The dataset contains the following information:

Dataset curtains 9 columns where data columns are converted into indexes so that we can track unique variables or features . dataset of air pollution in year 2010 to 2015

Laibreaies need to be installed

In [1]:
```
%load_ext autoreload
%autoreload 2
%matplotlib inline
%config InlineBackend.figure_format='retina'

from __future__ import absolute_import, division, print_function

import sys
import os

import pandas as pd
import numpy as np

# # Remote Data Access
# import pandas_datareader.data as web
# import datetime
# # reference: https://pandas-datareader.readthedocs.io/en/latest/remote_data.html

# TSA from Statsmodels
import statsmodels.api as sm
import statsmodels.formula.api as smf
import statsmodels.tsa.api as smt

# Display and Plotting
import matplotlib.pylab as plt
import seaborn as sns

pd.set_option('display.float_format', lambda x: '%.5f' % x) # pandas
np.set_printoptions(precision=5, suppress=True) # numpy

pd.set_option('display.max_columns', 100)
pd.set_option('display.max_rows', 100)
from pylab import rcParams
```

```
# seaborn plotting style
sns.set(style='ticks', context='poster')
```

In [2]:
```
import tensorflow as tf
from pylab import rcParams
seed = 42
tf.random.set_seed(seed)
np.random.seed(seed)
plt.style.use('bmh')
plt.rcParams['axes.labelsize'] = 14
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12
plt.rcParams['text.color'] = 'k'
print(tf.__version__)
```

```
2.10.0
```

In [3]:
```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa import api as smt
from statsmodels.tsa.ar_model import AR
from statsmodels.tsa.arima_model import ARIMA, ARMA
from statsmodels.tsa.holtwinters import ExponentialSmoothing, SimpleExpSmoothing
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.tsa.stattools import adfuller
from tqdm import tqdm
```

In [4]:
```
from sklearn import linear_model, svm
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import LinearRegression
from sklearn.metrics import make_scorer, mean_squared_error
from sklearn.model_selection import GridSearchCV, cross_val_score
from sklearn.neighbors import KNeighborsRegressor
from sklearn.preprocessing import StandardScaler
```

In [5]:
```
import warnings
warnings.filterwarnings("ignore")
```

The dataset has been loaded and a quick preview has been provided

In [6]:
```
#Read the data
air_poll = pd.read_csv('C:/Users/vimala/Downloads/air_pollutionta.csv',parse_dates=['date'])
```

```
air_poll.set_index('date', inplace=True)
air_poll.head()
```

Out[6]:

date	pollution_today	dew	temp	press	wnd_spd	snow	rain	pollution_yesterday	Unnamed: 9	Unnamed: 10	Unnamed: 11
2010-01-02	145.95833	-8.50000	-5.12500	1024.75000	24.86000	0.70833	0.00000	10.04167	NaN	NaN	NaN
2010-01-03	78.83333	-10.12500	-8.54167	1022.79167	70.93792	14.16667	0.00000	145.95833	NaN	NaN	NaN
2010-01-04	31.33333	-20.87500	-11.50000	1029.29167	111.16083	0.00000	0.00000	78.83333	NaN	NaN	NaN
2010-01-05	42.45833	-24.58333	-14.45833	1033.62500	56.92000	0.00000	0.00000	31.33333	NaN	NaN	NaN
2010-01-06	56.41667	-23.70833	-12.54167	1033.75000	18.51167	0.00000	0.00000	42.45833	NaN	NaN	NaN

Droping the unwanted columns

In [7]:
```
air_poll=air_poll.drop(['Unnamed: 9','Unnamed: 10','Unnamed: 11'], axis=1)
```

this is the main data we will being working with

In [8]:
```
air_poll.head()
```

Out[8]:

date	pollution_today	dew	temp	press	wnd_spd	snow	rain	pollution_yesterday
2010-01-02	145.95833	-8.50000	-5.12500	1024.75000	24.86000	0.70833	0.00000	10.04167
2010-01-03	78.83333	-10.12500	-8.54167	1022.79167	70.93792	14.16667	0.00000	145.95833
2010-01-04	31.33333	-20.87500	-11.50000	1029.29167	111.16083	0.00000	0.00000	78.83333
2010-01-05	42.45833	-24.58333	-14.45833	1033.62500	56.92000	0.00000	0.00000	31.33333
2010-01-06	56.41667	-23.70833	-12.54167	1033.75000	18.51167	0.00000	0.00000	42.45833

In [9]:
```
air_poll.tail()
```

Out[9]:

date	pollution_today	dew	temp	press	wnd_spd	snow	rain	pollution_yesterday
2014-12-27	238.66667	-9.66667	-1.79167	1027.83333	9.27833	0.00000	0.00000	170.25000
2014-12-28	197.37500	-10.79167	1.58333	1019.95833	10.94875	0.00000	0.00000	238.66667
2014-12-29	159.00000	-12.33333	0.75000	1013.75000	8.00000	0.00000	0.00000	197.37500
2014-12-30	46.08333	-13.91667	1.87500	1019.12500	9.77833	0.00000	0.00000	159.00000
2014-12-31	10.04167	-21.79167	-1.91667	1032.12500	167.45833	0.00000	0.00000	46.08333

summary of the data set

In [10]:
```
air_poll.describe()
```

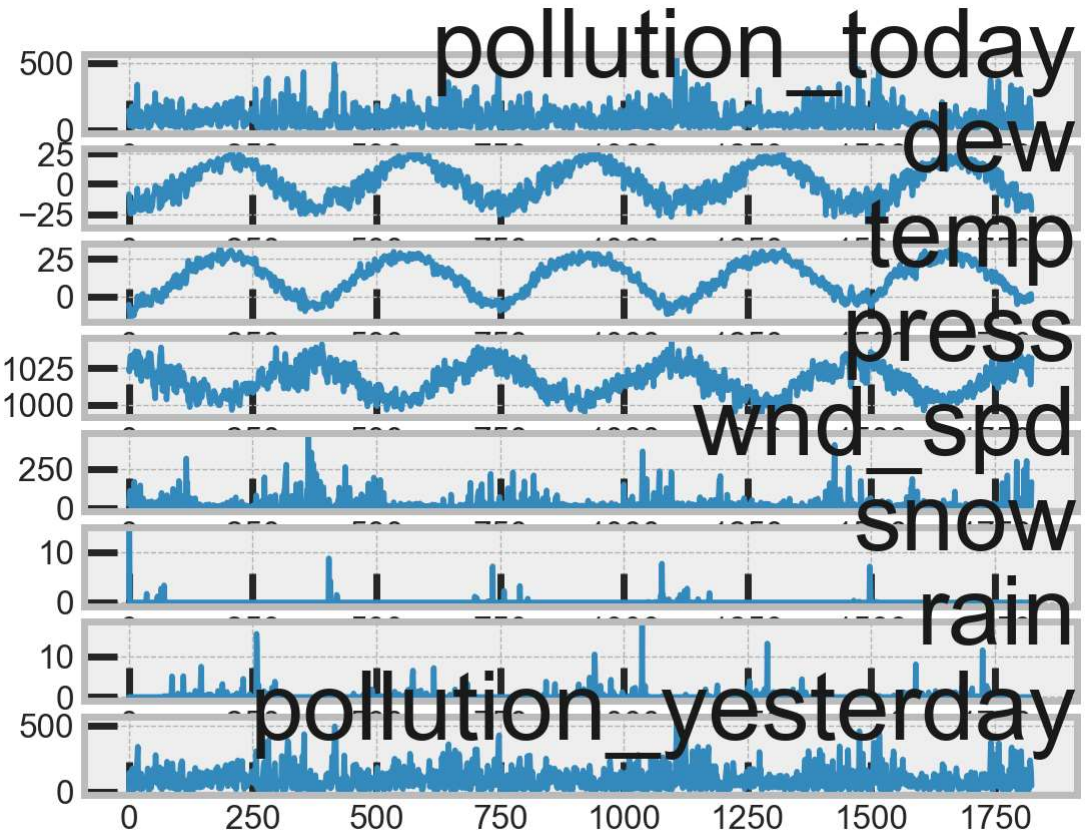
Out[10]:

	pollution_today	dew	temp	press	wnd_spd	snow	rain	pollution_yesterday
count	1825.00000	1825.00000	1825.00000	1825.00000	1825.00000	1825.00000	1825.00000	1825.00000
mean	98.24508	1.82852	12.45904	1016.44731	23.89431	0.05276	0.19502	98.24508
std	76.80770	14.16351	11.55300	10.07605	41.37316	0.54607	0.99392	76.80770
min	3.16667	-33.33333	-14.45833	994.04167	1.41250	0.00000	0.00000	3.16667
25%	42.33333	-10.08333	1.54167	1007.91667	5.90417	0.00000	0.00000	42.33333
50%	79.16667	2.04167	13.91667	1016.20833	10.95375	0.00000	0.00000	79.16667
75%	131.16667	15.08333	23.16667	1024.54167	22.23500	0.00000	0.00000	131.16667
max	541.89583	26.20833	32.87500	1043.45833	463.18792	14.16667	17.58333	541.89583

Let's examine each feature value

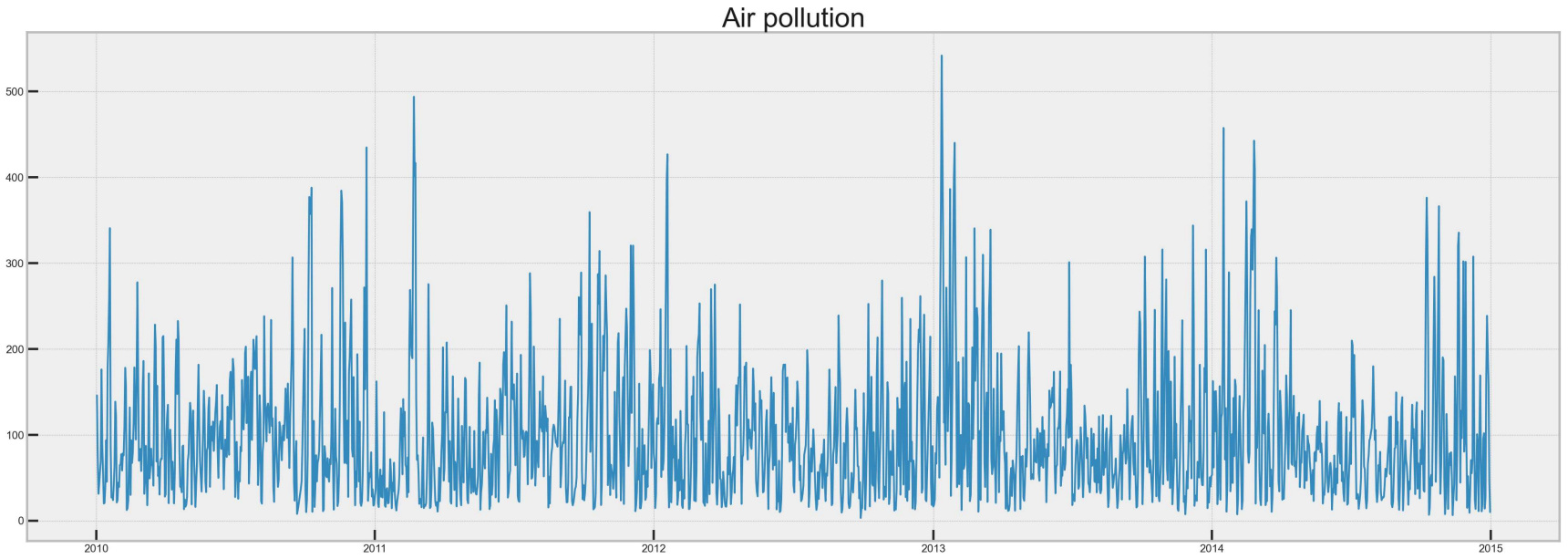
In [11]:

```
values = air_poll.values
groups = [0, 1, 2, 3, 4, 5, 6, 7]
i = 1
# plot each column
for group in groups:
    plt.subplot(len(groups), 1, i)
    plt.plot(values[:, group])
    plt.title(air_poll.columns[group], y=0.5, loc='right')
    i += 1


# f = plt.figure()
# f.set_figwidth(100)
# f.set_figheight(10)
plt.show()
```



In [12]:
```
plt.figure(num=None, figsize=(30, 10), dpi=80, facecolor='w', edgecolor='k')
plt.title('Air pollution', fontsize=30)

plt.plot(air_poll.pollution_today)
# plt.savefig("results/pollution.png")
```

Out[12]:
```
[<matplotlib.lines.Line2D at 0x18d63291760>]
```

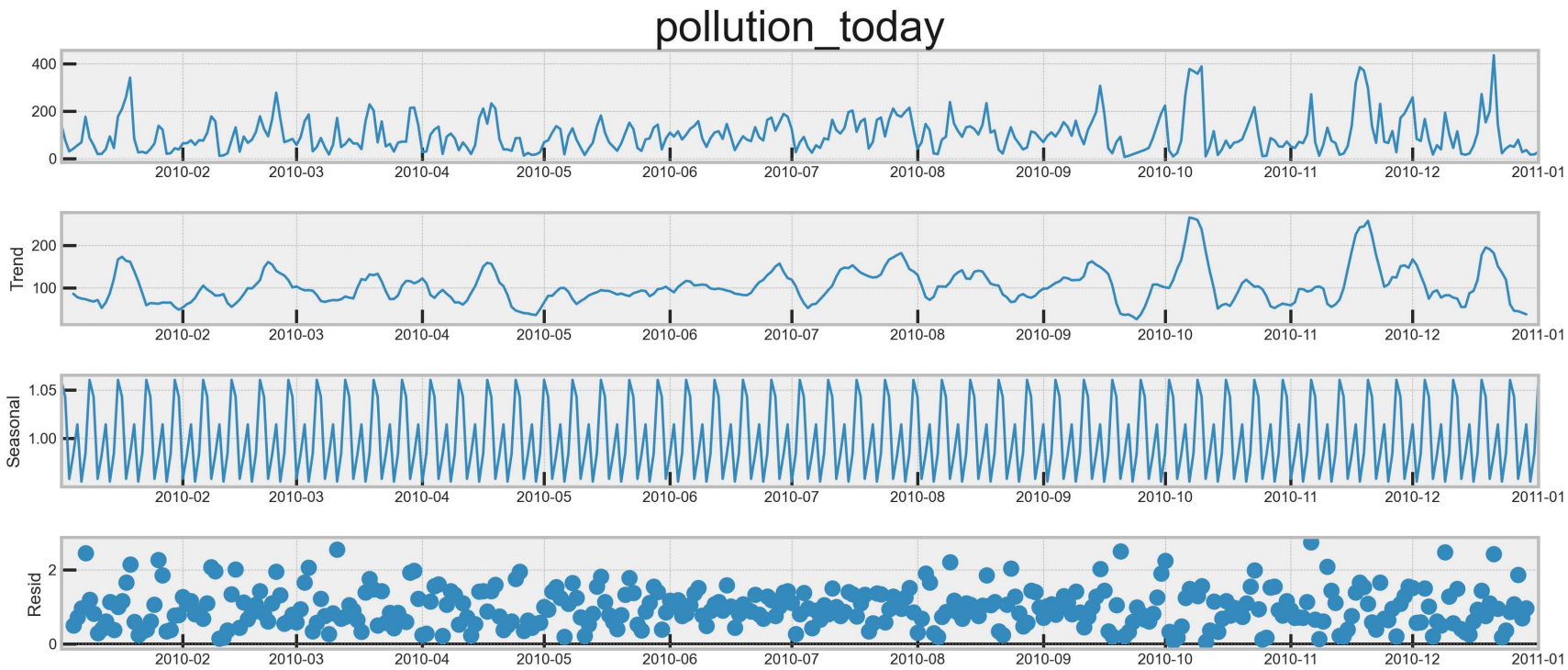


Automatic time series decomposition

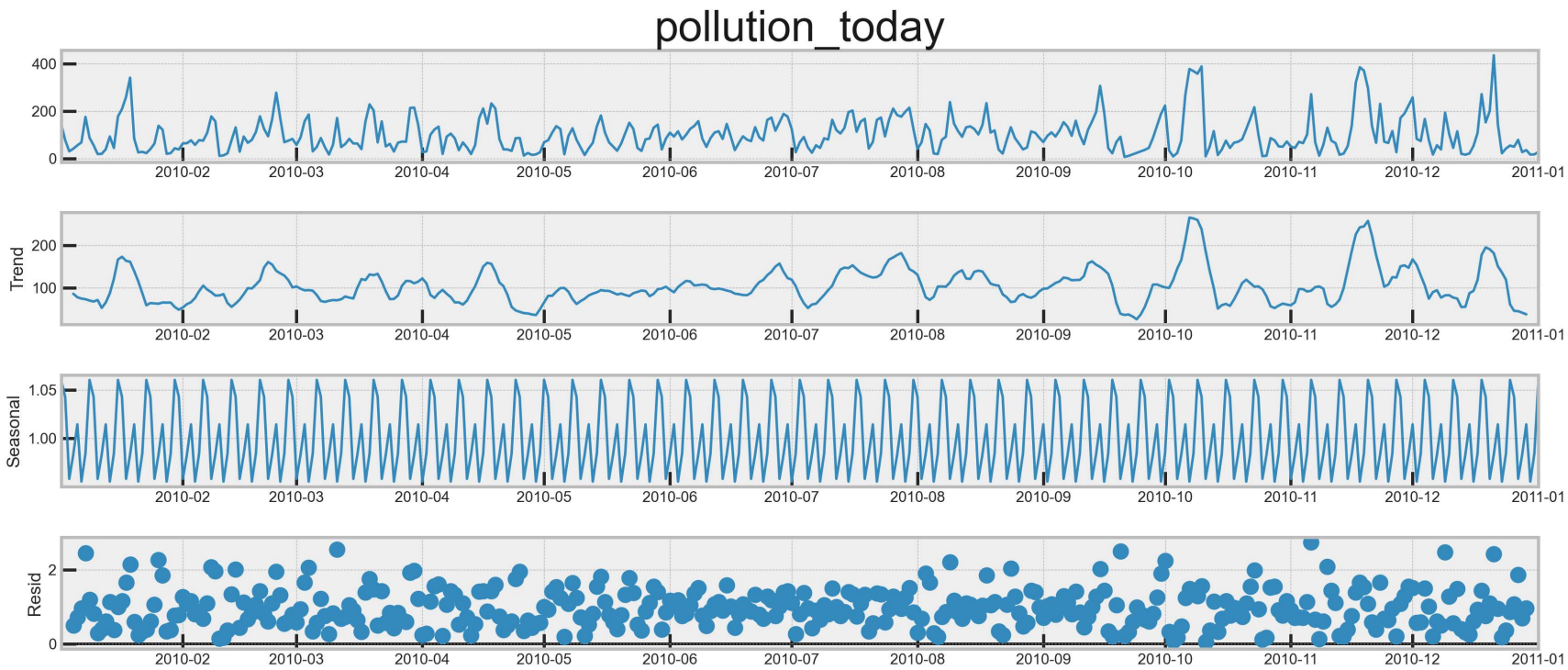
- before the decompositin lets know about types
  - Additive Model
  - Multiplicative model
- we can use an in bult Laibreaies ie., seasonal_decompose

In [13]:
```
rcParams['figure.figsize'] = 18, 8
plt.figure(num=None, figsize=(50, 20), dpi=80, facecolor='w', edgecolor='k')
series = air_poll.pollution_today[:365]
result = seasonal_decompose(series, model='multiplicative')
result.plot()
```

Out[13]:



<Figure size 4000x1600 with 0 Axes>



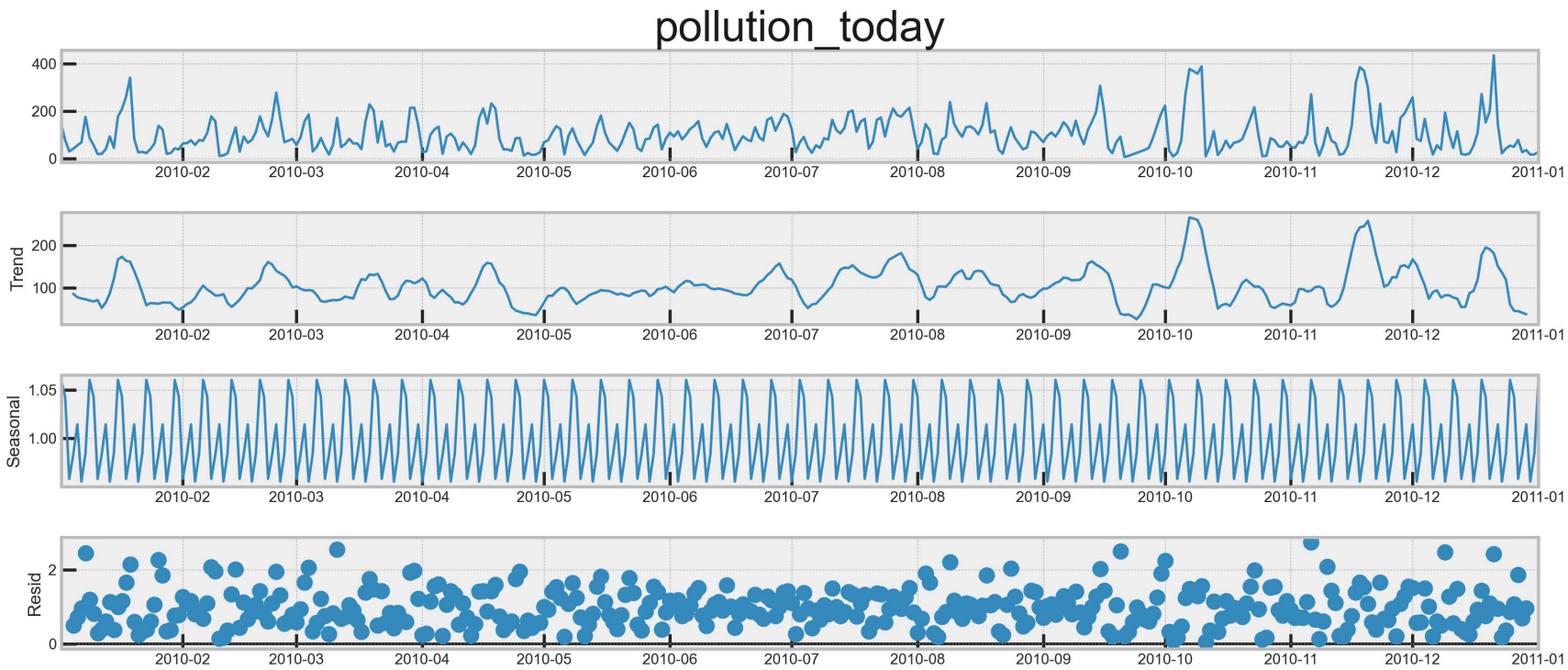
Level

- In simple terms, level is what we get after removing trend, seasonality, and random noise from the series. In our models, we will try to predict the true values from the series itself. The more our time series are composed of levels rather than trends, seasonality, and noise, the more benefit our models will receive. As well as models that handle seasonality and trend (non-stationary series), we present future work.

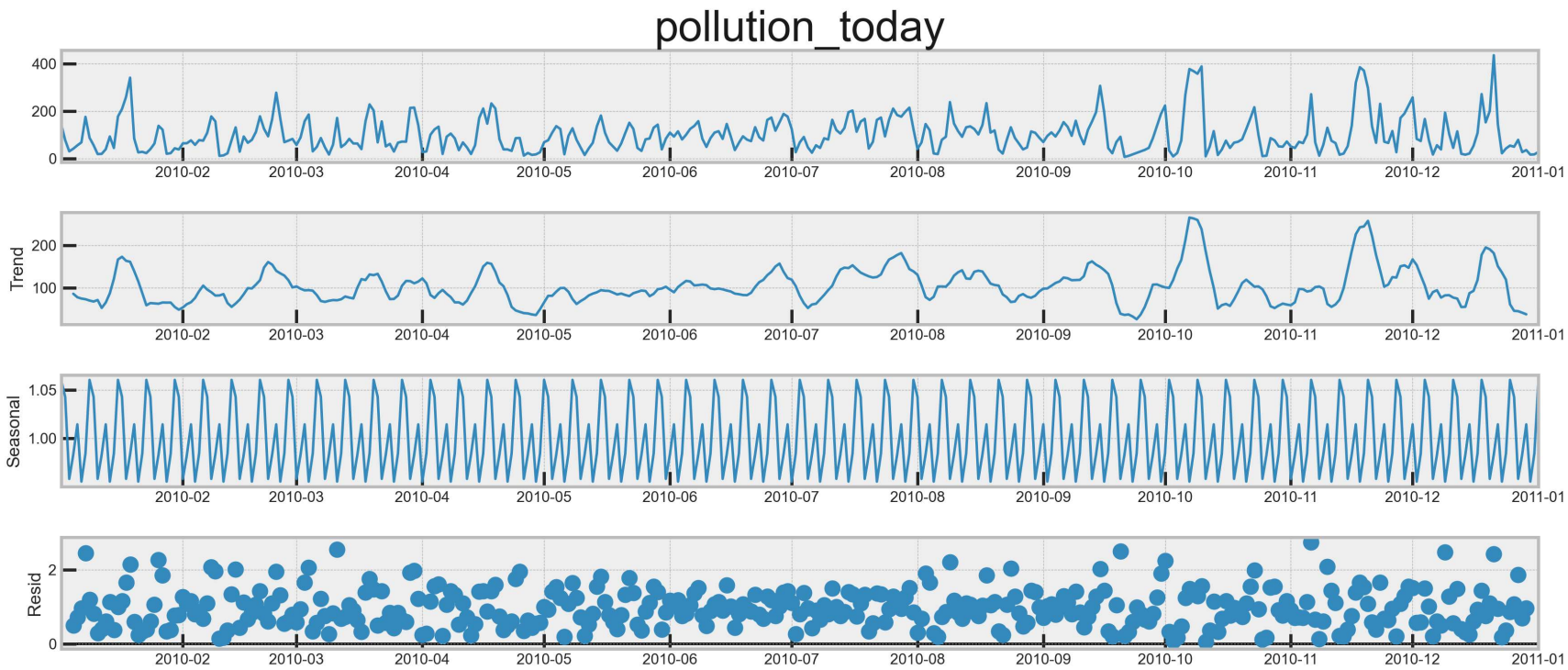
seasonalty for 2010

In [14]:
```
rcParams['figure.figsize'] = 18, 8
plt.figure(num=None, figsize=(50, 20), dpi=80, facecolor='y', edgecolor='k')
series = air_poll.pollution_today[:365]
result = seasonal_decompose(series, model='multiplicative')
result.plot()
```

Out[14]:



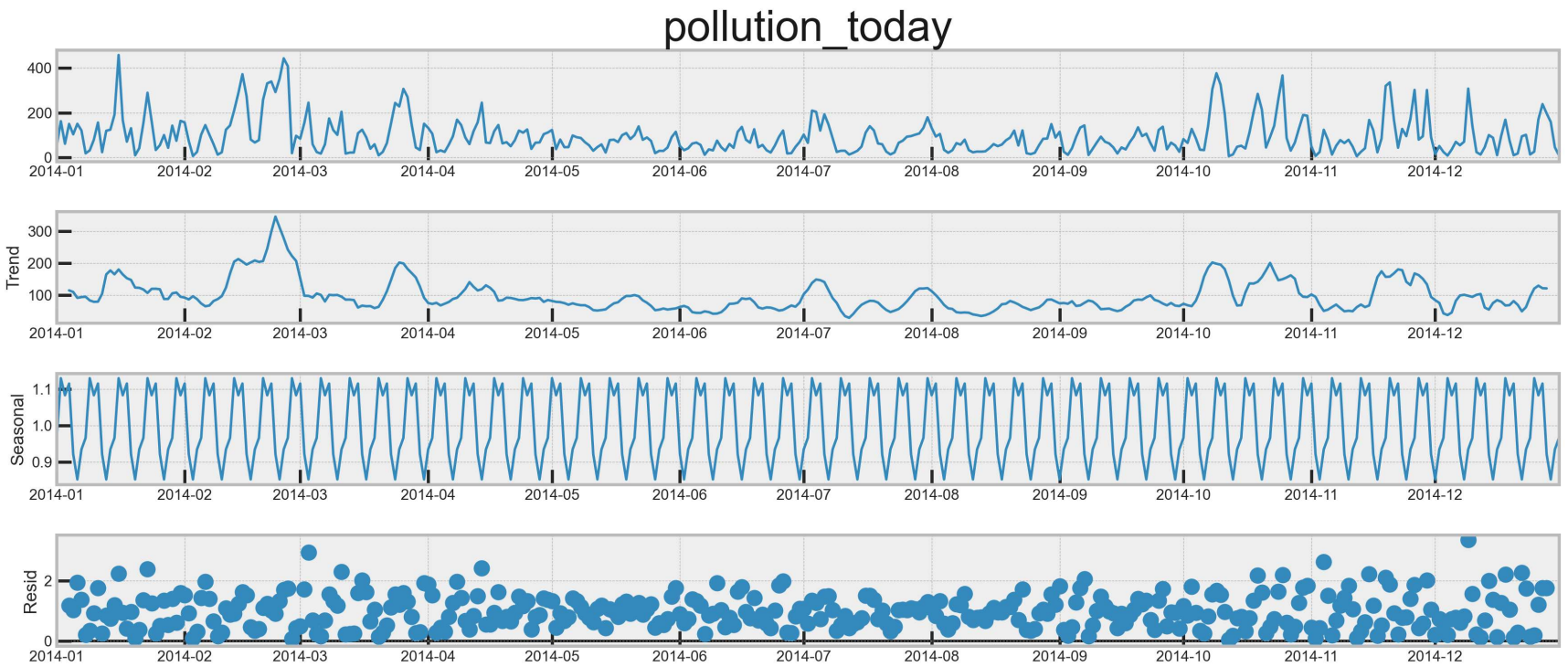
<Figure size 4000x1600 with 0 Axes>



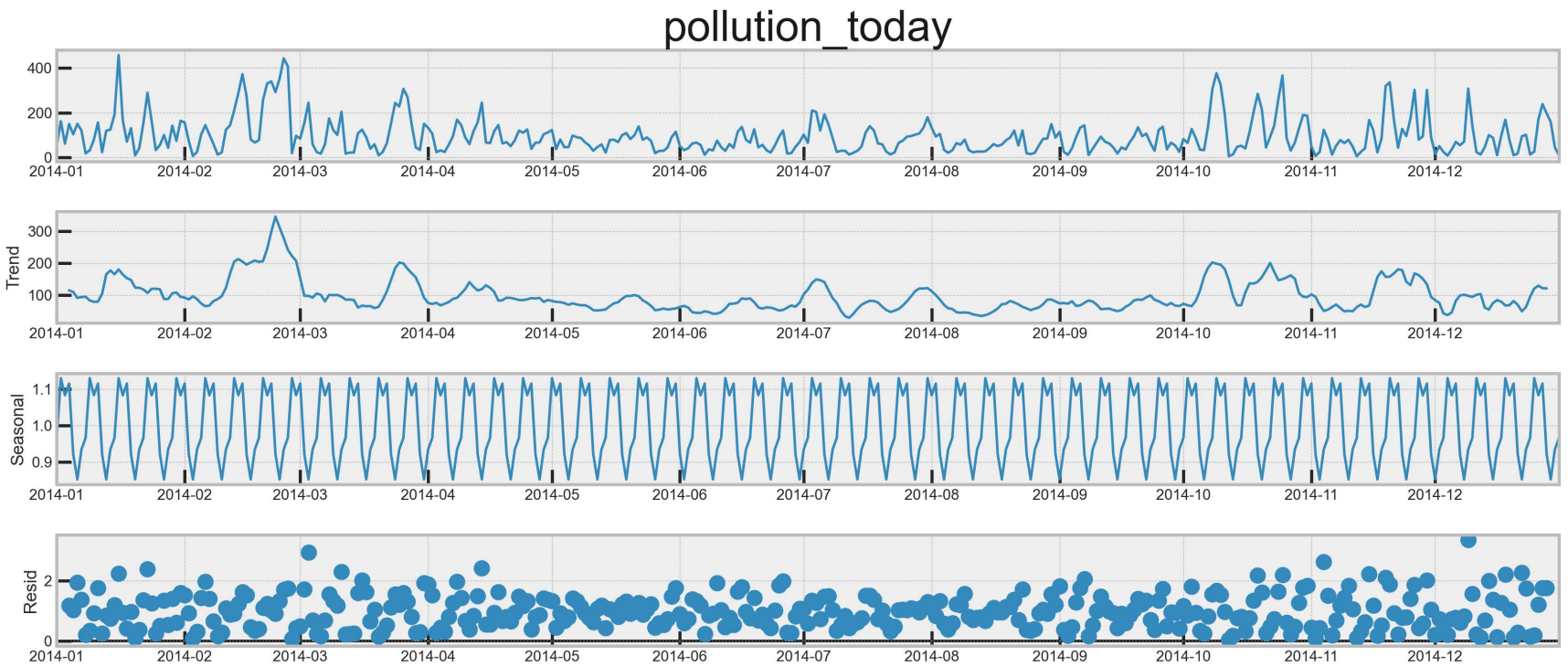
seasonalty for 2014

In [15]:
```
rcParams['figure.figsize'] = 18, 8
plt.figure(num=None, figsize=(50, 20), dpi=80, facecolor='w', edgecolor='k')
series = air_poll.pollution_today[-365:]
result = seasonal_decompose(series, model='multiplicative')
result.plot()
```

Out[15]:

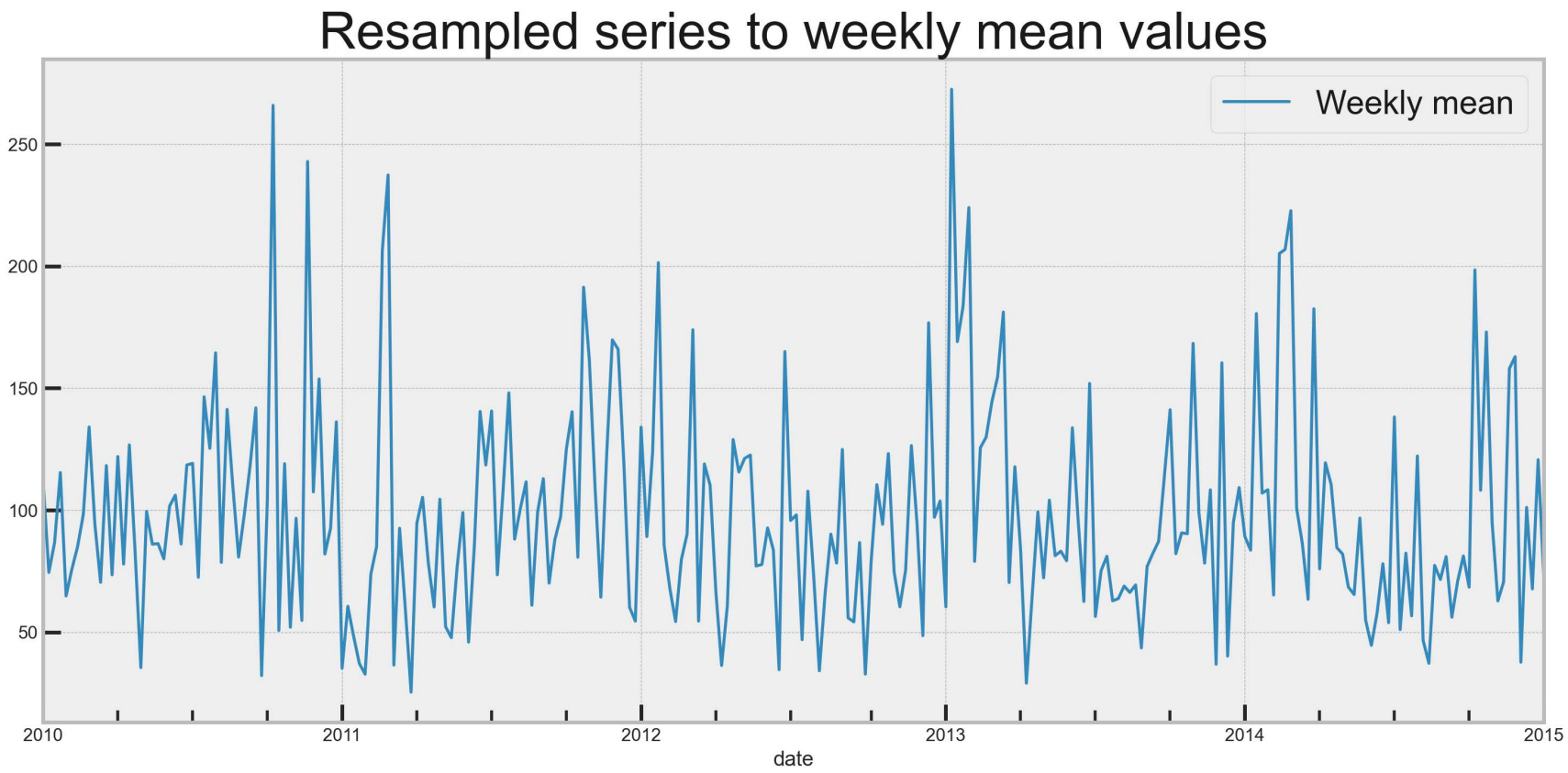


<Figure size 4000x1600 with 0 Axes>



In [16]:
```
resample = air_poll.resample('W')
weekly_mean = resample.mean()
weekly_mean.pollution_today.plot(label='Weekly mean')
plt.title("Resampled series to weekly mean values")
plt.legend()
```

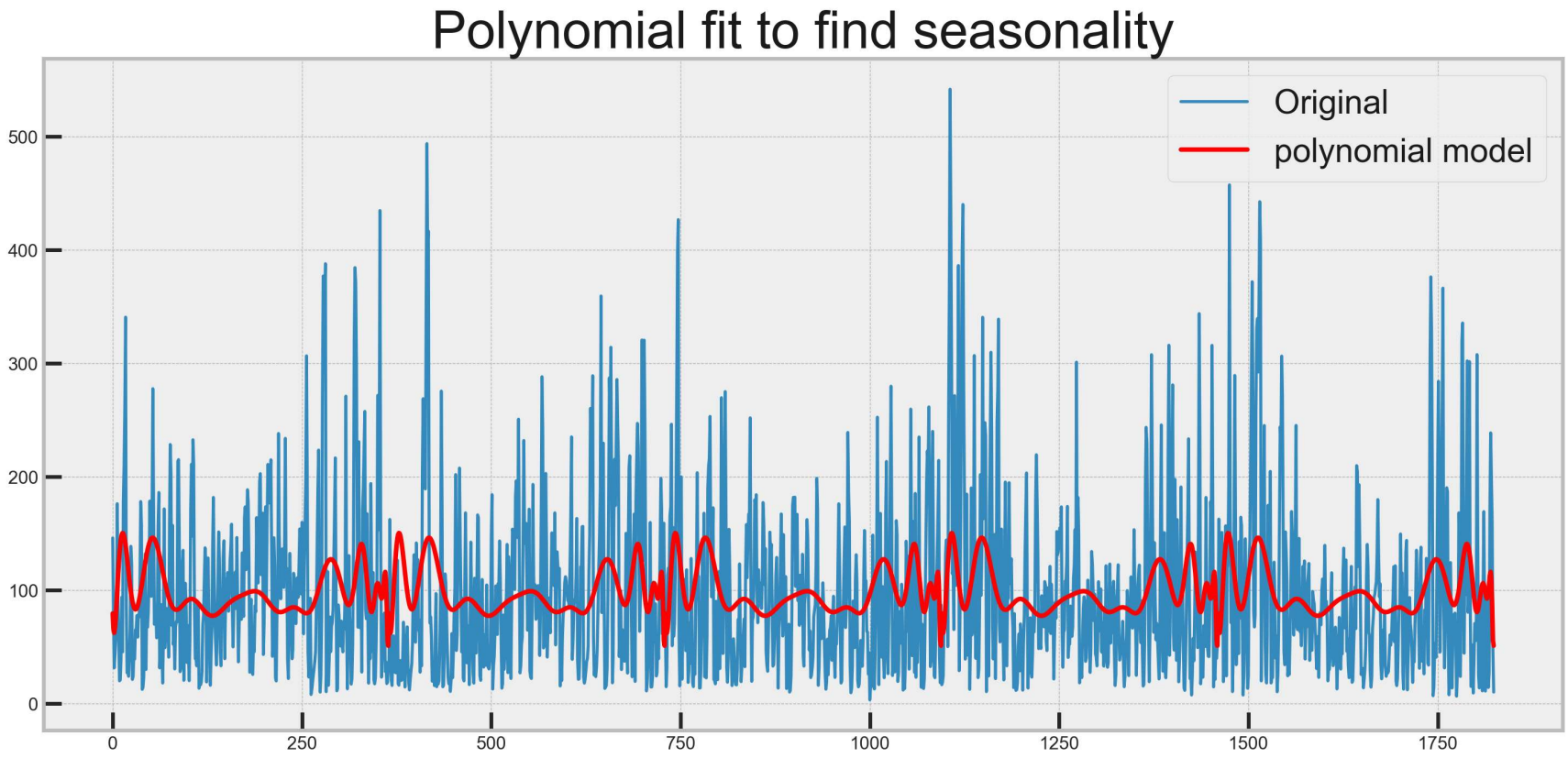
Out[16]:
```
<matplotlib.legend.Legend at 0x18d664ae700>
```



method to find the seasonality

In [17]:
```
# Fix xticks to show dates
# fit polynomial: x^2*b1 + x*b2 + ... + bn
series = air_poll.pollution_today.values
X = [i % 365 for i in range(0, len(series))]
y = series
degree = 100
coef = np.polyfit(X, y, degree)
# create curve
curve = list()
for i in range(len(X)):
    value = coef[-1]
    for d in range(degree):
        value += X[i]**(degree-d) * coef[d]
    curve.append(value)
# plot curve over original data
```

```
plt.plot(series, label='Original')
plt.plot(curve, color='red', linewidth=3, label='polynomial model')
plt.legend()
plt.title("Polynomial fit to find seasonality")
plt.show()
```



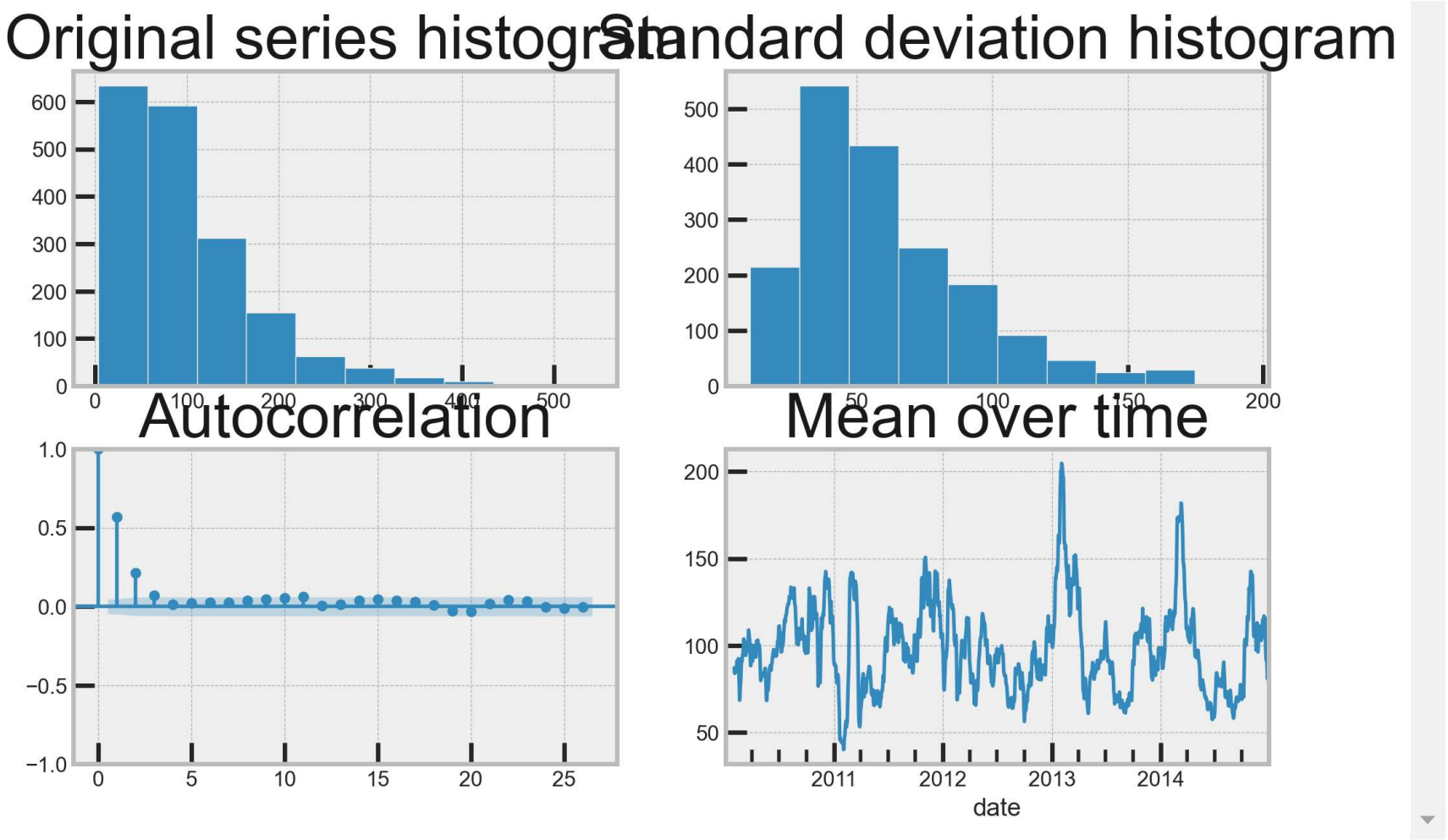
Noise

- We will also have a noise component to our time series, which is primarily white noise . White noise occurs when measurements have a mean of zero and are independent and identically distributed. All measurements will have the same variance, and no correlation will exist between them.
- We should aim for models with errors close to white noise if our time series consists of white noise (as it is random).
    - Is there white noise in our series?
    - Is our series histogram Gaussian? A mean of zero and a standard deviation of one

- Graphs of correlation
- Does it have a Gaussian standard deviation?
- Does the mean or level change over time?

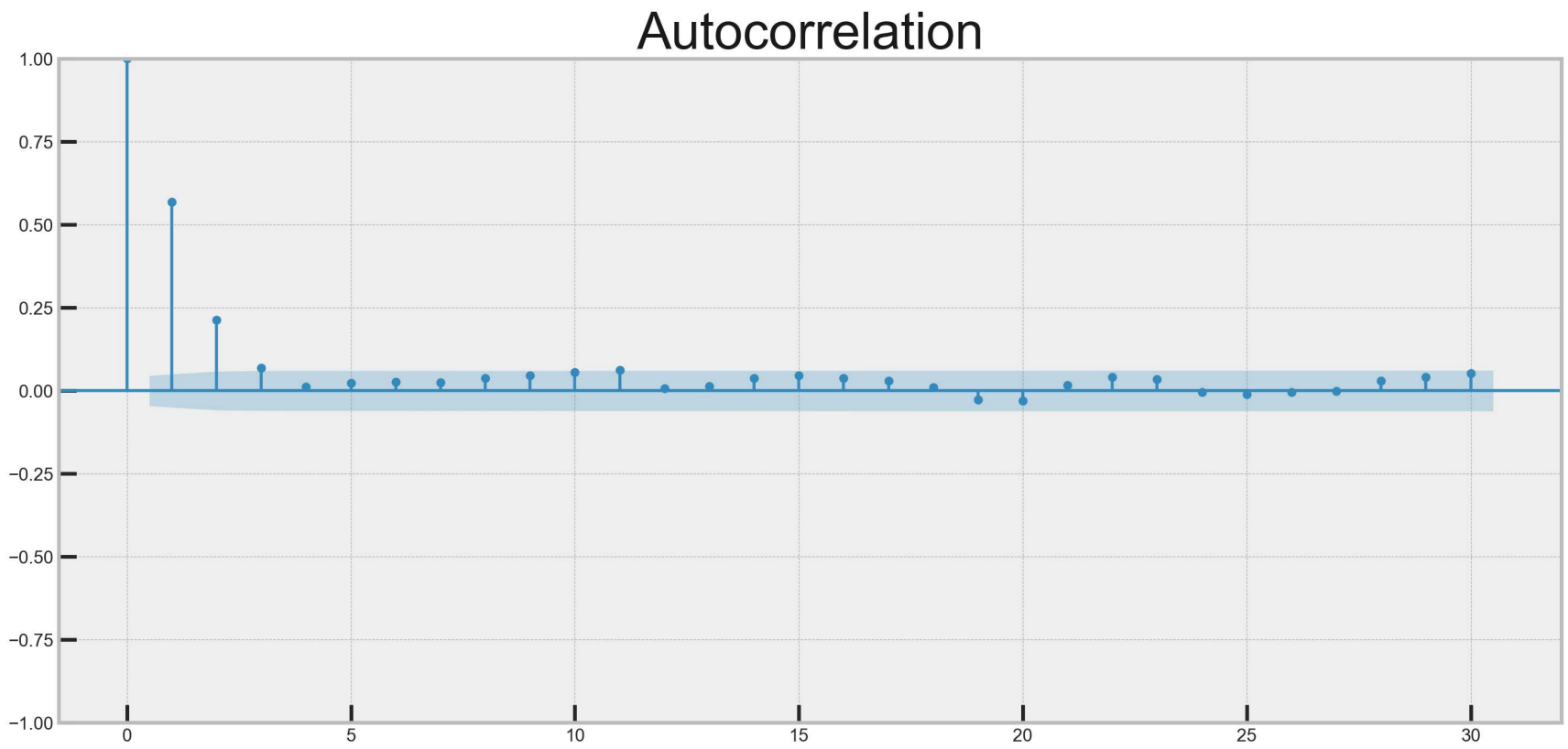
In [18]:
```
fig = plt.figure(figsize=(12, 7))
layout = (2, 2)
hist_ax = plt.subplot2grid(layout, (0, 0))
ac_ax = plt.subplot2grid(layout, (1, 0))
hist_std_ax = plt.subplot2grid(layout, (0, 1))
mean_ax = plt.subplot2grid(layout, (1, 1))

air_poll.pollution_today.hist(ax=hist_ax)
hist_ax.set_title("Original series histogram")

plot_acf(series, lags=26, ax=ac_ax)
ac_ax.set_title("Autocorrelation")

mm = air_poll.pollution_today.rolling(8).std()
mm.hist(ax=hist_std_ax)
hist_std_ax.set_title("Standard deviation histogram")

mm = air_poll.pollution_today.rolling(30).mean()
mm.plot(ax=mean_ax)
mean_ax.set_title("Mean over time")
```

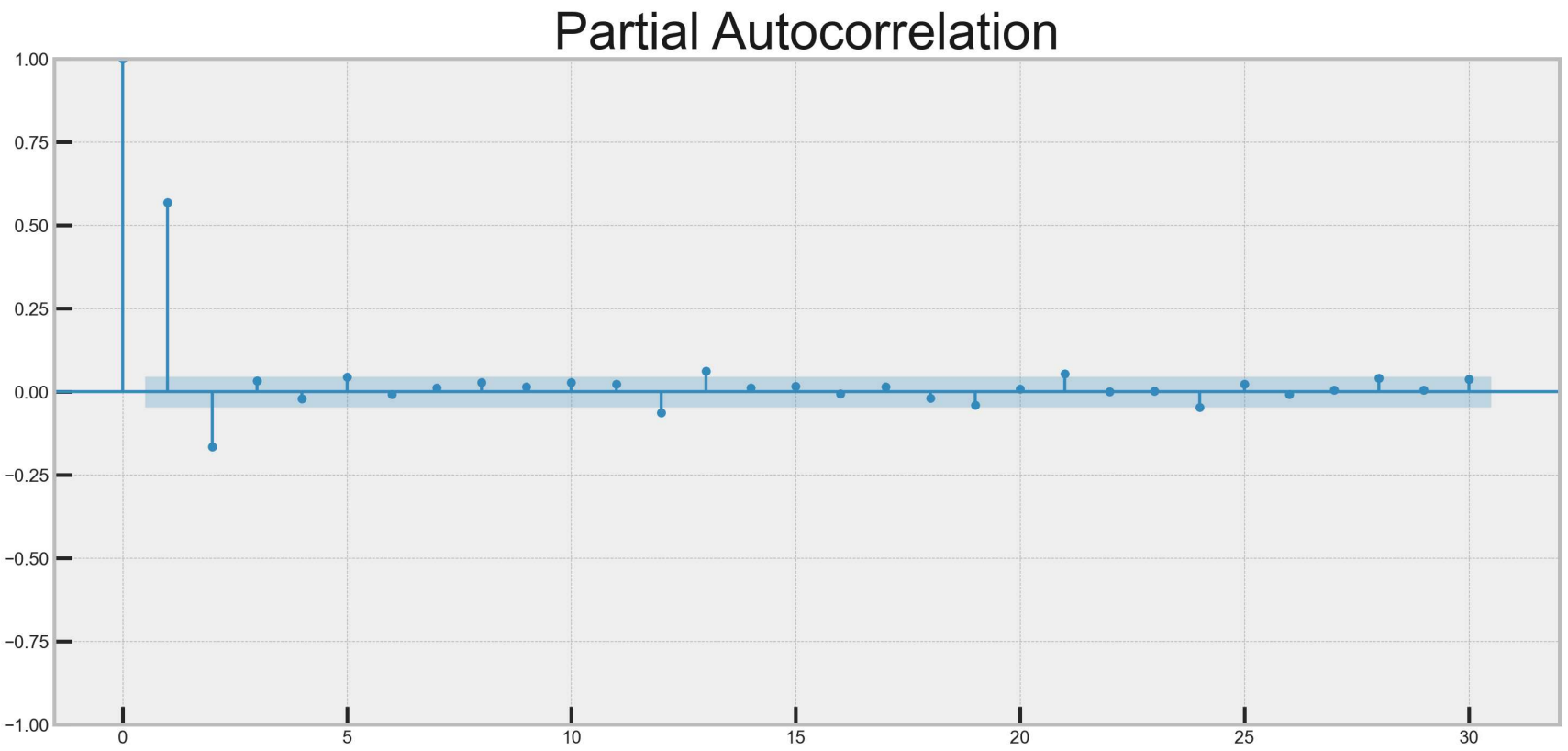
Out[18]:
```
Text(0.5, 1.0, 'Mean over time')
```



Before was the a manual method to test or find the seasonality and finding the white noice

In [19]:
```
plot_acf(series, lags=30)
plot_pacf(series, lags=30)
plt.show()
```

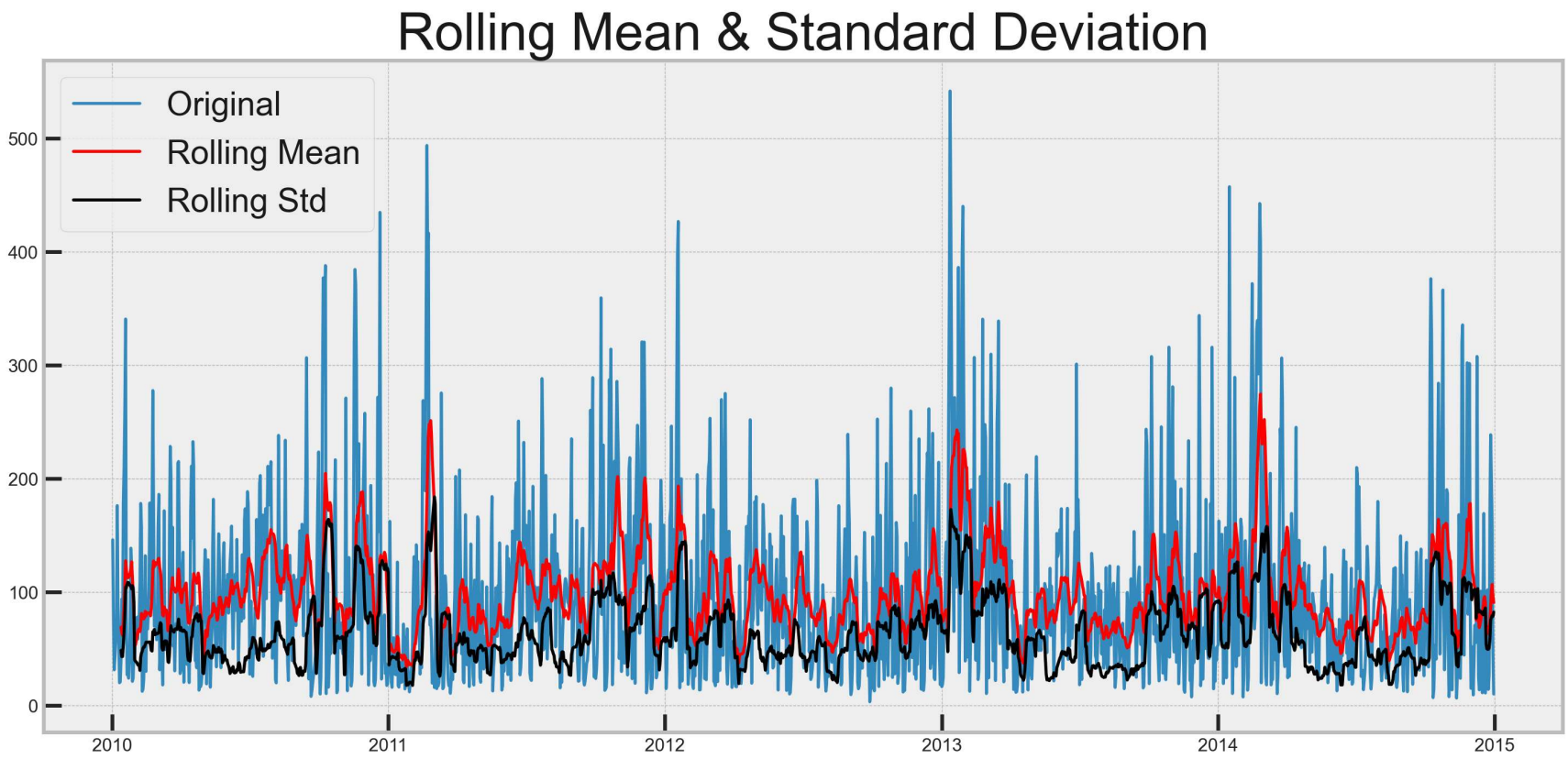

Autocorrelation
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
5
10
15
20
25
30



with mean and standard deviation of our series

- We can observe that although while our mean and standard deviation fluctuate from year to year, they consistently behave in the same way the following year. This serves as further evidence that the series is stationary.

In [20]:
```
# Determing rolling statistics
rolmean = air_poll.pollution_today.rolling(window=12).mean()
rolstd = air_poll.pollution_today.rolling(window=12).std()

# Plot rolling statistics:
orig = plt.plot(air_poll.pollution_today, label='Original')
mean = plt.plot(rolmean, color='red', label='Rolling Mean')
std = plt.plot(rolstd, color='black', label='Rolling Std')
plt.legend(loc='best')
plt.title('Rolling Mean & Standard Deviation')
plt.show(block=False)
```



Dickey-Fuller test

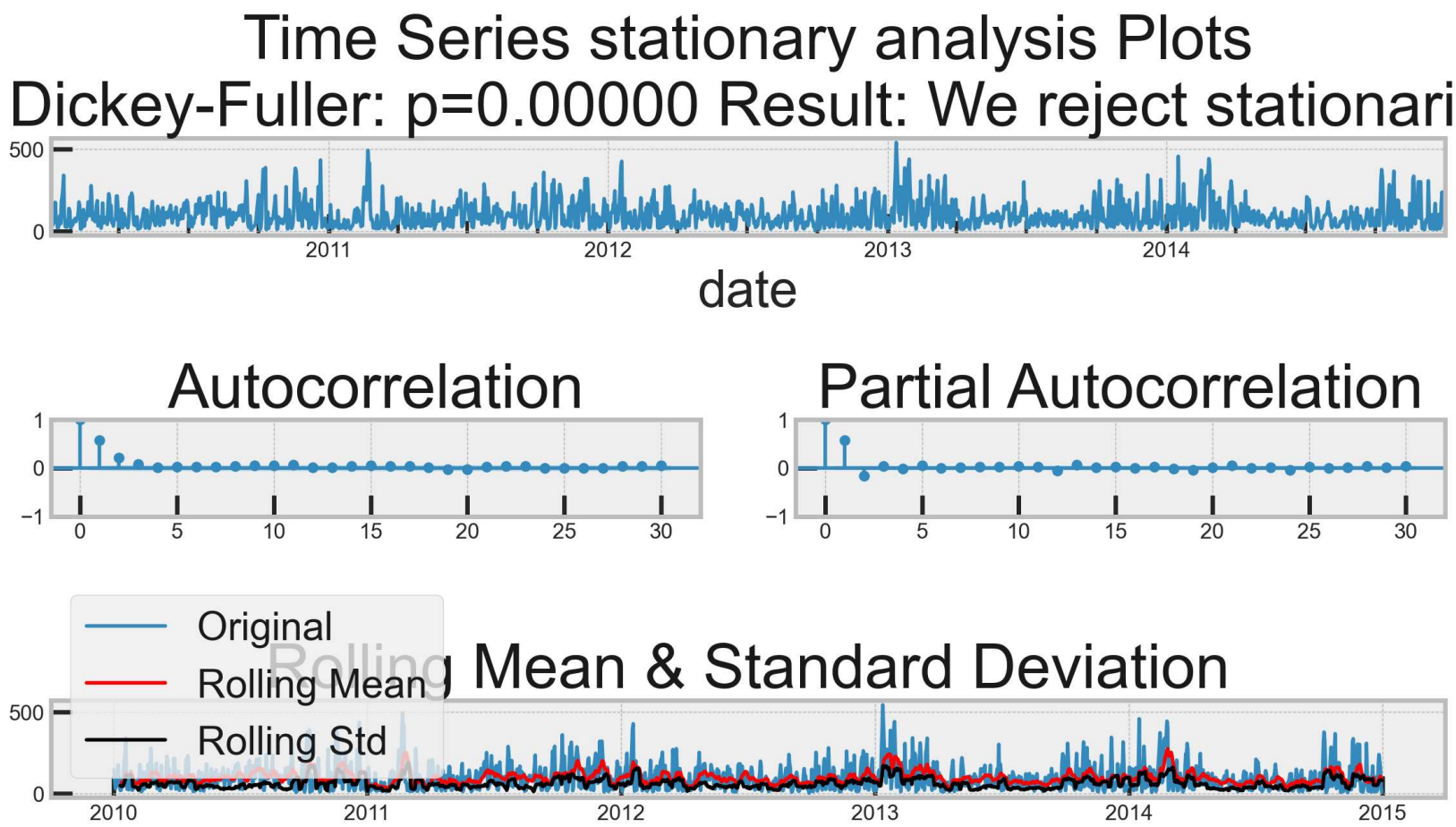
- A unit root test is a sort of statistical test that includes the Augmented Dickey-Fuller test. A unit root test is intended to assess the degree to which a time series is dominated by a trend. The Augmented Dickey-Fuller test may be one of the most popular unit root tests out of the many available. The information criteria is optimised over a range of various lag values using an autoregressive model.
- provide a method to quickly perform all the previous methods into one single function call and a graph

In [21]:
```
X = air_poll.pollution_today.values
result = adfuller(X)
print('ADF Statistic: %f' % result[0])
print('p-value: %f' % result[1])
print('Critical Values:')
for key, value in result[4].items():
    print('\t%s: %.3f' % (key, value))
```

```
ADF Statistic: -10.116719
p-value: 0.000000
Critical Values:
        1%: -3.434
        5%: -2.863
        10%: -2.568
```

In [22]:

```
def tsplot(y, lags=None, figsize=(12, 7), syle='bmh'):
    if not isinstance(y, pd.Series):
        y = pd.Series(y)

    with plt.style.context(style='bmh'):
        fig = plt.figure(figsize=(12, 7))
        layout = (3, 2)
        ts_ax = plt.subplot2grid(layout, (0, 0), colspan=2)
        acf_ax = plt.subplot2grid(layout, (1, 0))
        pacf_ax = plt.subplot2grid(layout, (1, 1))
        mean_std_ax = plt.subplot2grid(layout, (2, 0), colspan=2)
        y.plot(ax=ts_ax)
        p_value = sm.tsa.stattools.adfuller(y)[1]
        hypothesis_result = "We reject stationarity" if p_value <= 0.05 else "We can not reject stationarity"
        ts_ax.set_title(
            'Time Series stationary analysis Plots\n Dickey-Fuller: p={0:.5f} Result: {1}'.format(p_value, hypothesis_r
        smt.graphics.plot_acf(y, lags=lags, ax=acf_ax)
        smt.graphics.plot_pacf(y, lags=lags, ax=pacf_ax)
        plt.tight_layout()

        rolmean = air_poll.pollution_today.rolling(window=12).mean()
        rolstd = air_poll.pollution_today.rolling(window=10).std()

        # Plot rolling statistics:
        orig = plt.plot(air_poll.pollution_today, label='Original')
        mean = plt.plot(rolmean, color='red', label='Rolling Mean')
        std = plt.plot(rolstd, color='black', label='Rolling Std')
        plt.legend(loc='best')
        plt.title('Rolling Mean & Standard Deviation')


tsplot(air_poll.pollution_today, lags=30)
```

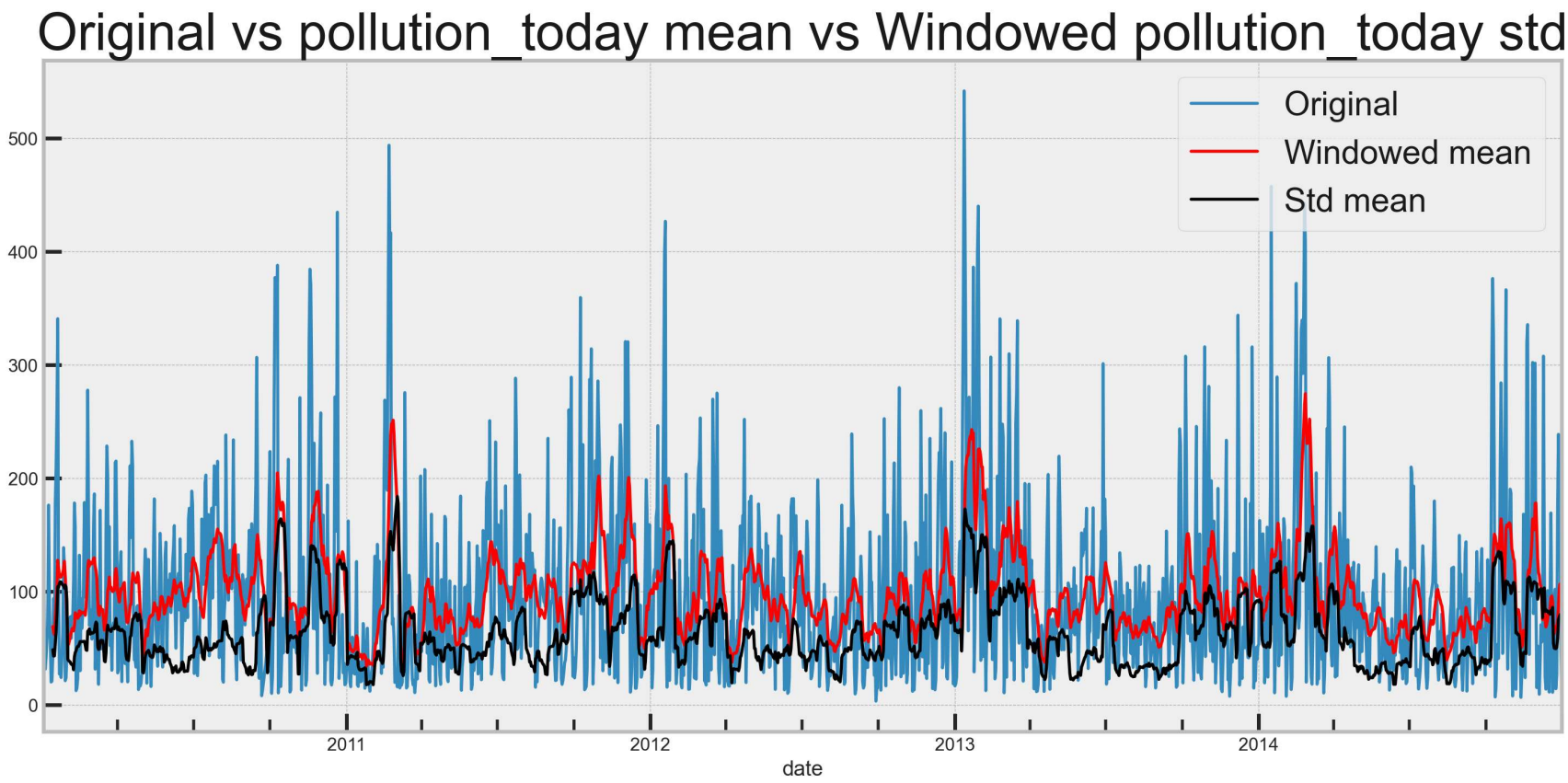


In [ ]:

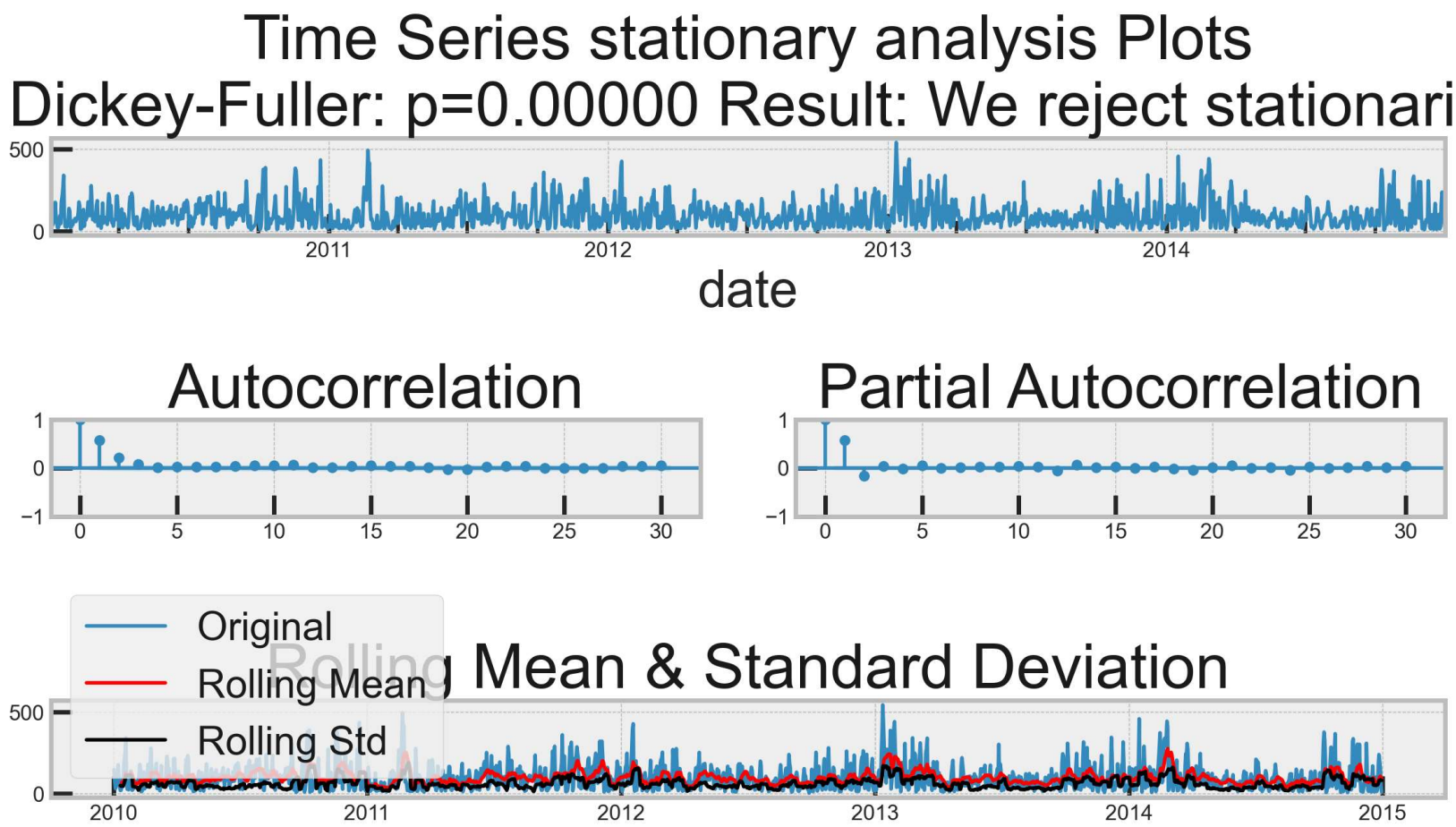
Making the Series stationary

In [23]:
```
air_poll.pollution_today.plot(label='Original')
air_poll.pollution_today.rolling(window=12).mean().plot(
    color='red', label='Windowed mean')
air_poll.pollution_today.rolling(window=12).std().plot(
    color='black', label='Std mean')
plt.legend()
plt.title('Original vs pollution_today mean vs Windowed pollution_today std')
```

Out[23]:
```
Text(0.5, 1.0, 'Original vs pollution_today mean vs Windowed pollution_today std')
```



In [24]:
```
tsplot(air_poll.pollution_today, lags=30)
```

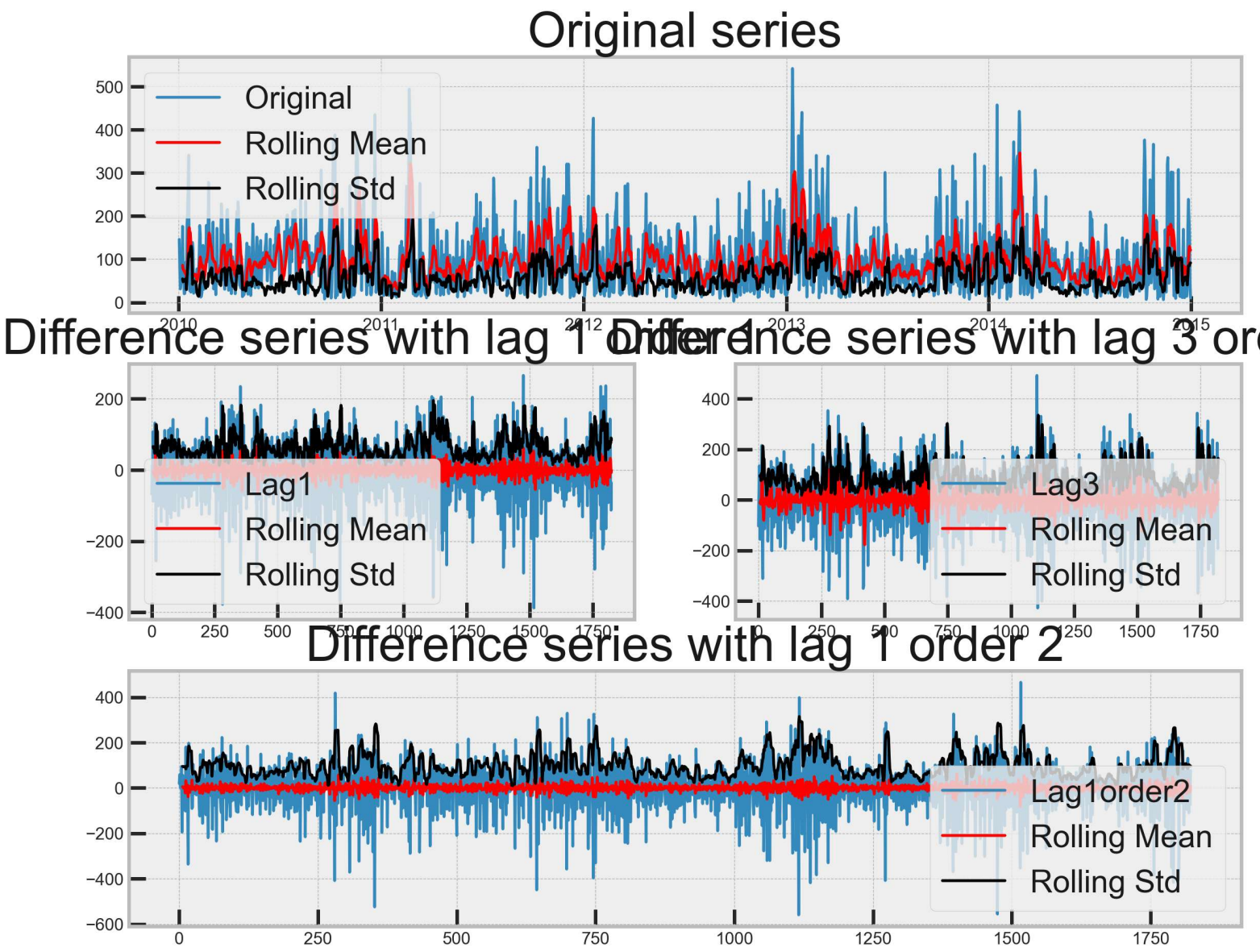


In [25]:
```
def difference(dataset, interval=1, order=1):
    for u in range(order):
        diff = list()
        for i in range(interval, len(dataset)):
            value = dataset[i] - dataset[i - interval]
            diff.append(value)
        dataset = diff
    return diff
```

In [26]:
```
lag1series = pd.Series(difference(air_poll.pollution_today, interval=1, order=1))
lag3series = pd.Series(difference(air_poll.pollution_today, interval=3, order=1))
lag1order2series = pd.Series(difference(
    air_poll.pollution_today, interval=1, order=2))

fig = plt.figure(figsize=(14, 11))
```

```
layout = (3, 2)
original = plt.subplot2grid(layout, (0, 0), colspan=2)
lag1 = plt.subplot2grid(layout, (1, 0))
lag3 = plt.subplot2grid(layout, (1, 1))
lag1order2 = plt.subplot2grid(layout, (2, 0), colspan=2)

original.set_title('Original series')
original.plot(air_poll.pollution_today, label='Original')
original.plot(air_poll.pollution_today.rolling(
    7).mean(), color='red', label='Rolling Mean')
original.plot(air_poll.pollution_today.rolling(7).std(),
              color='black', label='Rolling Std')
original.legend(loc='best')

lag1.set_title('Difference series with lag 1 order 1')
lag1.plot(lag1series, label="Lag1")
lag1.plot(lag1series.rolling(7).mean(), color='red', label='Rolling Mean')
lag1.plot(lag1series.rolling(7).std(), color='black', label='Rolling Std')
lag1.legend(loc='best')

lag3.set_title('Difference series with lag 3 order 1')
lag3.plot(lag3series, label="Lag3")
lag3.plot(lag3series.rolling(7).mean(), color='red', label='Rolling Mean')
lag3.plot(lag3series.rolling(7).std(), color='black', label='Rolling Std')
lag3.legend(loc='best')

lag1order2.set_title('Difference series with lag 1 order 2')
lag1order2.plot(lag1order2series, label="Lag1order2")
lag1order2.plot(lag1order2series.rolling(7).mean(),
                color='red', label='Rolling Mean')
lag1order2.plot(lag1order2series.rolling(7).std(),
                color='black', label='Rolling Std')
lag1order2.legend(loc='best')
```

Out[26]: <matplotlib.legend.Legend at 0x18d7579a070>

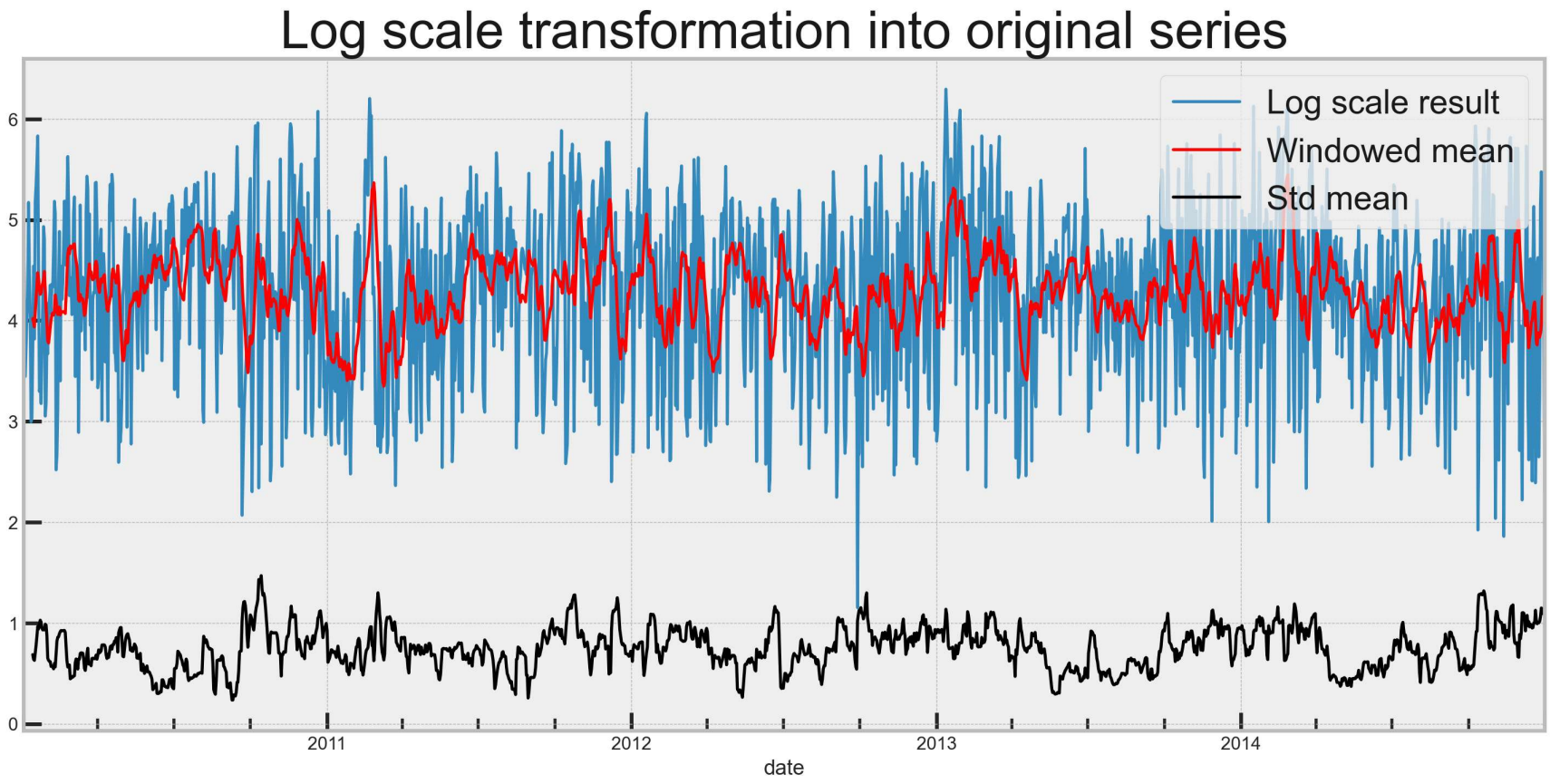


We can see how 1 order differencing doesnt really remove stationary but once we go with a order 2 difference it looks closer to a stationary series

scaling the lag

In [27]:
```
ts_log = np.log(air_poll.pollution_today)
ts_log.plot(label='Log scale result')
ts_log.rolling(window=12).mean().plot(color='red', label='Windowed mean')
ts_log.rolling(window=12).std().plot(color='black', label='Std mean')
plt.legend()
plt.title('Log scale transformation into original series')
```

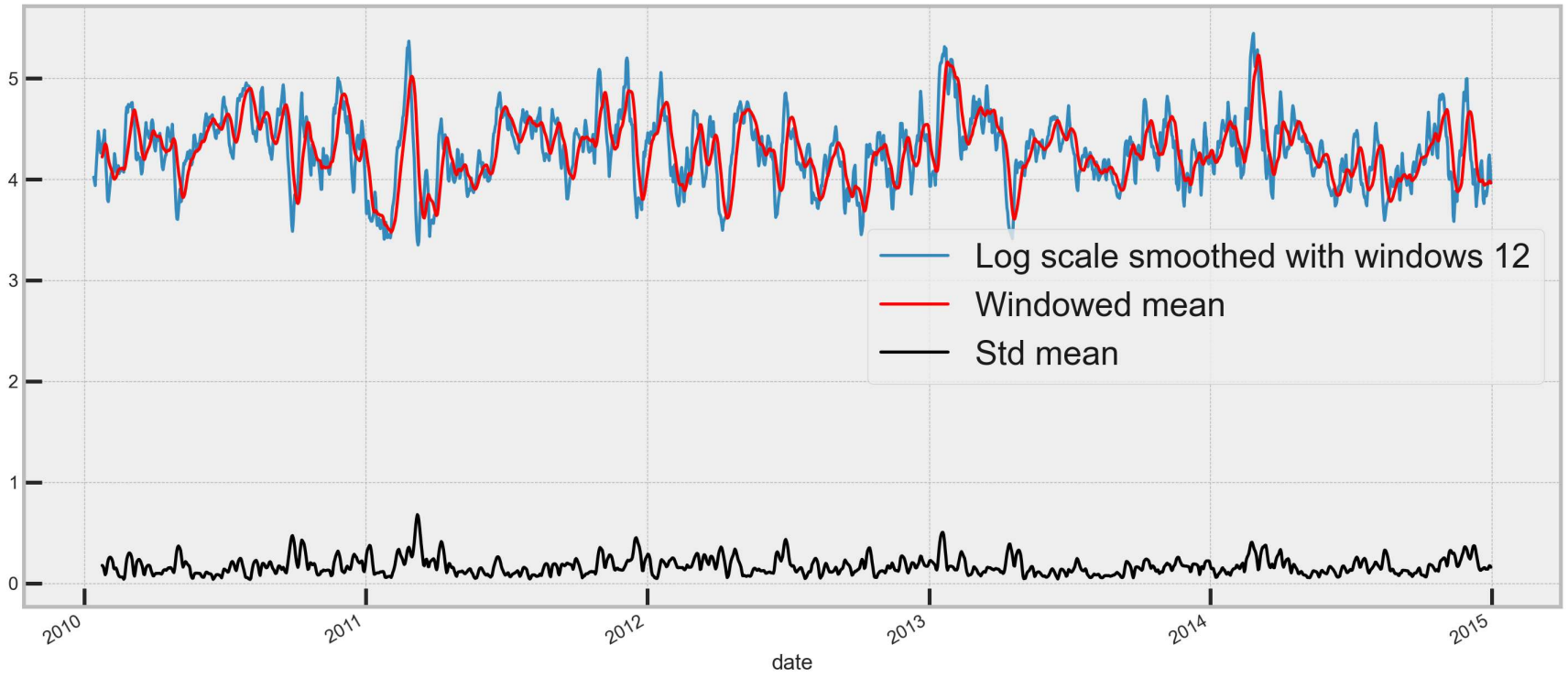
Out[27]:
```
Text(0.5, 1.0, 'Log scale transformation into original series')
```



smoothing the series

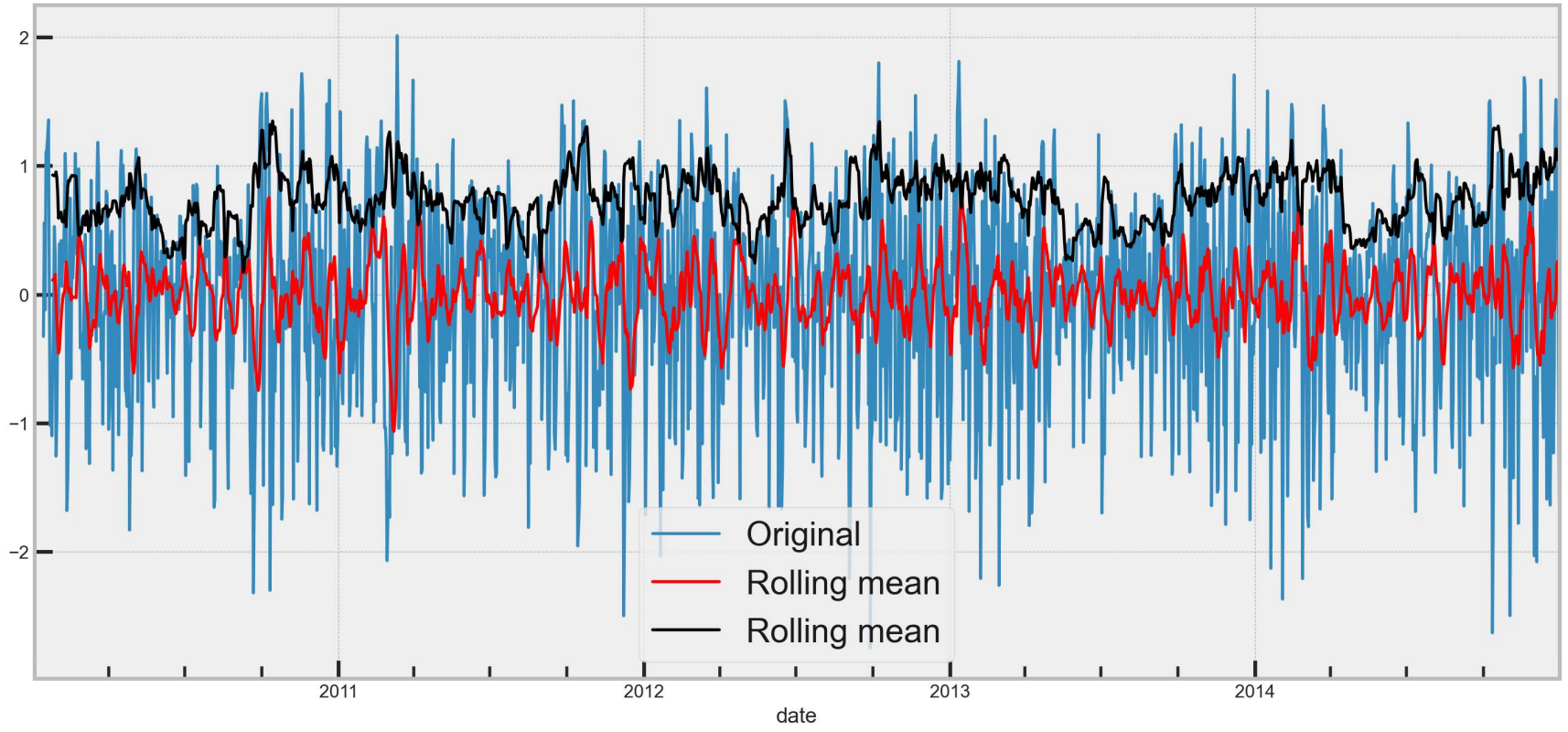
In [28]:
```
avg = pd.Series(ts_log).rolling(12).mean()
plt.plot(avg, label='Log scale smoothed with windows 12')
avg.rolling(window=12).mean().plot(color='red', label='Windowed mean')
avg.rolling(window=12).std().plot(color='black', label='Std mean')
plt.legend()
```

Out[28]: <matplotlib.legend.Legend at 0x18d72d798b0>



In [29]:
```
ts_log_moving_avg_diff = ts_log - avg

ts_log_moving_avg_diff.plot(label='Original')
ts_log_moving_avg_diff.rolling(12).mean().plot(
    color='red', label="Rolling mean")
ts_log_moving_avg_diff.rolling(12).std().plot(
    color='black', label="Rolling mean")
plt.legend(loc='best')
```

Out[29]: <matplotlib.legend.Legend at 0x18d6dd28d90>



Methods for time series forecasting

- spliting the data into trin and test

In [30]:
```
resultsDict = {}
predictionsDict = {}

split_date = '2014-01-01'
df_training = air_poll.loc[air_poll.index <= split_date]
df_test = air_poll.loc[air_poll.index > split_date]
print(f"{len(df_training)} days of training data \n {len(df_test)} days of testing data ")
```

```
1461 days of training data
 364 days of testing data
```

- t is also very important to include some naive forecast as the series mean or previous value to make sure our models perform better than the simplest of the simplest. We dont want to introduce any complexity if it does not provides any performance gain.

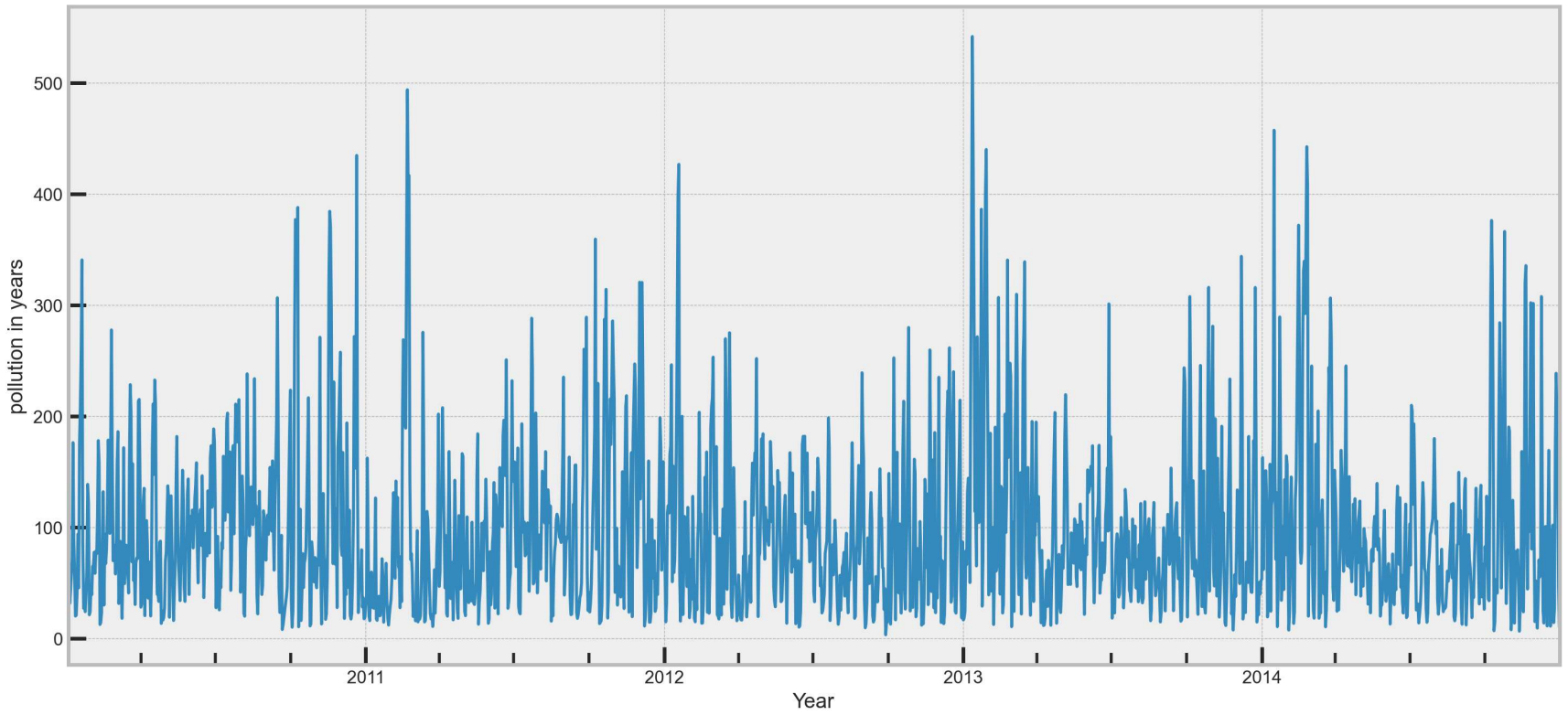
Univariate-time-series-forecasting

- ExponentialSmoothing
- SimpleExpSmoothing
- Holt

In [31]:
```
from statsmodels.tsa.api import ExponentialSmoothing, SimpleExpSmoothing, Holt
```

In [32]:
```
ax = air_poll.pollution_today.plot()
ax.set_xlabel("Year")
ax.set_ylabel("pollution in years")
```

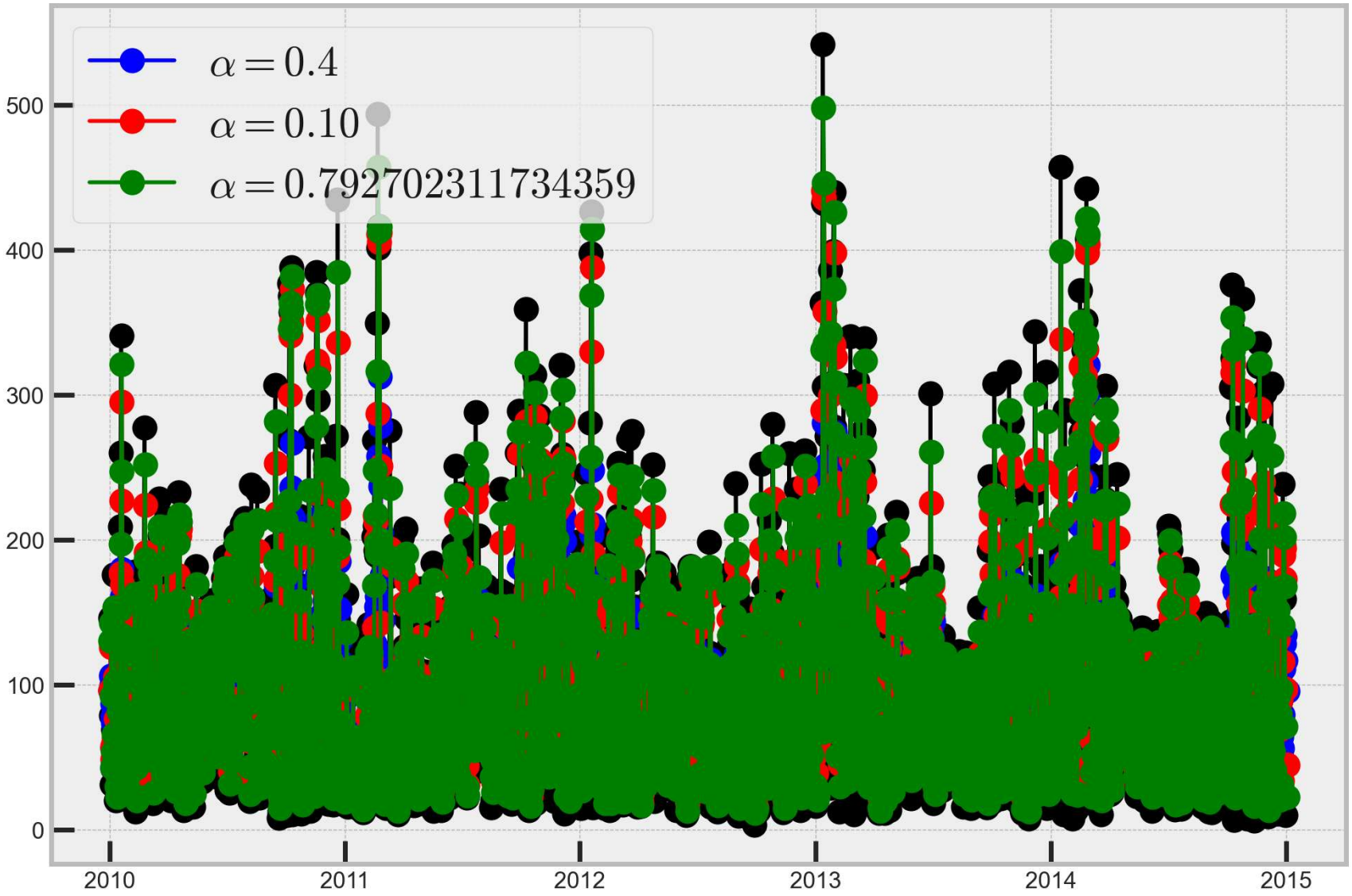
Out[32]:
```
Text(0, 0.5, 'pollution in years')
```



In [33]:
```
fit1 = SimpleExpSmoothing(air_poll.pollution_today, initialization_method="heuristic").fit(
    smoothing_level=0.2, optimized=False
)
fcast1 = fit1.forecast(3).rename(r"$\alpha=0.4$")
fit2 = SimpleExpSmoothing(air_poll.pollution_today, initialization_method="heuristic").fit(
    smoothing_level=0.6, optimized=False
)
fcast2 = fit2.forecast(3).rename(r"$\alpha=0.10$")
fit3 = SimpleExpSmoothing(air_poll.pollution_today, initialization_method="estimated").fit()
fcast3 = fit3.forecast(3).rename(r"$\alpha=%s$" % fit3.model.params["smoothing_level"])

plt.figure(figsize=(12, 8))
plt.plot(air_poll.pollution_today, marker="o", color="black")
plt.plot(fit1.fittedvalues, marker="o", color="blue")
(line1,) = plt.plot(fcast1, marker="o", color="blue")
plt.plot(fit2.fittedvalues, marker="o", color="red")
(line2,) = plt.plot(fcast2, marker="o", color="red")
plt.plot(fit3.fittedvalues, marker="o", color="green")
```

```
(line3,) = plt.plot(fcast3, marker="o", color="green")
plt.legend([line1, line2, line3], [fcast1.name, fcast2.name, fcast3.name])
```

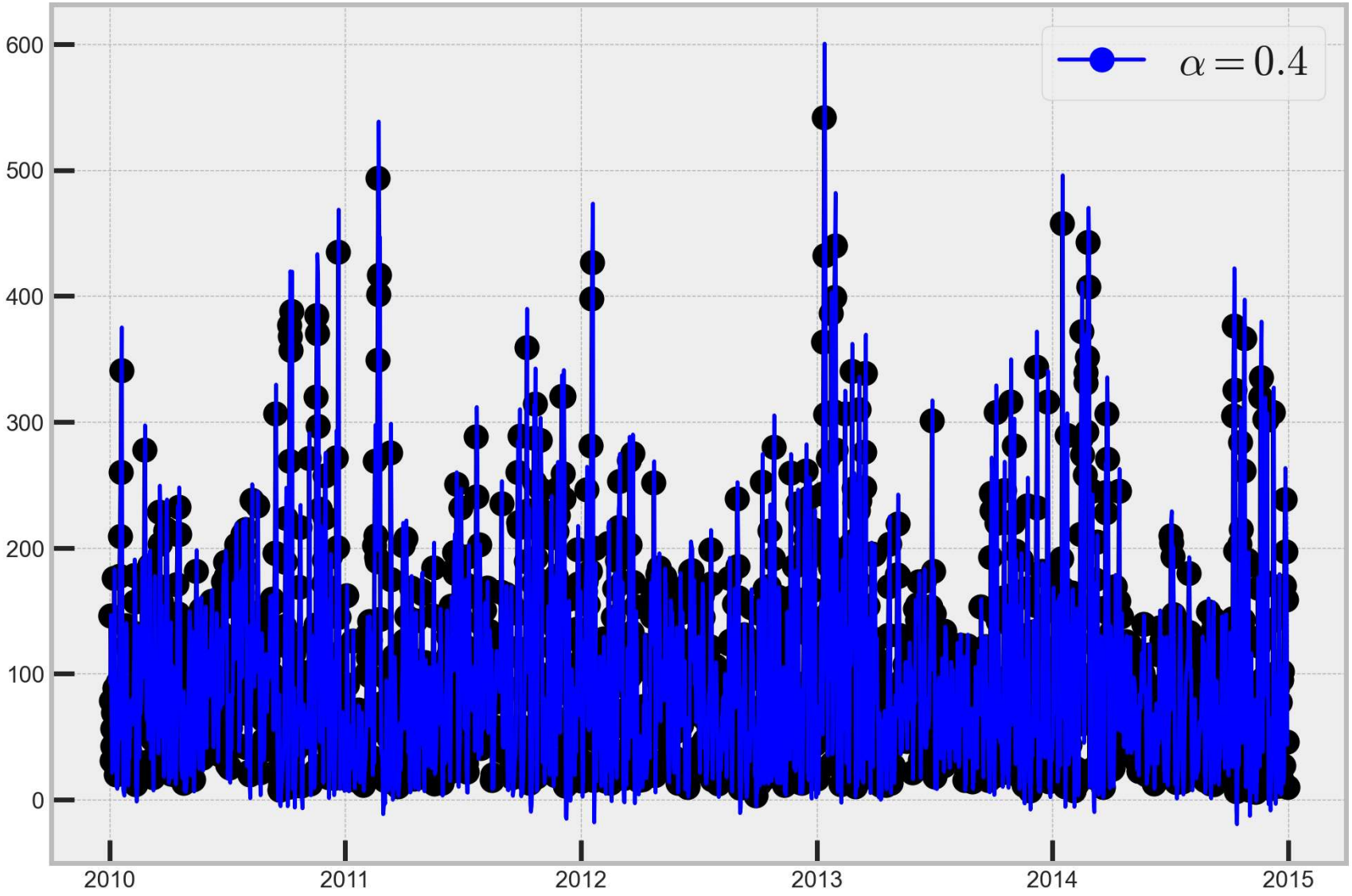
Out[33]: <matplotlib.legend.Legend at 0x18d62bc5b20>



In [34]:
```
fit1 = Holt(air_poll.pollution_today, initialization_method="estimated").fit(
    smoothing_level=0.9, smoothing_trend=0.2, optimized=False
)
plt.figure(figsize=(12, 8))
```

```
plt.plot(air_poll.pollution_today, marker="o", color="black")
plt.plot(fit1.fittedvalues, color="blue")
plt.legend([line1], [fcast1.name])
```

Out[34]: <matplotlib.legend.Legend at 0x18d62ae01c0>



In [35]:
```
from statsmodels.tsa.api import SimpleExpSmoothing

ses = SimpleExpSmoothing(air_poll.pollution_today)
```

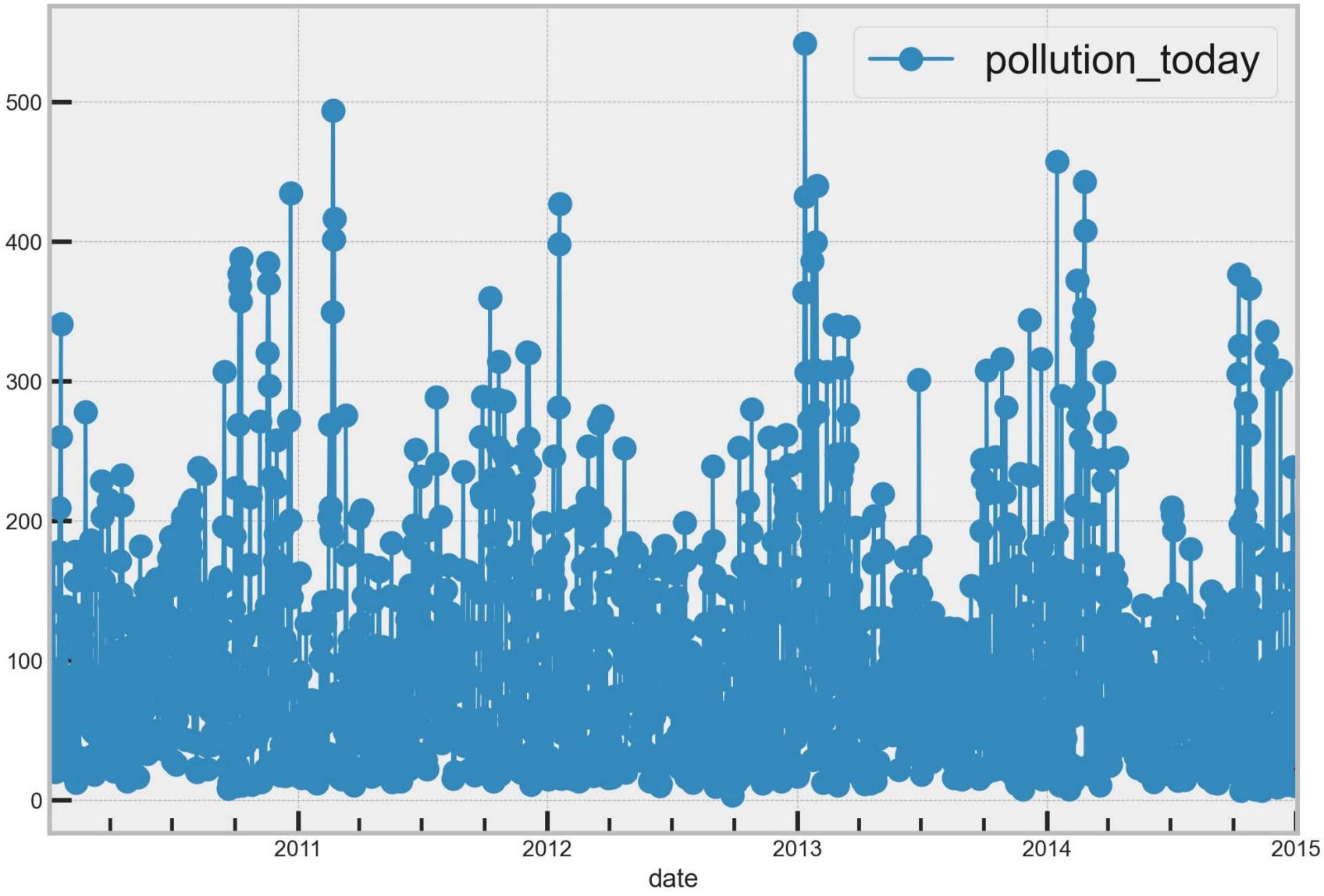
```
alpha = 0.8
model = ses.fit(smoothing_level = alpha, optimized = False)
```

In [36]:
```
forcast = model.forecast(3)
forcast
```

Out[36]:
```
2015-01-01   22.10872
2015-01-02   22.10872
2015-01-03   22.10872
Freq: D, dtype: float64
```

In [37]:
```
ax = air_poll.pollution_today.plot(marker = 'o', figsize = (12,8), legend = True)

forcast.plot(ax = ax)
```

Out[37]:
```
<AxesSubplot:xlabel='date'>
```



Simple Exponential Smoothing (SES)

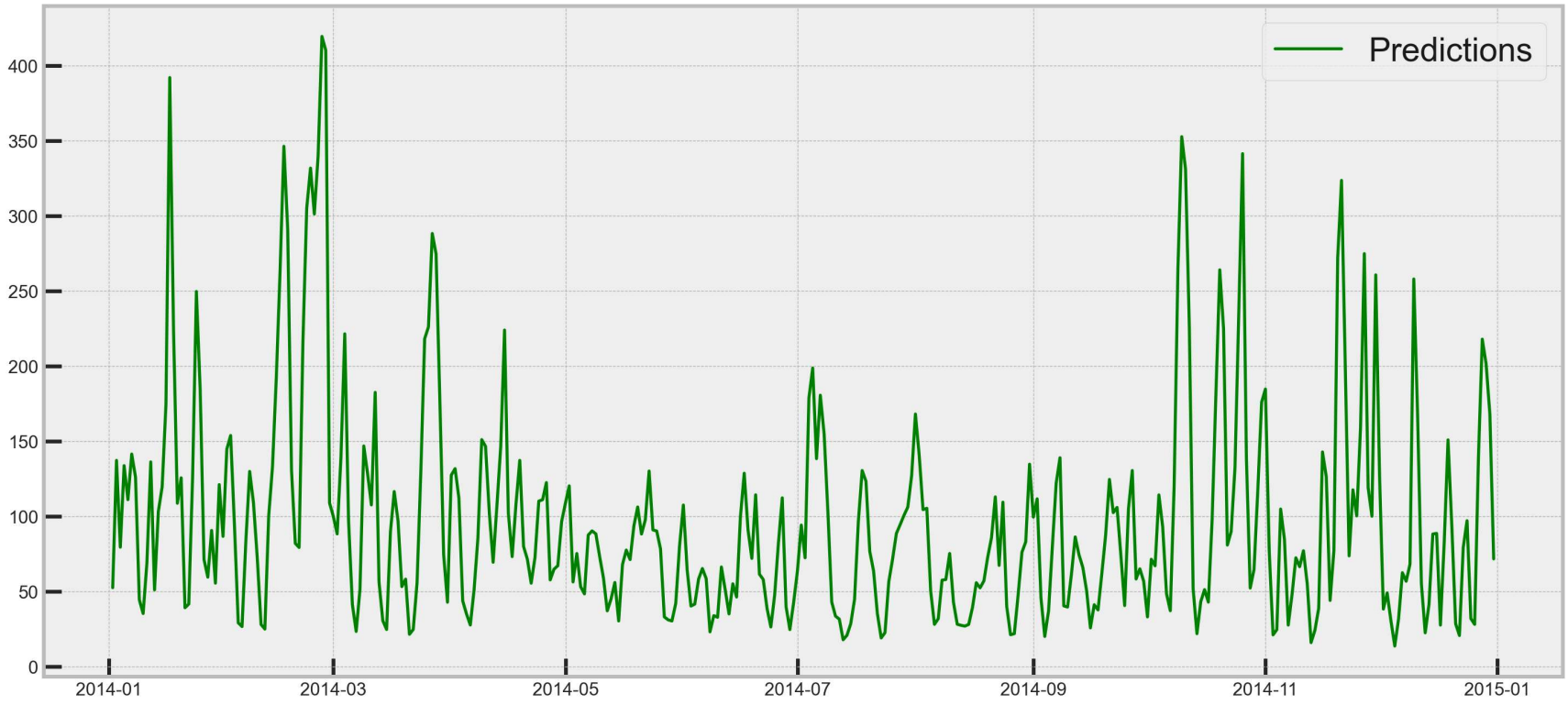
In [38]:
```
index = len(df_training)
yhat = list()
for t in tqdm(range(len(df_test.pollution_today))):
    temp_train = air_poll[:len(df_training)+t]
    model = SimpleExpSmoothing(temp_train.pollution_today)
```

```
    model_fit = model.fit()
    predictions = model_fit.predict(start=len(temp_train), end=len(temp_train))
    yhat = yhat + [predictions]

yhat = pd.concat(yhat)
predictionsDict['SES'] = yhat.values
```

```
100%|██████████| 364/364 [00:04<00:00, 81.40it/s]
```

In [39]:
```
plt.plot(yhat, color='green', label = 'Predictions')
plt.legend()
```

Out[39]:
```
<matplotlib.legend.Legend at 0x18d648819a0>
```



Holt Winter's Exponential Smoothing (HWES)

In [40]:
```
# Walk throught the test data, training and predicting 1 day ahead for all the test data
index = len(df_training)
yhat = list()
```

```
for t in tqdm(range(len(df_test.pollution_today))):
    temp_train = air_poll[:len(df_training)+t]
    model = ExponentialSmoothing(temp_train.pollution_today)
    model_fit = model.fit()
    predictions = model_fit.predict(start=len(temp_train), end=len(temp_train))
    yhat = yhat + [predictions]

yhat = pd.concat(yhat)
predictionsDict['SES'] = yhat.values
```

```
100%|██████████| 364/364 [00:04<00:00, 85.15it/s]
```

Autoregression (AR)

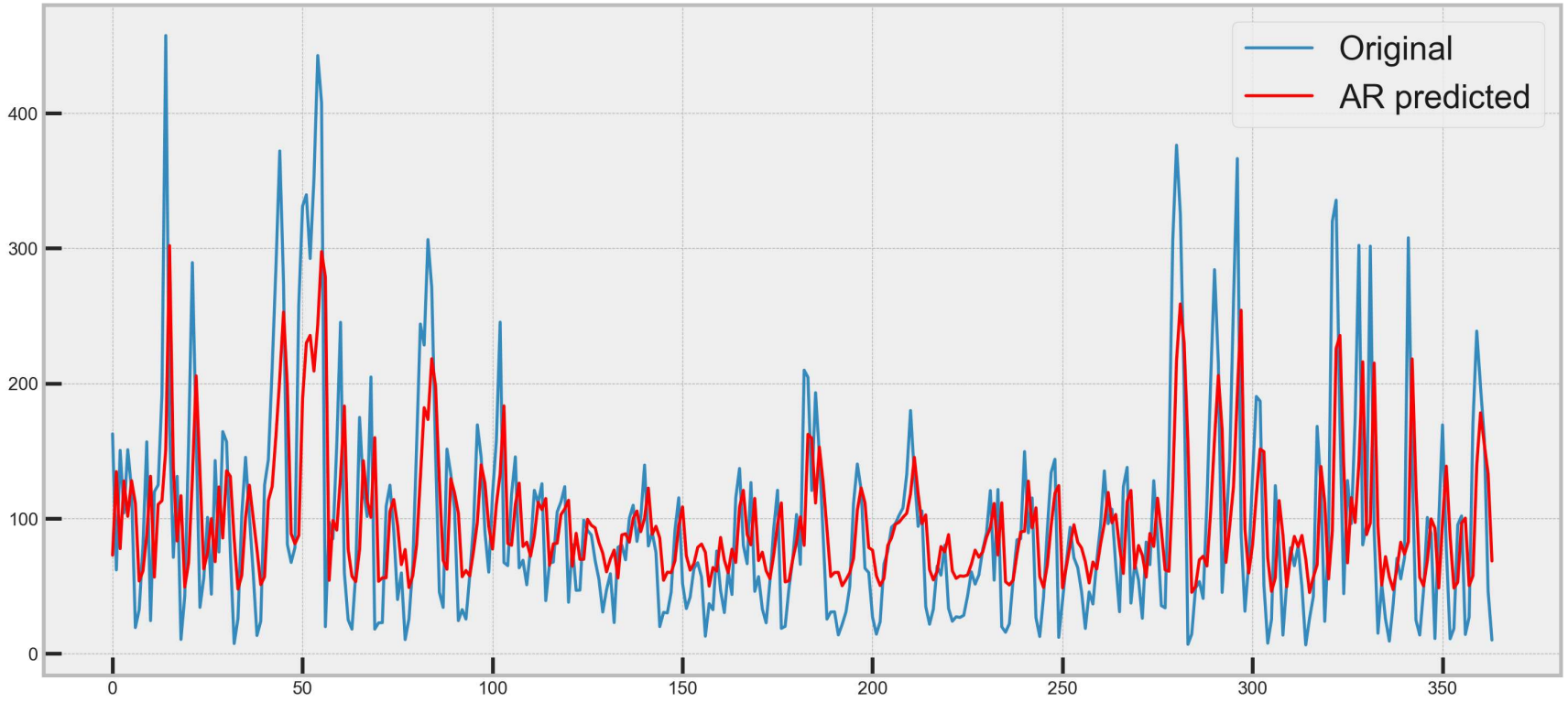
In [41]:
```
import statsmodels.api as sm
from statsmodels.tsa.ar_model import AutoReg
```

In [42]:
```
index = len(df_training)
yhat = list()
for t in tqdm(range(len(df_test.pollution_today))):
    temp_train = air_poll[:len(df_training)+t]
    model = AutoReg(temp_train.pollution_today,lags=1)
    model_fit = model.fit()
    predictions = model_fit.predict(
        start=len(temp_train), end=len(temp_train), dynamic=False)
    yhat = yhat + [predictions]

yhat = pd.concat(yhat)
predictionsDict['AutoReg'] = yhat.values
```

```
100%|██████████| 364/364 [00:01<00:00, 222.85it/s]
```

In [43]:
```
plt.plot(df_test.pollution_today.values, label='Original')
plt.plot(yhat.values, color='red', label='AR predicted')
plt.legend()
```

Out[43]:
```
<matplotlib.legend.Legend at 0x18d0360e400>
```



Moving Average (MA)

In [44]:
```
import statsmodels.api as sm
from statsmodels.tsa.ar_model import AutoReg
```

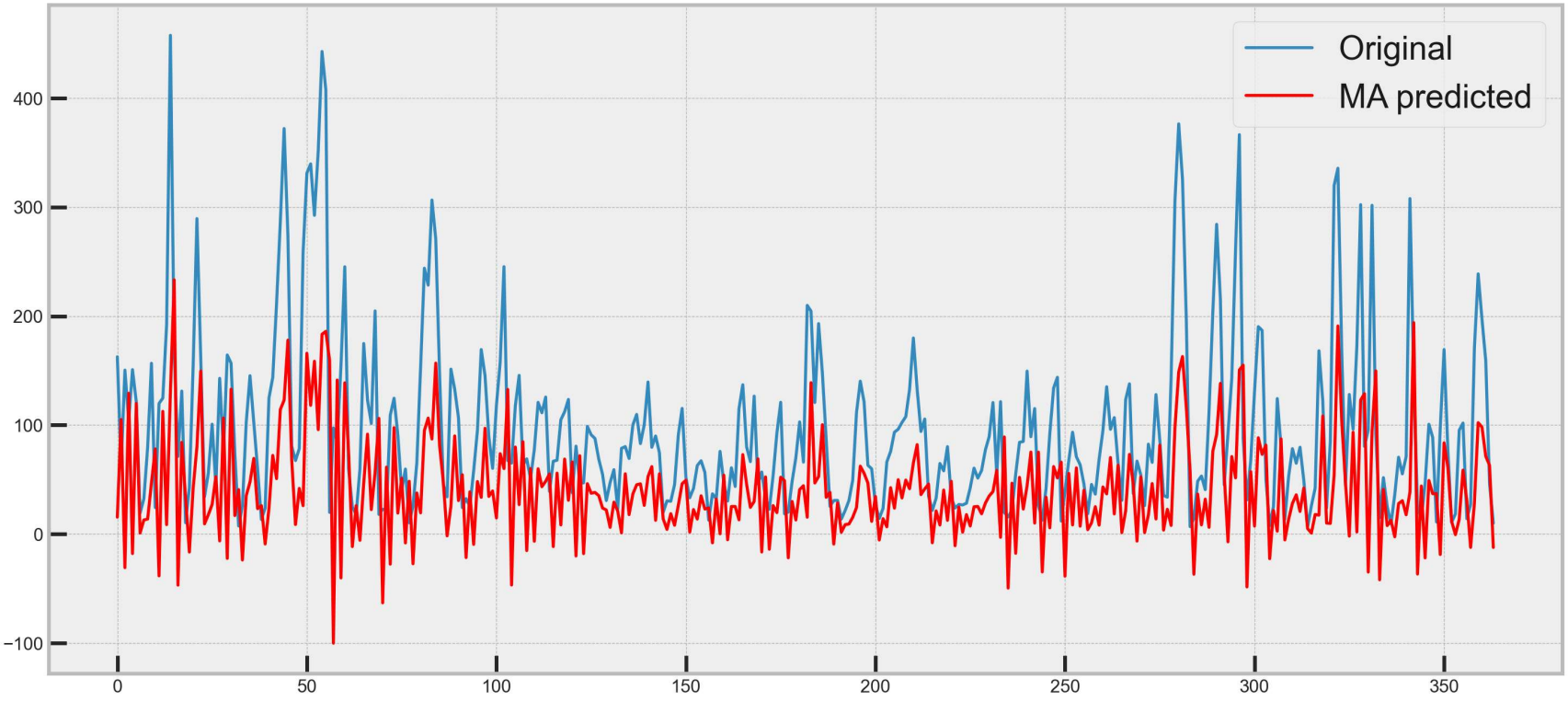
In [45]:
```
# MA example

# Walk throught the test data, training and predicting 1 day ahead for all the test data
index = len(df_training)
yhat = list()
for t in tqdm(range(len(df_test.pollution_today))):
    temp_train = air_poll[:len(df_training)+t]
    model = SARIMAX(temp_train.pollution_today, order=(0,0,1))
    model_fit = model.fit(disp=False)
    predictions = model_fit.predict(
        start=len(temp_train), end=len(temp_train), dynamic=False)
    yhat = yhat + [predictions]
```

```
yhat = pd.concat(yhat)
# resultsDict['ARMA'] = evaluate(df_test.pollution_today, yhat.values)
predictionsDict['MA'] = yhat.values
```

```
100%|██████████| 364/364 [00:50<00:00,  7.25it/s]
```

In [46]:
```
plt.plot(df_test.pollution_today.values, label='Original')
plt.plot(yhat.values, color='red', label='MA predicted')
plt.legend()
```

Out[46]:
```
<matplotlib.legend.Legend at 0x18d036c6c70>
```



Autoregressive Moving Average (ARMA)

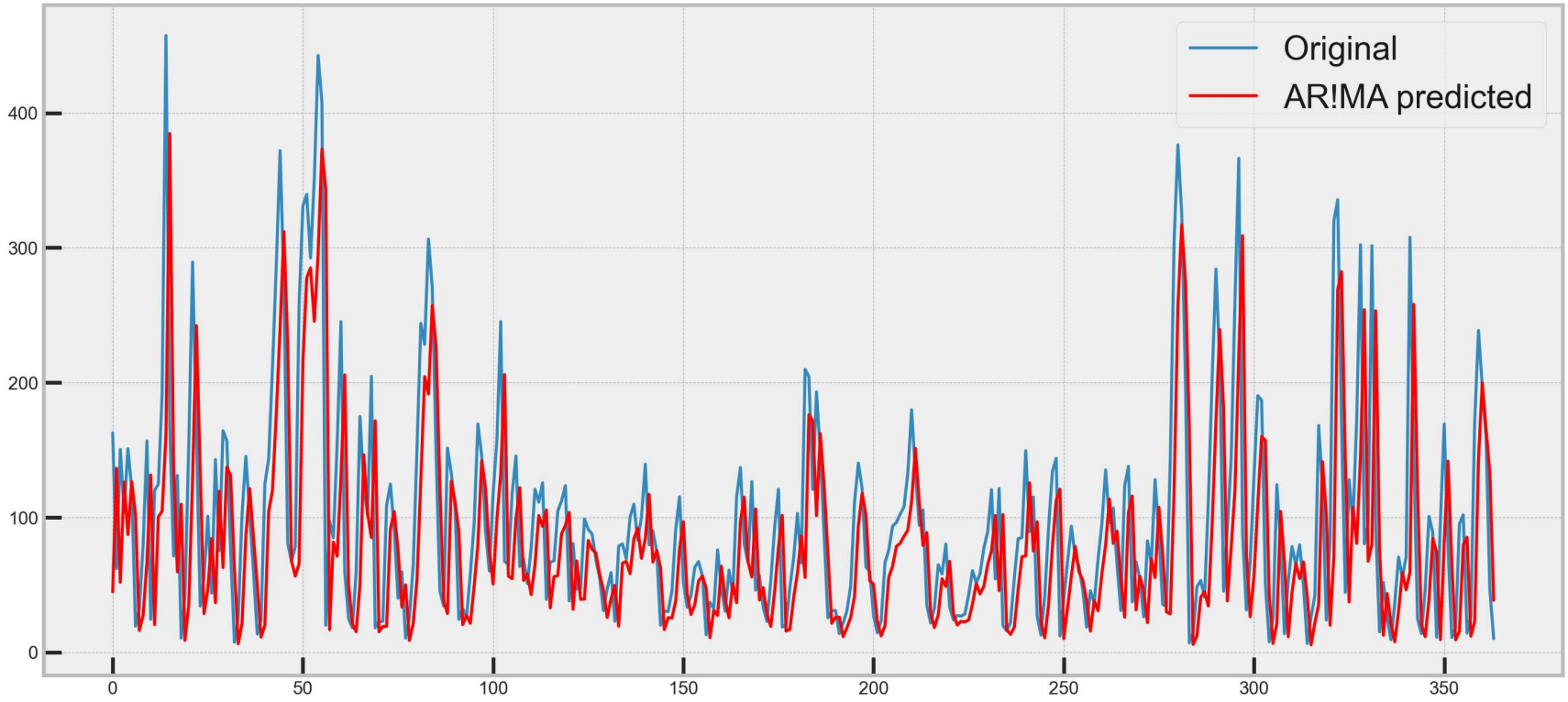
In [47]:
```
# Walk throught the test data, training and predicting 1 day ahead for all the test data
index = len(df_training)
yhat = list()
for t in tqdm(range(len(df_test.pollution_today))):
    temp_train = air_poll[:len(df_training)+t]
```

```
    model = SARIMAX(temp_train.pollution_today, order=(1,0,0))
    model_fit = model.fit(disp=False)
    predictions = model_fit.predict(
        start=len(temp_train), end=len(temp_train), dynamic=False)
    yhat = yhat + [predictions]

yhat = pd.concat(yhat)
# resultsDict['ARMA'] = evaluate(df_test.pollution_today, yhat.values)
predictionsDict['ARIMA'] = yhat.values
```

```
100%|██████████| 364/364 [00:17<00:00, 21.21it/s]
```

In [48]:
```
plt.plot(df_test.pollution_today.values, label='Original')
plt.plot(yhat.values, color='red', label='ARIMA predicted')
plt.legend()
```

Out[48]:
```
<matplotlib.legend.Legend at 0x18d03791670>
```



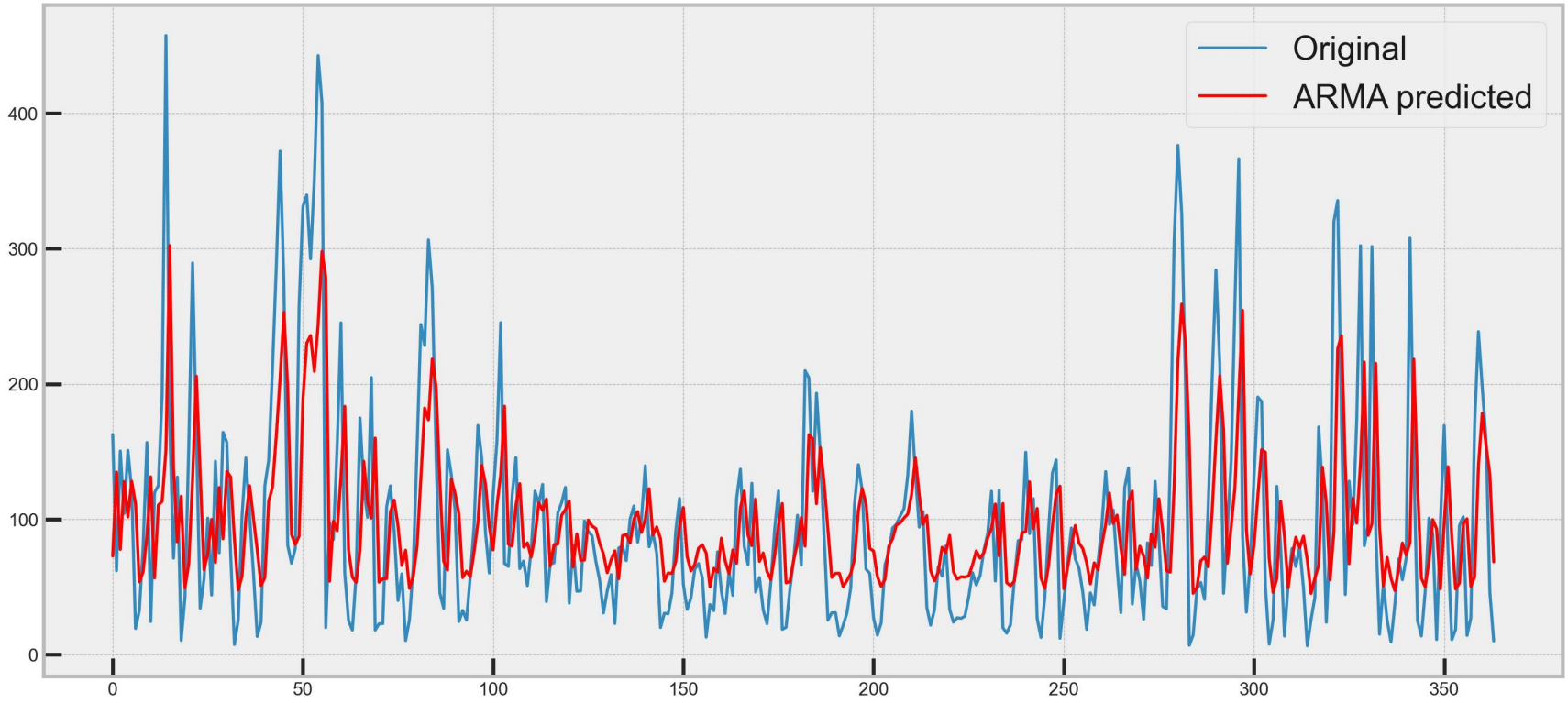
Autoregressive Moving Average (ARMA)

In [49]:
```
# Walk throught the test data, training and predicting 1 day ahead for all the test data
index = len(df_training)
yhat = list()
for t in tqdm(range(len(df_test.pollution_today))):
    temp_train = air_poll[:len(df_training)+t]
    model = SARIMAX(temp_train.pollution_today, order=(1,1,1))
    model_fit = model.fit(disp=False)
    predictions = model_fit.predict(
        start=len(temp_train), end=len(temp_train), dynamic=False)
    yhat = yhat + [predictions]

yhat = pd.concat(yhat)
# resultsDict['ARMA'] = evaluate(df_test.pollution_today, yhat.values)
predictionsDict['ARMA'] = yhat.values
```

```
100%|██████████| 364/364 [02:52<00:00,  2.11it/s]
```

In [50]:
```
plt.plot(df_test.pollution_today.values, label='Original')
plt.plot(yhat.values, color='red', label='ARMA predicted')
plt.legend()
```

Out[50]:
```
<matplotlib.legend.Legend at 0x18d0379eb50>
```



In [51]:
```
import pmdarima as pm
```

In [52]:
```
# building the model

autoModel = pm.auto_arima(df_training.pollution_today, trace=True,
                          error_action='ignore', suppress_warnings=True, seasonal=False)
autoModel.fit(df_training.pollution_today)
```

```
Performing stepwise search to minimize aic
 ARIMA(2,0,2)(0,0,0)[0]             : AIC=inf, Time=1.17 sec
 ARIMA(0,0,0)(0,0,0)[0]             : AIC=18232.724, Time=0.01 sec
 ARIMA(1,0,0)(0,0,0)[0]             : AIC=16461.265, Time=0.03 sec
 ARIMA(0,0,1)(0,0,0)[0]             : AIC=17191.945, Time=0.12 sec
 ARIMA(2,0,0)(0,0,0)[0]             : AIC=16462.630, Time=0.05 sec
 ARIMA(1,0,1)(0,0,0)[0]             : AIC=16462.086, Time=0.13 sec
 ARIMA(2,0,1)(0,0,0)[0]             : AIC=16455.385, Time=0.44 sec
 ARIMA(3,0,1)(0,0,0)[0]             : AIC=inf, Time=1.01 sec
 ARIMA(1,0,2)(0,0,0)[0]             : AIC=16266.588, Time=0.45 sec
 ARIMA(0,0,2)(0,0,0)[0]             : AIC=16869.411, Time=0.19 sec
 ARIMA(1,0,3)(0,0,0)[0]             : AIC=16217.321, Time=0.77 sec
 ARIMA(0,0,3)(0,0,0)[0]             : AIC=16711.266, Time=0.24 sec
 ARIMA(2,0,3)(0,0,0)[0]             : AIC=inf, Time=1.72 sec
 ARIMA(1,0,4)(0,0,0)[0]             : AIC=inf, Time=1.60 sec
 ARIMA(0,0,4)(0,0,0)[0]             : AIC=16638.862, Time=0.45 sec
 ARIMA(2,0,4)(0,0,0)[0]             : AIC=inf, Time=2.05 sec
 ARIMA(1,0,3)(0,0,0)[0] intercept   : AIC=16199.618, Time=0.76 sec
 ARIMA(0,0,3)(0,0,0)[0] intercept   : AIC=16195.678, Time=0.95 sec
 ARIMA(0,0,2)(0,0,0)[0] intercept   : AIC=16206.790, Time=0.68 sec
 ARIMA(0,0,4)(0,0,0)[0] intercept   : AIC=16197.407, Time=1.30 sec
 ARIMA(1,0,2)(0,0,0)[0] intercept   : AIC=16196.487, Time=1.27 sec
 ARIMA(1,0,4)(0,0,0)[0] intercept   : AIC=16199.461, Time=0.46 sec

Best model:  ARIMA(0,0,3)(0,0,0)[0] intercept
Total fit time: 15.879 seconds
```

Out[52]:

```
ARIMA
ARIMA(0,0,3)(0,0,0)[0] intercept
```

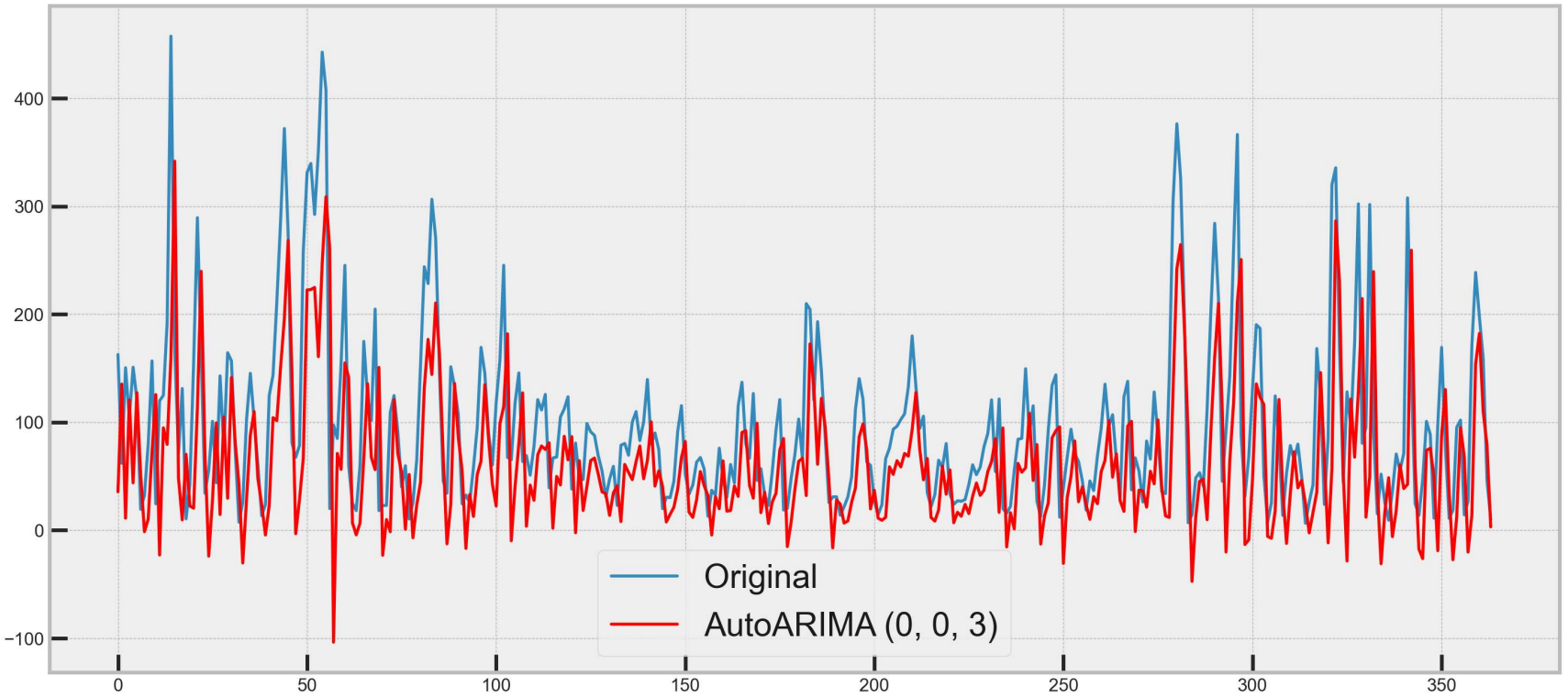
In [53]:

```
order = autoModel.order
yhat = list()
for t in tqdm(range(len(df_test.pollution_today))):
    temp_train = air_poll[:len(df_training)+t]
    model = SARIMAX(temp_train.pollution_today, order=order)
    model_fit = model.fit(disp=False)
    predictions = model_fit.predict(
        start=len(temp_train), end=len(temp_train), dynamic=False)
    yhat = yhat + [predictions]

yhat = pd.concat(yhat)
# resultsDict['AutoARIMA {0}'.format(order)] = evaluate(
#     df_test.pollution_today, yhat)
predictionsDict['AutoARIMA {0}'.format(order)] = yhat.values
```

```
100%|██████████| 364/364 [02:05<00:00,  2.90it/s]
```

In [54]:
```
plt.plot(df_test.pollution_today.values, label='Original')
plt.plot(yhat.values, color='red', label='AutoARIMA {0}'.format(order))
plt.legend()
```

Out[54]:
```
<matplotlib.legend.Legend at 0x18d0366ac40>
```



Seasonal Autoregressive Integrated Moving-Average (SARIMA)

- The extension of ARIMA known as Seasonal Autoregressive Integrated Moving Average, or Seasonal ARIMA, specifically supports univariate time series data with a seasonal component.
- It includes an additional parameter for the seasonality period as well as three new hyperparameters to determine the autoregression (AR), differencing (I), and moving average (MA) for the seasonal component of the series.
- Trend components:

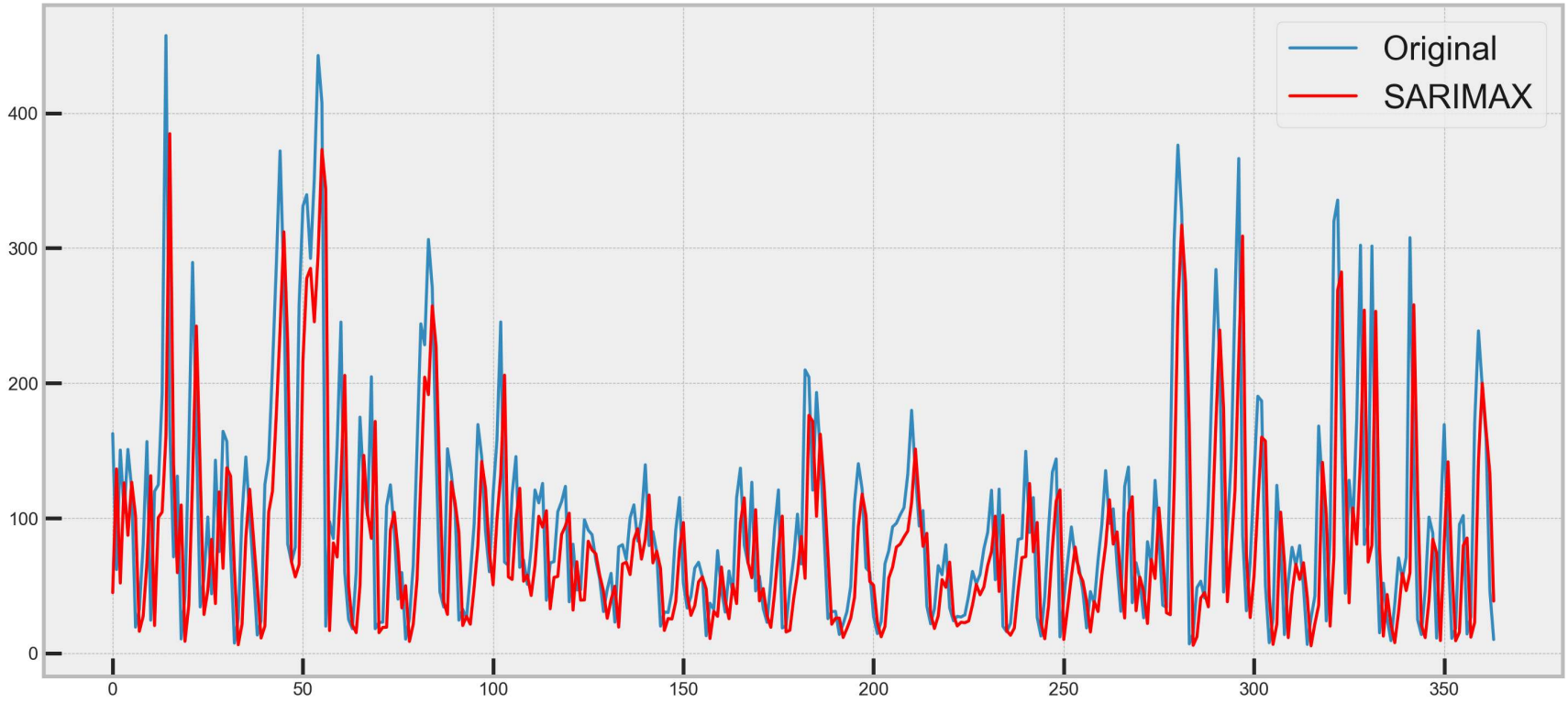
- There are three trend components that need setting up. In particular, they are the same as the ARIMA model.
  - p: Order of trend autoregression.
  - d: Order of trend difference.
  - q: Order of the trend moving average.
- Seasonal components
- Four seasonal components that are not a part of ARIMA must be configured; they are as follows:
- Autoregressive seasonal order.
  - D: Seasonal variation in sequence.
  - Moving average order by season.
  - m: A seasonal period's total number of time steps.

In [55]:
```
# SARIMA example

# Walk throught the test data, training and predicting 1 day ahead for all the test data
index = len(df_training)
yhat = list()
for t in tqdm(range(len(df_test.pollution_today))):
    temp_train = air_poll[:len(df_training)+t]
    model = SARIMAX(temp_train.pollution_today, order=(
        1, 0, 0), seasonal_order=(0, 0, 0, 3))
    model_fit = model.fit(disp=False)
    predictions = model_fit.predict(
        start=len(temp_train), end=len(temp_train), dynamic=False)
    yhat = yhat + [predictions]

yhat = pd.concat(yhat)
# resultsDict['SARIMAX'] = evaluate(df_test.pollution_today, yhat.values)
predictionsDict['SARIMAX'] = yhat.values
```

```
100%|██████████| 364/364 [00:16<00:00, 21.46it/s]
```

In [56]:
```
plt.plot(df_test.pollution_today.values, label='Original')
plt.plot(yhat.values, color='red', label='SARIMAX')
plt.legend()
```

Out[56]:
```
<matplotlib.legend.Legend at 0x18d001aa400>
```



In [57]:
```
# building the model
autoModel = pm.auto_arima(df_training.pollution_today, trace=True, error_action='ignore',
                          suppress_warnings=True, seasonal=True, m=6, stepwise=True)
autoModel.fit(df_training.pollution_today)
```

```
Performing stepwise search to minimize aic
 ARIMA(2,0,2)(1,0,1)[6] intercept   : AIC=16199.941, Time=3.27 sec
 ARIMA(0,0,0)(0,0,0)[6] intercept   : AIC=16788.406, Time=0.03 sec
 ARIMA(1,0,0)(1,0,0)[6] intercept   : AIC=16229.161, Time=0.28 sec
 ARIMA(0,0,1)(0,0,1)[6] intercept   : AIC=16265.917, Time=0.77 sec
 ARIMA(0,0,0)(0,0,0)[6]             : AIC=18232.724, Time=0.01 sec
 ARIMA(2,0,2)(0,0,1)[6] intercept   : AIC=16200.349, Time=0.72 sec
 ARIMA(2,0,2)(1,0,0)[6] intercept   : AIC=16200.355, Time=0.67 sec
 ARIMA(2,0,2)(2,0,1)[6] intercept   : AIC=16203.879, Time=2.18 sec
 ARIMA(2,0,2)(1,0,2)[6] intercept   : AIC=16203.861, Time=2.42 sec
 ARIMA(2,0,2)(0,0,0)[6] intercept   : AIC=16198.453, Time=0.36 sec
 ARIMA(1,0,2)(0,0,0)[6] intercept   : AIC=16196.487, Time=1.56 sec
 ARIMA(1,0,2)(1,0,0)[6] intercept   : AIC=16198.378, Time=2.12 sec
 ARIMA(1,0,2)(0,0,1)[6] intercept   : AIC=16198.371, Time=1.80 sec
 ARIMA(1,0,2)(1,0,1)[6] intercept   : AIC=16200.253, Time=1.34 sec
 ARIMA(0,0,2)(0,0,0)[6] intercept   : AIC=16206.790, Time=0.68 sec
 ARIMA(1,0,1)(0,0,0)[6] intercept   : AIC=16194.487, Time=0.54 sec
 ARIMA(1,0,1)(1,0,0)[6] intercept   : AIC=16196.379, Time=1.42 sec
 ARIMA(1,0,1)(0,0,1)[6] intercept   : AIC=16196.372, Time=0.92 sec
 ARIMA(1,0,1)(1,0,1)[6] intercept   : AIC=16198.261, Time=0.99 sec
 ARIMA(0,0,1)(0,0,0)[6] intercept   : AIC=16263.917, Time=0.34 sec
 ARIMA(1,0,0)(0,0,0)[6] intercept   : AIC=16227.307, Time=0.06 sec
 ARIMA(2,0,1)(0,0,0)[6] intercept   : AIC=16196.487, Time=1.00 sec
 ARIMA(2,0,0)(0,0,0)[6] intercept   : AIC=16196.154, Time=0.09 sec
 ARIMA(1,0,1)(0,0,0)[6]             : AIC=16462.086, Time=0.08 sec

Best model:  ARIMA(1,0,1)(0,0,0)[6] intercept
Total fit time: 23.679 seconds
```

Out[57]:

```
ARIMA
ARIMA(1,0,1)(0,0,0)[6] intercept
```

In [58]:

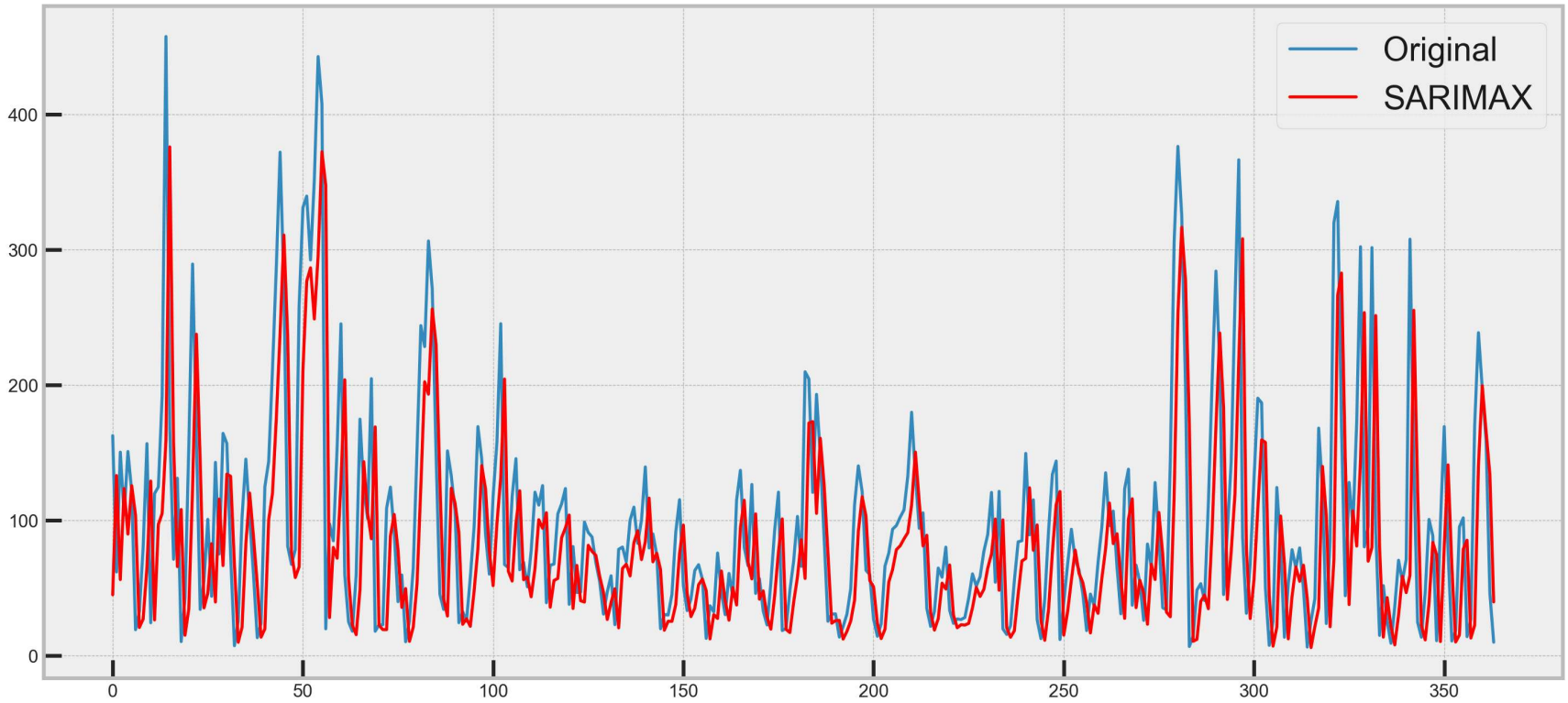
```
order = autoModel.order
seasonalOrder = autoModel.seasonal_order
yhat = list()
for t in tqdm(range(len(df_test.pollution_today))):
    temp_train = air_poll[:len(df_training)+t]
    model = SARIMAX(temp_train.pollution_today, order=order,
                    seasonal_order=seasonalOrder)
    model_fit = model.fit(disp=False)
    predictions = model_fit.predict(
        start=len(temp_train), end=len(temp_train), dynamic=False)
    yhat = yhat + [predictions]
```

```
yhat = pd.concat(yhat)

predictionsDict['AutoSARIMAX {0},{1}'.format(
    order, seasonalOrder)] = yhat.values
```

```
100%|██████████| 364/364 [00:38<00:00,  9.52it/s]
```

In [59]:
```
plt.plot(df_test.pollution_today.values, label='Original')
plt.plot(yhat.values, color='red', label='SARIMAX')
plt.legend()
```

Out[59]:
```
<matplotlib.legend.Legend at 0x18d000faf10>
```



Preprocessin for ML and DL models

In [62]:
```
# ADD time features to our model
def create_time_features(df, target=None):
    """
    Creates time series features from datetime index
```

```
    """
    df['date'] = df.index
    df['hour'] = df['date'].dt.hour
    df['dayofweek'] = df['date'].dt.dayofweek
    df['quarter'] = df['date'].dt.quarter
    df['month'] = df['date'].dt.month
    df['year'] = df['date'].dt.year
    df['dayofyear'] = df['date'].dt.dayofyear
    df['sin_day'] = np.sin(df['dayofyear'])
    df['cos_day'] = np.cos(df['dayofyear'])
    df['dayofmonth'] = df['date'].dt.day
    df['weekofyear'] = df['date'].dt.weekofyear
    X = df.drop(['date'], axis=1)
    if target:
        y = df[target]
        X = X.drop([target], axis=1)
        return X, y

    return X
```

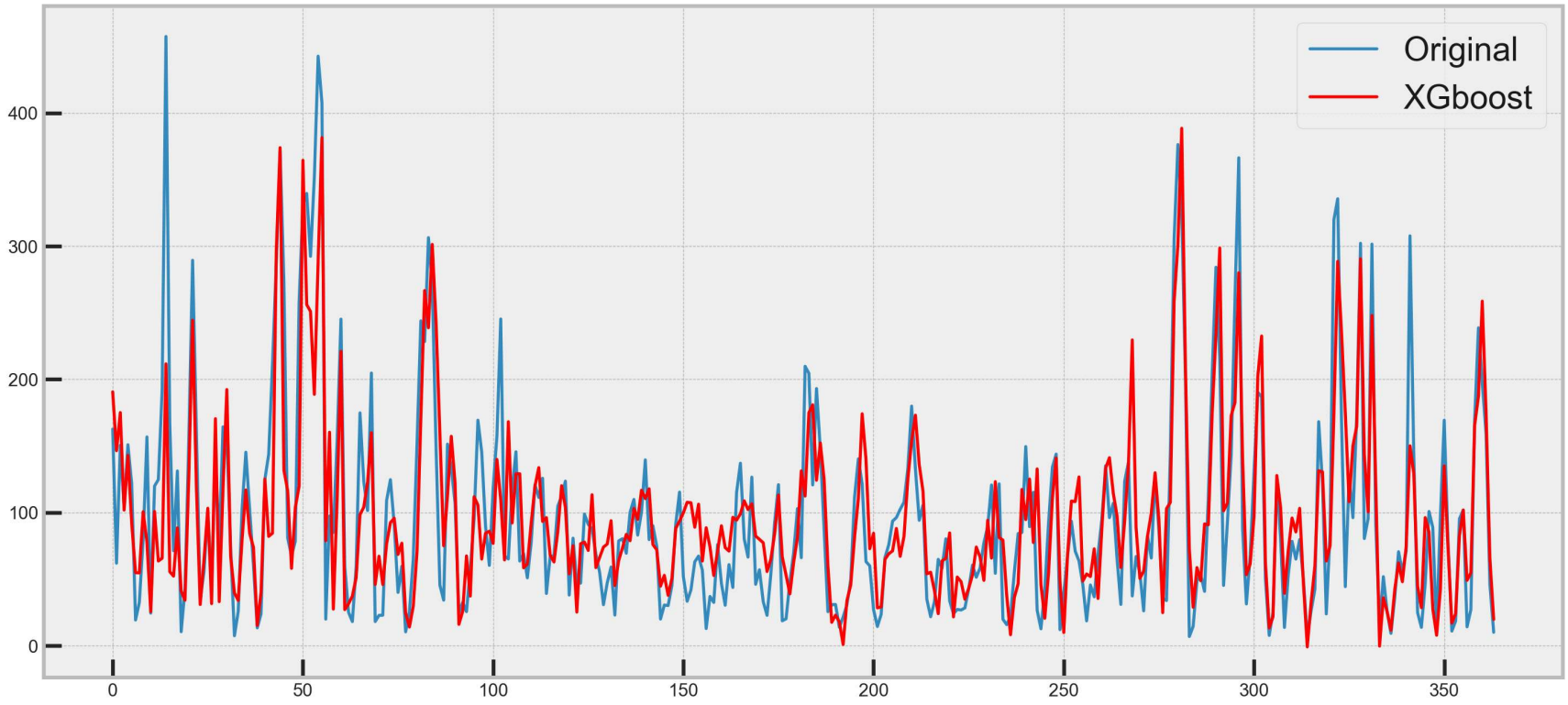
In [63]:
```
X_train_df, y_train = create_time_features(
    df_training, target='pollution_today')
X_test_df, y_test = create_time_features(df_test, target='pollution_today')
scaler = StandardScaler()
scaler.fit(X_train_df)  # No cheating, never scale on the training+test!
X_train = scaler.transform(X_train_df)
X_test = scaler.transform(X_test_df)

X_train_df = pd.DataFrame(X_train, columns=X_train_df.columns)
X_test_df = pd.DataFrame(X_test, columns=X_test_df.columns)
```

In [60]:
```
import xgboost as xgb
```

In [64]:
```
reg = xgb.XGBRegressor(objective='reg:squarederror', n_estimators=1000)
reg.fit(X_train, y_train,
        verbose=False)  # Change verbose to True if you want to see it train
yhat = reg.predict(X_test)
predictionsDict['XGBoost'] = yhat
```

In [65]:
```
plt.plot(df_test.pollution_today.values, label='Original')
plt.plot(yhat, color='red', label='XGboost')
plt.legend()
```

Out[65]:
```
<matplotlib.legend.Legend at 0x18d005a3040>
```

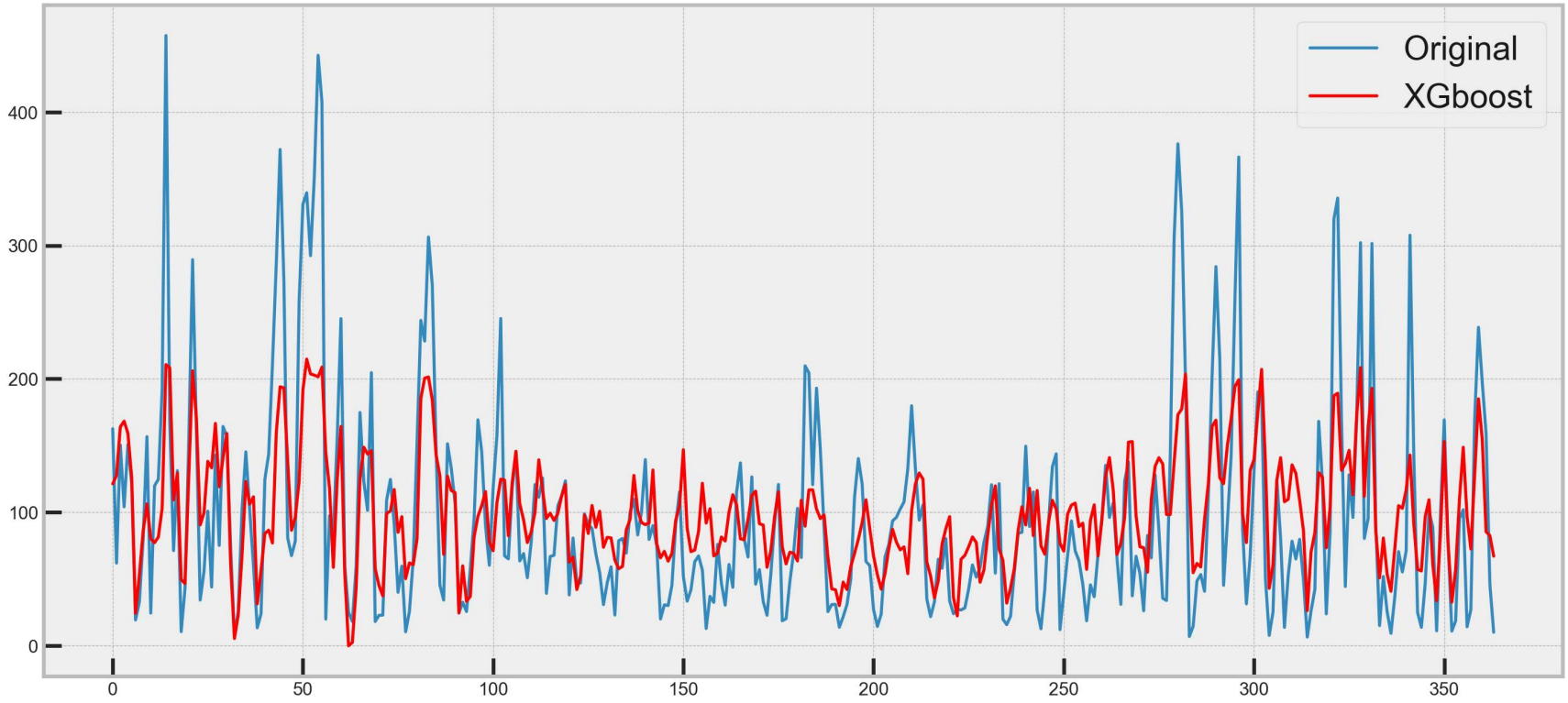


Support vector machines

In [66]:
```
reg = svm.SVR(kernel='rbf', C=100, gamma=0.1, epsilon=.1)
reg.fit(X_train, y_train)
yhat = reg.predict(X_test)
predictionsDict['SVM RBF'] = yhat
```

In [67]:
```
plt.plot(df_test.pollution_today.values, label='Original')
plt.plot(yhat, color='red', label='XGboost')
plt.legend()
```

Out[67]:
```
<matplotlib.legend.Legend at 0x18d0063b760>
```

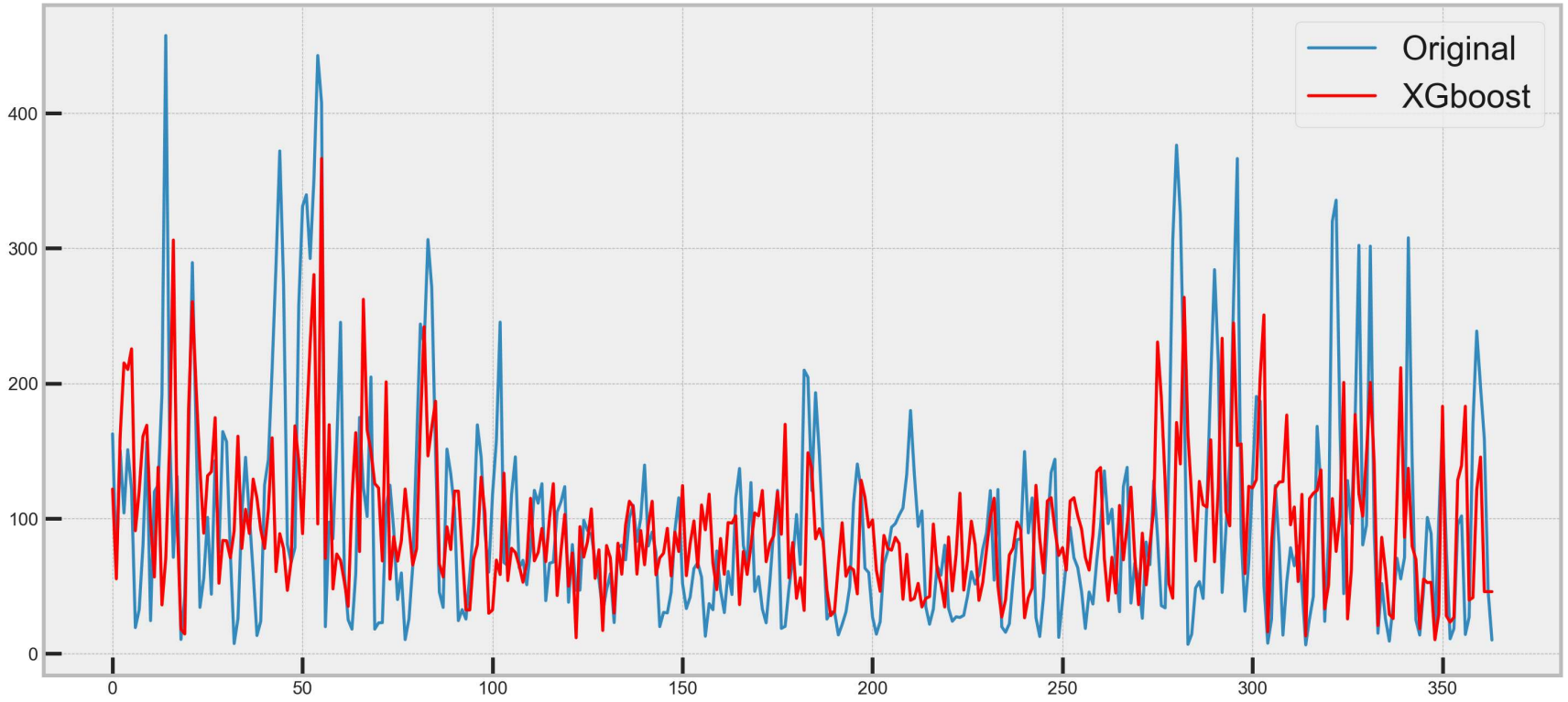


Nearest neighbors

In [68]:
```
reg = KNeighborsRegressor(n_neighbors=2)
reg.fit(X_train, y_train)
yhatn = reg.predict(X_test)
# resultsDict['Kneighbors'] = evaluate(df_test.pollution_today, yhat)
predictionsDict['Kneighbors'] = yhatn
```

In [69]:
```
plt.plot(df_test.pollution_today.values, label='Original')
plt.plot(yhatn, color='red', label='XGboost')
plt.legend()
```

Out[69]:
```
<matplotlib.legend.Legend at 0x18d00730c40>
```



Tensorlfow LSTM

In [70]:
```
# For our dl model we will create windows of data that will be feeded into the datasets, for each timestemp T we will a
BATCH_SIZE = 64
BUFFER_SIZE = 100
WINDOW_LENGTH = 24


def window_data(X, Y, window=7):
    '''
    The dataset length will be reduced to guarante all samples have the window, so new length will be len(dataset)-wind
    '''
    x = []
    y = []
    for i in range(window-1, len(X)):
        x.append(X[i-window+1:i+1])
        y.append(Y[i])
    return np.array(x), np.array(y)
```

```
# Since we are doing sliding, we need to join the datasets again of train and test
X_w = np.concatenate((X_train, X_test))
y_w = np.concatenate((y_train, y_test))

X_w, y_w = window_data(X_w, y_w, window=WINDOW_LENGTH)
X_train_w = X_w[:-len(X_test)]
y_train_w = y_w[:-len(X_test)]
X_test_w = X_w[-len(X_test):]
y_test_w = y_w[-len(X_test):]

# Check we will have same test set as in the previous models, make sure we didnt screw up on the windowing
print(f"Test set equal: {np.array_equal(y_test_w,y_test)}")

train_data = tf.data.Dataset.from_tensor_slices((X_train_w, y_train_w))
train_data = train_data.cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE).repeat()

val_data = tf.data.Dataset.from_tensor_slices((X_test_w, y_test_w))
val_data = val_data.batch(BATCH_SIZE).repeat()
```

```
Test set equal: True
```

In [71]:
```
dropout = 0.0
simple_lstm_model = tf.keras.models.Sequential([
    tf.keras.layers.LSTM(
        128, input_shape=X_train_w.shape[-2:], dropout=dropout),
    tf.keras.layers.Dense(128),
    tf.keras.layers.Dense(128),
    tf.keras.layers.Dense(1)
])

simple_lstm_model.compile(optimizer='rmsprop', loss='mae')

# logdir = "logs/scalars/" + datetime.now().strftime("%Y%m%d-%H%M%S") #Support for tensorboard tracking!
# tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=logdir)
```

In [72]:
```
EVALUATION_INTERVAL = 200
EPOCHS = 8

model_history = simple_lstm_model.fit(train_data, epochs=EPOCHS,
                                      steps_per_epoch=EVALUATION_INTERVAL,
                                      validation_data=val_data, validation_steps=50)  # ,callbacks=[tensorboard_callbac
```

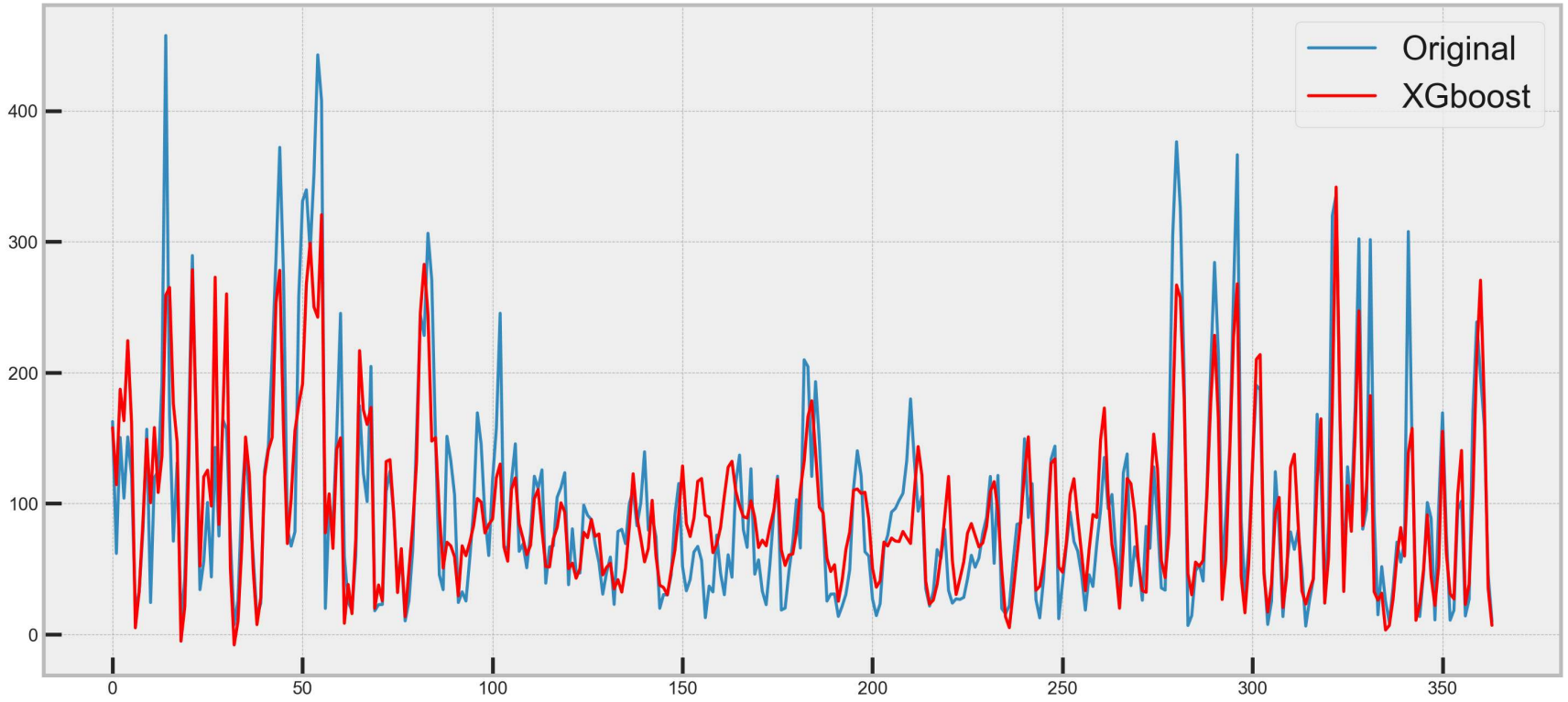
```
Epoch 1/8
200/200 [==============================] - 9s 23ms/step - loss: 52.0130 - val_loss: 43.8591
Epoch 2/8
200/200 [==============================] - 4s 19ms/step - loss: 35.3048 - val_loss: 33.6138
Epoch 3/8
200/200 [==============================] - 4s 19ms/step - loss: 29.2664 - val_loss: 31.1268
Epoch 4/8
200/200 [==============================] - 4s 19ms/step - loss: 26.2593 - val_loss: 29.1163
Epoch 5/8
200/200 [==============================] - 4s 19ms/step - loss: 23.9484 - val_loss: 30.0124
Epoch 6/8
200/200 [==============================] - 4s 20ms/step - loss: 21.5508 - val_loss: 33.1394
Epoch 7/8
200/200 [==============================] - 4s 19ms/step - loss: 19.0570 - val_loss: 30.5987
Epoch 8/8
200/200 [==============================] - 4s 19ms/step - loss: 16.9947 - val_loss: 30.7979
```

In [73]:
```
yhatf = simple_lstm_model.predict(X_test_w).reshape(1, -1)[0]
predictionsDict['Tensorflow simple LSTM'] = yhatf
```

```
12/12 [==============================] - 1s 6ms/step
```

In [74]:
```
plt.plot(df_test.pollution_today.values, label='Original')
plt.plot(yhatf, color='red', label='XGboost')
plt.legend()
```

Out[74]:
```
<matplotlib.legend.Legend at 0x18d040be310>
```



In [75]:
```
models = ['Tensorflow simple LSTM','XGBoost']
resis = pd.DataFrame(data={k: df_test.pollution_today.values -
                    v for k, v in predictionsDict.items()})[models]
corr = resis.corr()
print("Residuals correlation")
corr.style.background_gradient(cmap='coolwarm')
```

```
Residuals correlation
```

Out[75]:

	Tensorflow simple LSTM	XGBoost
Tensorflow simple LSTM	1.000000	0.567136
XGBoost	0.567136	1.000000

In [76]:
```
from bayes_opt import BayesianOptimization
```

Grid search - SVM

In [77]:
```
def rms(y_actual, y_predicted):
    return sqrt(mean_squared_error(y_actual, y_predicted))


my_scorer = make_scorer(rms, greater_is_better=False)
pbounds = {
    'n_estimators': (100, 10000),
    'max_depth': (3, 15),
    'min_samples_leaf': (1, 4),
    'min_samples_split': (2, 10),
}


def rf_hyper_param(n_estimators,
                   max_depth,
                   min_samples_leaf,
                   min_samples_split):

    max_depth = int(max_depth)
    n_estimators = int(n_estimators)

    clf = RandomForestRegressor(n_estimators=n_estimators,
                                max_depth=int(max_depth),
                                min_samples_leaf=int(min_samples_leaf),
                                min_samples_split=int(min_samples_split),
                                n_jobs=1)

    return -np.mean(cross_val_score(clf, X_train, y_train, cv=3))


optimizer = BayesianOptimization(
    f=rf_hyper_param,
    pbounds=pbounds,
    random_state=1,
)
```

Hyperparameter optimization

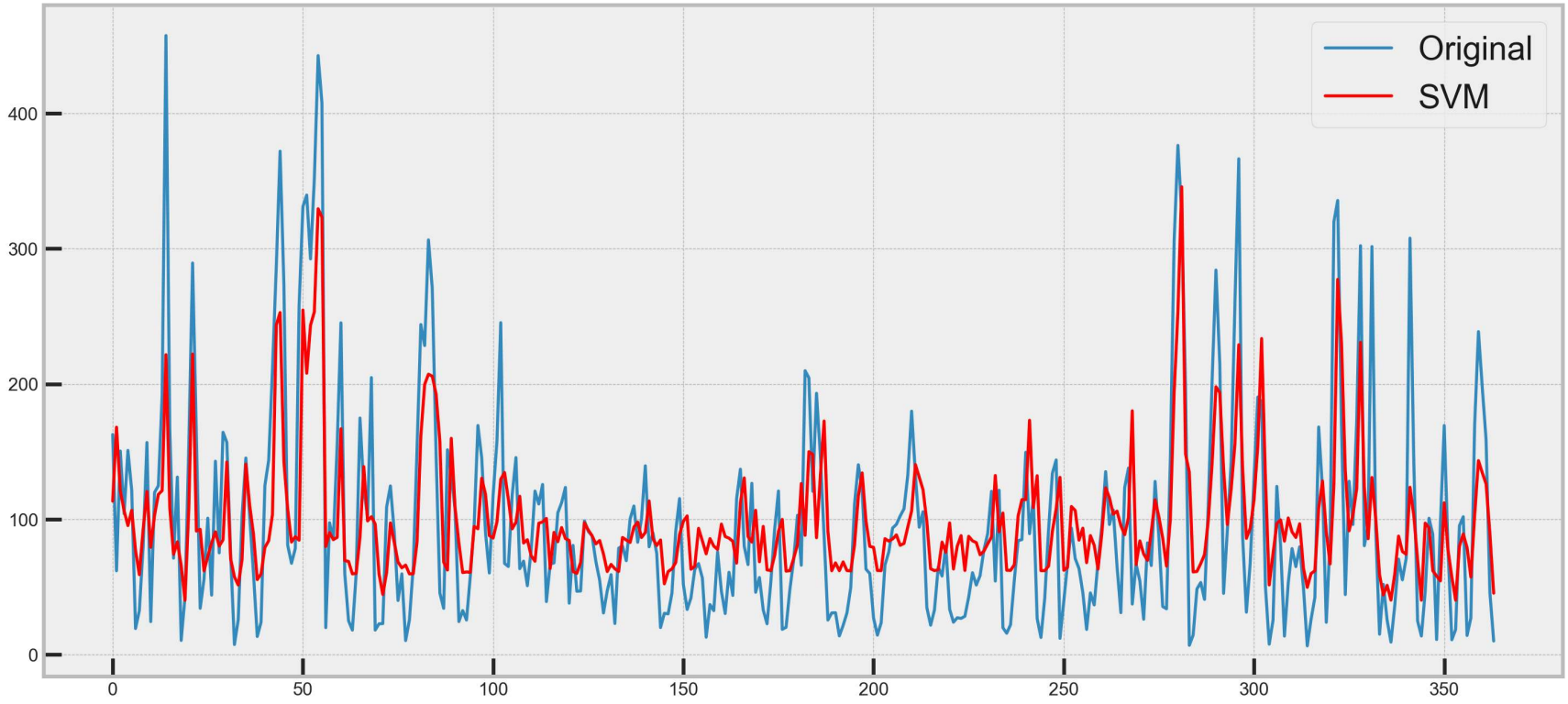
In [78]:
```
optimizer.maximize(
    init_points=3,
    n_iter=20,
    acq='ei'
)
```

iter	target	max_depth	min_sa...	min_sa...	n_esti...
1	-0.5472	8.004	3.161	2.001	3.093e+03
2	-0.5114	4.761	1.277	3.49	3.521e+03
3	-0.5465	7.761	2.616	5.354	6.884e+03
4	-0.5083	4.298	3.175	5.441	3.52e+03
5	-0.543	7.947	3.133	9.549	3.506e+03
6	-0.5459	7.243	1.491	8.858	3.524e+03
7	-0.4608	3.549	3.036	6.999	3.518e+03
8	-0.4619	3.891	2.847	7.469	3.52e+03
9	-0.5395	6.417	2.061	4.706	3.516e+03
10	-0.553	14.77	1.811	8.47	4.521e+03
11	-0.5283	5.947	2.751	8.017	3.517e+03
12	-0.5381	6.627	1.359	8.891	3.52e+03
13	-0.4623	3.286	2.762	8.621	3.52e+03
14	-0.4631	3.114	2.651	3.296	4.83e+03
15	-0.5464	7.162	2.005	4.747	4.829e+03
16	-0.4623	3.23	1.671	7.848	3.52e+03
17	-0.4627	3.721	2.176	8.349	3.518e+03
18	-0.5118	4.174	1.104	4.611	4.837e+03
19	-0.4604	3.058	3.419	6.25	4.825e+03
20	-0.4611	3.479	3.8	8.409	4.822e+03
21	-0.5092	4.461	2.674	7.546	4.822e+03
22	-0.5093	4.708	2.258	6.952	4.792e+03
23	-0.5548	12.82	1.491	2.12	1.694e+03

In [79]:
```
params = optimizer.max['params']

# Converting the max_depth and n_estimator values from float to int
params['max_depth'] = int(params['max_depth'])
params['n_estimators'] = int(params['n_estimators'])
params['min_samples_leaf'] = int(params['min_samples_leaf'])
params['min_samples_split'] = int(params['min_samples_split'])

# Initialize an XGBRegressor with the tuned parameters and fit the training data
tunned_rf = RandomForestRegressor(**params)
# Change verbose to True if you want to see it train
tunned_rf.fit(X_train, y_train)

yhats = tunned_rf.predict(X_test)
```

In [80]:
```
plt.plot(df_test.pollution_today.values, label='Original')
plt.plot(yhats, color='red', label='SVM')
plt.legend()
```

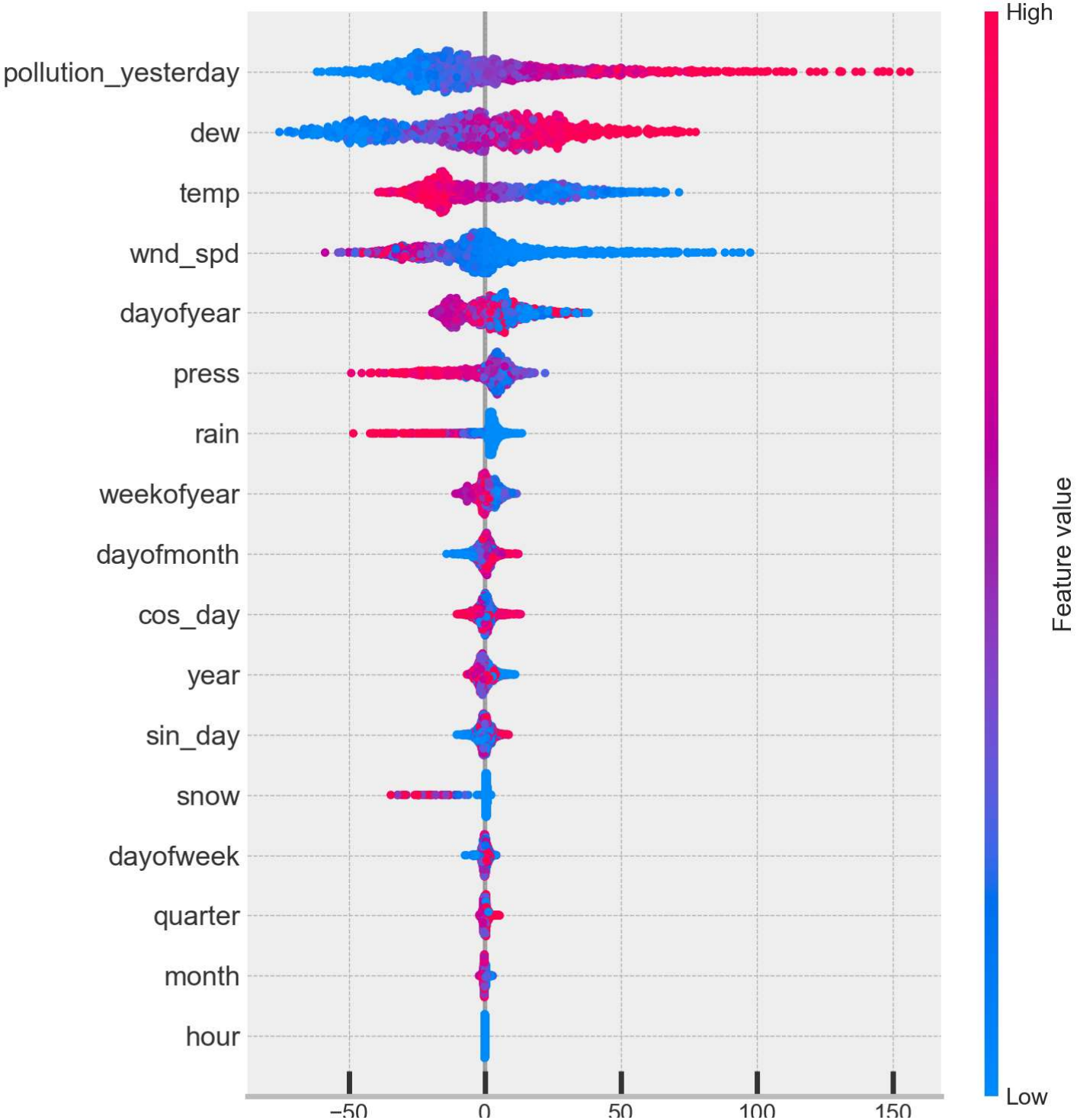
Out[80]:
```
<matplotlib.legend.Legend at 0x18d03fc2340>
```



To get the Feature importance

In [84]:
```
import lightgbm as lgb
lightGBM = lgb.LGBMRegressor()
lightGBM.fit(X_train, y_train)
yhat = lightGBM.predict(X_test)
# resultsDict['Lightgbm'] = evaluate(df_test.pollution_today, yhat)
predictionsDict['Lightgbm'] = yhat
```

In [85]:
```
import shap
import lightgbm as lgb
explainer = shap.TreeExplainer(lightGBM)
shap_values = explainer.shap_values(X_train_df)
shap.summary_plot(shap_values, X_train_df)
```


pollution_yesterday
dew
temp
wnd_spd
dayofyear
press
rain
weekofyear
dayofmonth
cos_day
year
sin_day
snow
dayofweek
quarter
month
hour
−50
0
50
100
150
High
Feature value
Low

In [ ]: