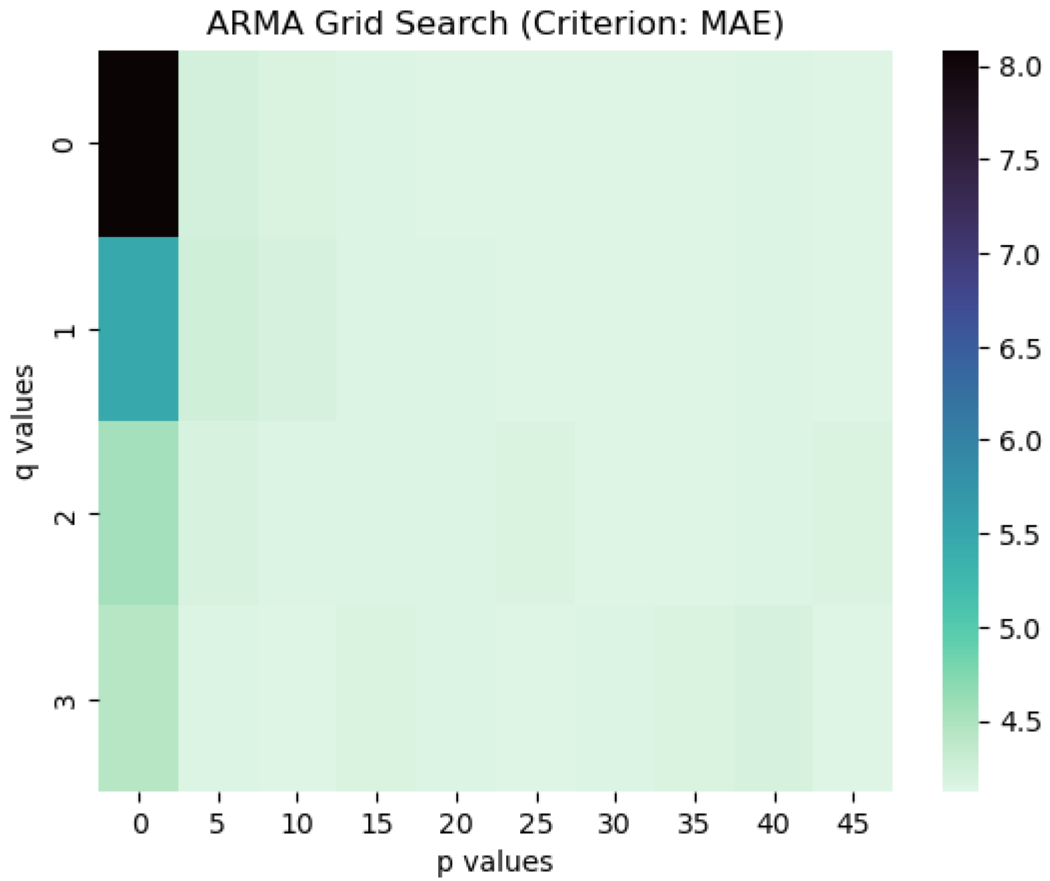
Out[20]:

	0	5	10	15	20	25	30	35	40	45
0	8.081900	4.226500	4.165800	4.140600	4.137800	4.128900	4.127400	4.134500	4.139800	4.129200
1	5.469800	4.232000	4.186800	4.140800	4.139400	4.128900	4.127900	4.135100	4.138700	4.128300
2	4.545100	4.175700	4.142700	4.141000	4.139600	4.162400	4.128000	4.135100	4.138700	4.158000
3	4.425700	4.145400	4.137400	4.155300	4.139400	4.129600	4.152500	4.160500	4.187600	4.122900

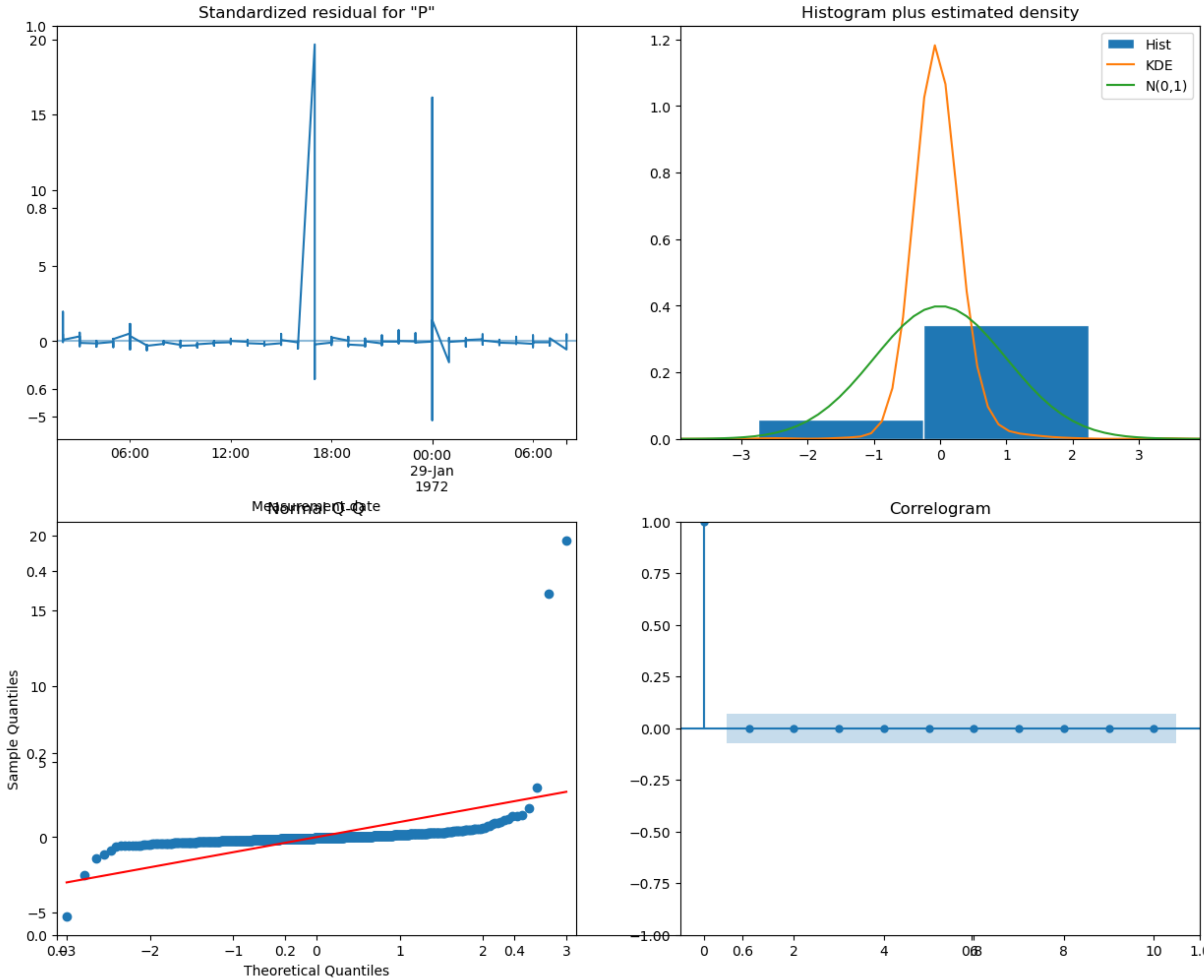
> The optimal the values for p and q are the parameters that best balance model performance with computational time.
>
> In this case, where p = 30 and q = 1.
>
> It is important to note that the parameters that produce the best model performance are when p = 30 and q = 0.

In [21]:

```
sns.heatmap(
    data = mae_df,
    cmap = "mako_r"
)
plt.xlabel("p values")
plt.ylabel("q values")
plt.title("ARMA Grid Search (Criterion: MAE)");
```



In [22]:
```
fig, ax = plt.subplots(figsize = (15, 12))
model.plot_diagnostics(fig = fig);
```



Standardized residual for "P"
Measurement date
Histogram plus estimated density
Hist
KDE
N(0,1)
Normal Q-Q
Sample Quantiles
Theoretical Quantiles
Correlogram

Evaluate

Walk-Forward Validation

In [23]:
```
y_pred_wfv = pd.Series().astype(str)
history = y_train.copy()
for i in range(len(y_test)):
    model = ARIMA(history, order = (30, 0, 1)).fit()
    next_pred = model.forecast()
    y_pred_wfv = pd.concat([y_pred_wfv, next_pred])
    history = pd.concat([history, y_test[next_pred.index]])
```

In [24]:
```
test_mae = mean_absolute_error(y_test, y_pred_wfv)
print("Test Mae (Walk-Forward Validation):", round(test_mae, 2))
```

```
Test Mae (Walk-Forward Validation): 8.76
```

Communicate

In [25]:
```
df_predictions = pd.DataFrame({"y_test" : y_test, "y_pred_wfv": y_pred_wfv})
fig = px.line(df_predictions, labels = {"value": "PM2.5"})
fig.show()
```

> The Baseline Mae was 8.08.
>
> The mae from the Test Data with walk-forward validation was at 8.76.
>
> November 1st experienced significantly higher air pollution than what was represented in the training data from the month of October.
>
> Thus, the model underestimated the pollution increase.