Group 4 - ADS 506

November 29, 2022

[630]:
```
#Import all required packages:
import warnings
warnings.filterwarnings("ignore")
import os
import itertools
import pandas as pd
import numpy as np
import datetime
import statsmodels.api as sm
from pandas.plotting import autocorrelation_plot
import statsmodels.formula.api as smf
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.seasonal import seasonal_decompose
from sklearn.metrics import mean_squared_error
from dateutil.parser import parse
from statsmodels.tsa.api import ExponentialSmoothing, SimpleExpSmoothing, Holt
import seaborn as sns
import matplotlib as mpl
import matplotlib.pyplot as plt
from prettytable import PrettyTable
%matplotlib inline
```

[631]:
```
df = pd.read_excel('/Users/JohnnyBlaze/Website Data Sets/Online Retail.xlsx',␣
↪parse_dates=[4])
```

[632]:
```
df.head()
```

[632]:
```
  InvoiceNo StockCode                          Description  Quantity  \
0    536365    85123A   WHITE HANGING HEART T-LIGHT HOLDER         6
1    536365     71053                  WHITE METAL LANTERN         6
2    536365    84406B       CREAM CUPID HEARTS COAT HANGER         8
3    536365    84029G  KNITTED UNION FLAG HOT WATER BOTTLE         6
4    536365    84029E       RED WOOLLY HOTTIE WHITE HEART.         6

          InvoiceDate  UnitPrice  CustomerID         Country
0 2010-12-01 08:26:00       2.55     17850.0  United Kingdom
1 2010-12-01 08:26:00       3.39     17850.0  United Kingdom
```

```
2 2010-12-01 08:26:00       2.75     17850.0  United Kingdom
3 2010-12-01 08:26:00       3.39     17850.0  United Kingdom
4 2010-12-01 08:26:00       3.39     17850.0  United Kingdom
```

[633]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 541909 entries, 0 to 541908
Data columns (total 8 columns):
 #   Column       Non-Null Count   Dtype
---  ------       --------------   -----
 0   InvoiceNo    541909 non-null  object
 1   StockCode    541909 non-null  object
 2   Description  540455 non-null  object
 3   Quantity     541909 non-null  int64
 4   InvoiceDate  541909 non-null  datetime64[ns]
 5   UnitPrice    541909 non-null  float64
 6   CustomerID   406829 non-null  float64
 7   Country      541909 non-null  object
dtypes: datetime64[ns](1), float64(2), int64(1), object(4)
memory usage: 33.1+ MB
```

[634]:

```
df.dropna()
```

[634]:

```
       InvoiceNo StockCode                          Description  Quantity  \
0         536365    85123A   WHITE HANGING HEART T-LIGHT HOLDER         6
1         536365     71053                  WHITE METAL LANTERN         6
2         536365    84406B       CREAM CUPID HEARTS COAT HANGER         8
3         536365    84029G  KNITTED UNION FLAG HOT WATER BOTTLE         6
4         536365    84029E       RED WOOLLY HOTTIE WHITE HEART.         6
…            …         …                                  …       …
541904    581587     22613          PACK OF 20 SPACEBOY NAPKINS        12
541905    581587     22899         CHILDREN'S APRON DOLLY GIRL          6
541906    581587     23254        CHILDRENS CUTLERY DOLLY GIRL          4
541907    581587     23255      CHILDRENS CUTLERY CIRCUS PARADE         4
541908    581587     22138        BAKING SET 9 PIECE RETROSPOT          3

               InvoiceDate  UnitPrice  CustomerID         Country
0      2010-12-01 08:26:00       2.55     17850.0  United Kingdom
1      2010-12-01 08:26:00       3.39     17850.0  United Kingdom
2      2010-12-01 08:26:00       2.75     17850.0  United Kingdom
3      2010-12-01 08:26:00       3.39     17850.0  United Kingdom
4      2010-12-01 08:26:00       3.39     17850.0  United Kingdom
…                      …        …          …             …
541904 2011-12-09 12:50:00       0.85     12680.0          France
541905 2011-12-09 12:50:00       2.10     12680.0          France
541906 2011-12-09 12:50:00       4.15     12680.0          France
541907 2011-12-09 12:50:00       4.15     12680.0          France
```

```
541908 2011-12-09 12:50:00       4.95     12680.0          France

[406829 rows x 8 columns]
```

[635]:
```python
df = df[df['Quantity'] > 0 ]
```

[636]:
```python
len(df)
```

[636]:
```
531285
```

[637]:
```python
df['Sales'] = (df['Quantity'] * df['UnitPrice'])
```

[638]:
```python
df.head()
```

[638]:
```
  InvoiceNo StockCode                          Description  Quantity  \
0    536365    85123A   WHITE HANGING HEART T-LIGHT HOLDER         6
1    536365     71053                  WHITE METAL LANTERN         6
2    536365    84406B       CREAM CUPID HEARTS COAT HANGER         8
3    536365    84029G  KNITTED UNION FLAG HOT WATER BOTTLE         6
4    536365    84029E       RED WOOLLY HOTTIE WHITE HEART.         6

          InvoiceDate  UnitPrice  CustomerID         Country  Sales
0 2010-12-01 08:26:00       2.55     17850.0  United Kingdom  15.30
1 2010-12-01 08:26:00       3.39     17850.0  United Kingdom  20.34
2 2010-12-01 08:26:00       2.75     17850.0  United Kingdom  22.00
3 2010-12-01 08:26:00       3.39     17850.0  United Kingdom  20.34
4 2010-12-01 08:26:00       3.39     17850.0  United Kingdom  20.34
```

[639]:
```python
clock = df[df['Description'].str.contains('CLOCK', na=False)]
```

[640]:
```python
len(clock)
```

[640]:
```
7180
```

[641]:
```python
clock['Category'] = 'Clock'
```

[642]:
```python
clock.drop(['InvoiceNo', 'StockCode', 'Description', 'CustomerID', 'Quantity',␣
↪'UnitPrice'], axis=1, inplace=True)
```

[643]:
```python
clock.head()
```

[643]:
```
            InvoiceDate         Country  Sales Category
26   2010-12-01 08:45:00          France   90.0    Clock
27   2010-12-01 08:45:00          France   90.0    Clock
28   2010-12-01 08:45:00          France   45.0    Clock
149  2010-12-01 09:45:00  United Kingdom   15.0    Clock
204  2010-12-01 10:03:00       Australia   17.0    Clock
```

[644]:
```python
clock['Country'].value_counts()
```

[644]:
```
United Kingdom          6415
France                   184
Germany                  141
EIRE                     127
Belgium                   63
Australia                 38
Switzerland               35
Denmark                   25
Spain                     24
Norway                    21
Iceland                   18
Channel Islands           13
Netherlands               13
Portugal                  11
Finland                   11
Singapore                  7
Cyprus                     7
Israel                     6
Canada                     5
Poland                     3
Greece                     2
Brazil                     2
Malta                      2
Italy                      2
RSA                        2
Hong Kong                  1
European Community         1
Unspecified                1
Name: Country, dtype: int64
```

[645]:
```python
clock = clock.loc[clock['Country'] == 'United Kingdom']
```

[646]:
```python
clock.drop('Country', axis=1, inplace=True)
```

[647]:
```python
len(clock)
```

[647]:
```
6415
```

[648]:
```python
clock.head()
```

[648]:
```
            InvoiceDate  Sales Category
149 2010-12-01 09:45:00   15.0    Clock
271 2010-12-01 10:47:00   15.0    Clock
272 2010-12-01 10:47:00   30.0    Clock
273 2010-12-01 10:47:00   30.0    Clock
274 2010-12-01 10:47:00   30.0    Clock
```

[649]:
```python
type(clock)
```

[649]: pandas.core.frame.DataFrame

[650]:
```python
# clock3 = pd.DataFrame(clock.groupby("InvoiceDate")['Sales'].sum().
↪reset_index())
# clock3.head()
```

[651]:
```python
# clock3.plot(figsize=(12, 4))
# plt.title('Clock Sales', fontweight='bold', size=20)
# plt.show()
```

[652]:
```python
# clock.resample('H', on='InvoiceDate').Sales.sum()
```

[653]:
```python
clock2 = clock.resample('D', on='InvoiceDate').Sales.sum().reset_index()
```

[654]:
```python
clock2 = pd.DataFrame(clock2)
```

[655]:
```python
clock2.head()
```

[655]:

	InvoiceDate	Sales
0	2010-12-01	568.40
1	2010-12-02	798.25
2	2010-12-03	587.62
3	2010-12-04	0.00
4	2010-12-05	596.00

[656]:
```python
clock2 = clock2[clock2['Sales'] > 1]
```

[657]:
```python
type(clock2)
```

[657]: pandas.core.frame.DataFrame

[658]:
```python
len(clock2)
```

[658]: 303

[659]:
```python
clock2 = clock2.rename(columns={'Sales': 'Clock_Sales'})
```

[660]:
```python
clock2.head()
```

[660]:

	InvoiceDate	Clock_Sales
0	2010-12-01	568.40
1	2010-12-02	798.25
2	2010-12-03	587.62
4	2010-12-05	596.00
5	2010-12-06	475.97

[661]:
```
clock2.tail()
```

[661]:

	InvoiceDate	Clock_Sales
369	2011-12-05	1246.29
370	2011-12-06	1358.44
371	2011-12-07	2496.14
372	2011-12-08	1195.32
373	2011-12-09	421.82

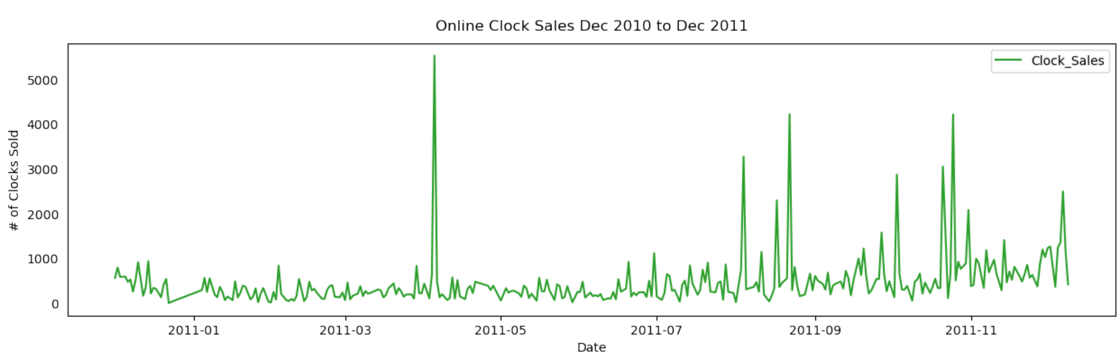
[662]:
```
clock2['InvoiceDate'].min()
```

[662]: Timestamp('2010-12-01 00:00:00')

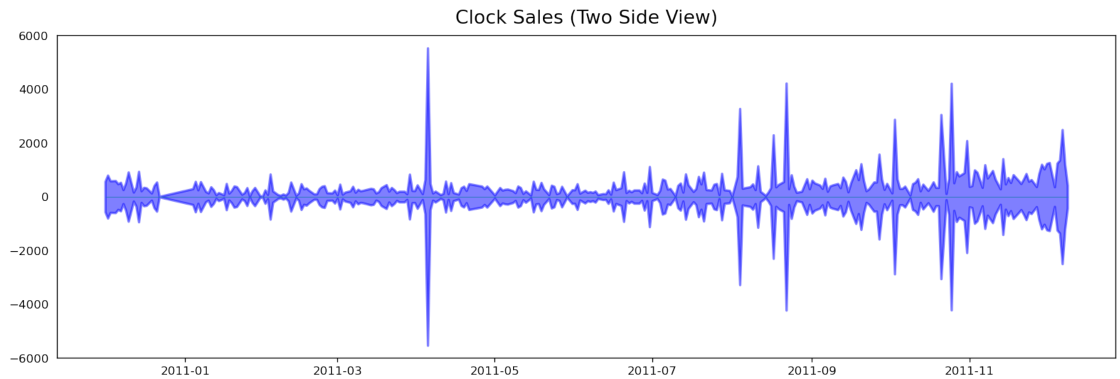
[663]:
```
clock2.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 303 entries, 0 to 373
Data columns (total 2 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   InvoiceDate  303 non-null    datetime64[ns]
 1   Clock_Sales  303 non-null    float64
dtypes: datetime64[ns](1), float64(1)
memory usage: 7.1 KB
```

[664]:
```
def plot_df(clock2, x, y, title="", xlabel='Date', ylabel='# of Clocks Sold',␣
↪dpi=100):
    plt.figure(figsize=(15,4), dpi=dpi)
    plt.plot(x, y, color='tab:green')
    plt.gca().set(title=title, xlabel=xlabel, ylabel=ylabel)
    plt.legend(['Clock_Sales'])
    plt.show()

plot_df(df, x=clock2['InvoiceDate'], y=clock2['Clock_Sales'], title='Online␣
↪Clock Sales Dec 2010 to Dec 2011')
```

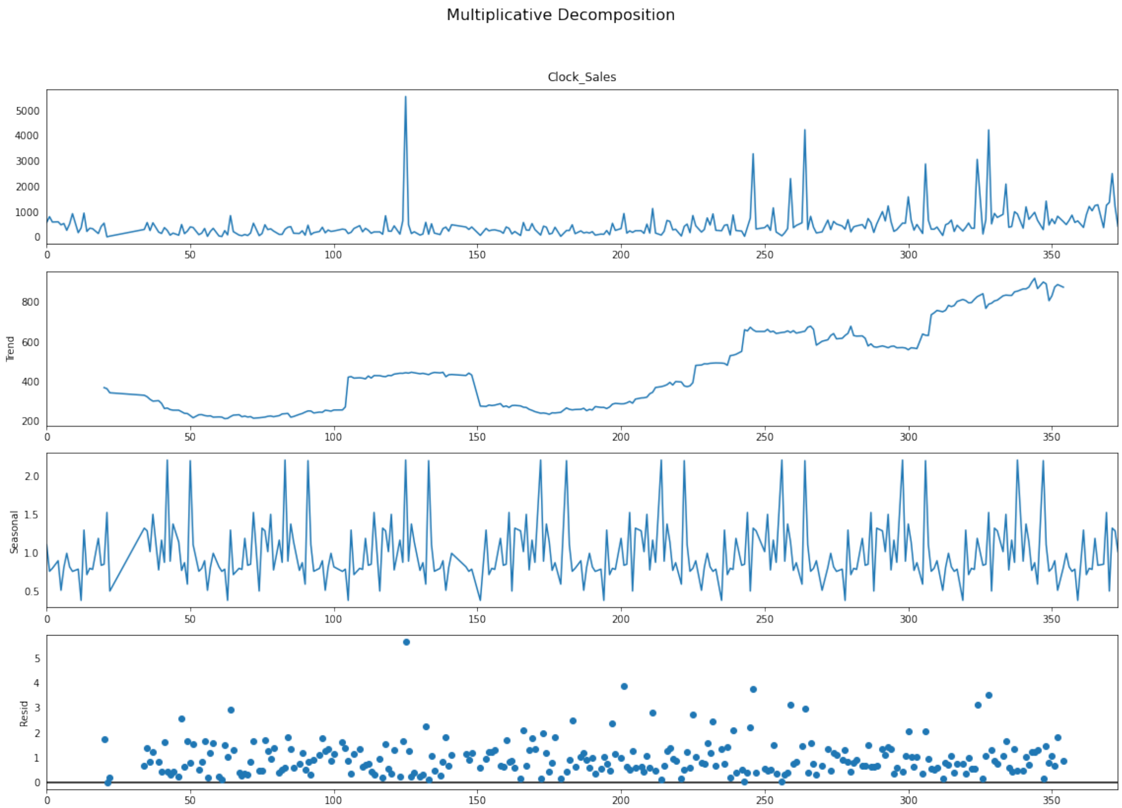


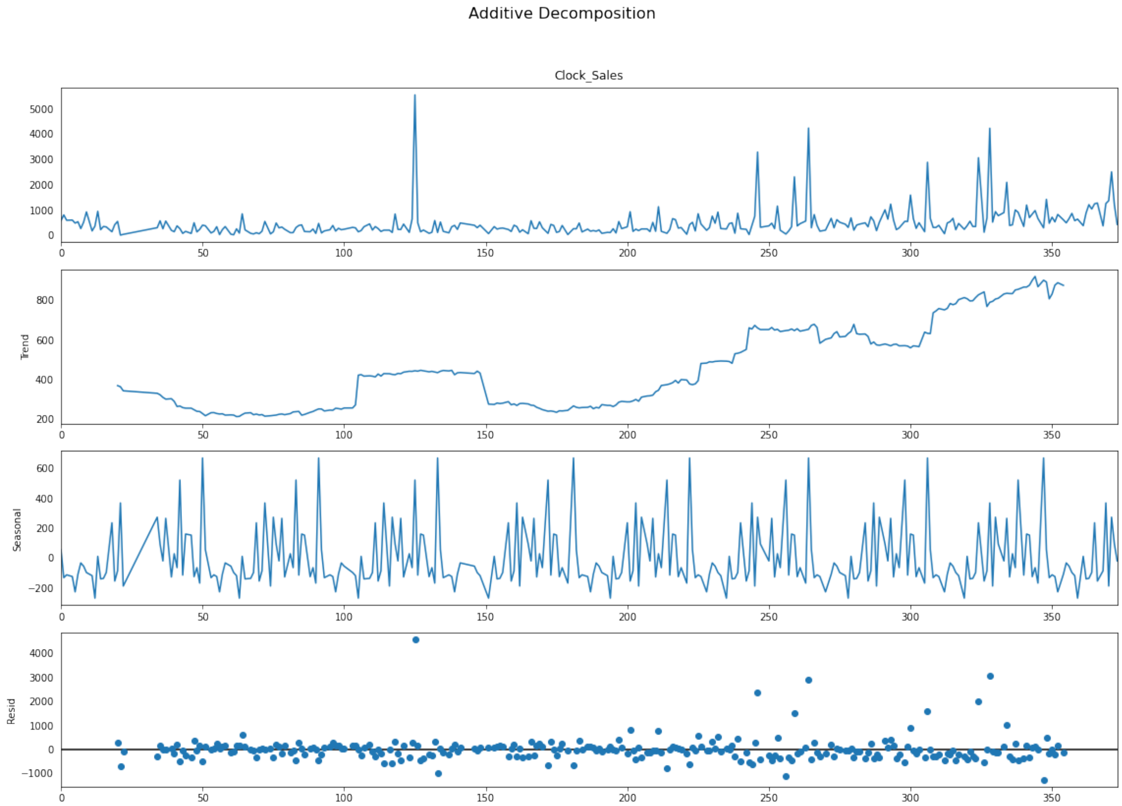
[665]:
```
x = clock2['InvoiceDate'].values
y1 = clock2['Clock_Sales'].values

# Plot
fig, ax = plt.subplots(1, 1, figsize=(16,5), dpi= 120)
plt.fill_between(x, y1=y1, y2=-y1, alpha=0.5, linewidth=2, color='blue')
plt.ylim(-6000, 6000)
plt.title('Clock Sales (Two Side View)', fontsize=16)
plt.hlines(y=0, xmin=np.min(clock2['InvoiceDate']), xmax=np.
↪max(clock2['InvoiceDate']), linewidth=.5)
plt.show()
```



[666]:
```
# Decomposition
# Decomposition of a time series can be performed by considering the series as␣
↪an additive or multiplicative combination of the base level, trend, seasonal␣
↪index and the residual term.

from statsmodels.tsa.seasonal import seasonal_decompose
from dateutil.parser import parse

# Multiplicative Decomposition
multiplicative_decomposition = seasonal_decompose(clock2['Clock_Sales'],␣
↪model='multiplicative', period=35)

# Additive Decomposition
additive_decomposition = seasonal_decompose(clock2['Clock_Sales'],␣
↪model='additive', period=35)

# Plot
plt.rcParams.update({'figure.figsize': (16,12)})
multiplicative_decomposition.plot().suptitle('Multiplicative Decomposition',␣
↪fontsize=16)
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
```

```
additive_decomposition.plot().suptitle('Additive Decomposition', fontsize=16)
plt.tight_layout(rect=[0, 0.03, 1, 0.95])

plt.show()
```

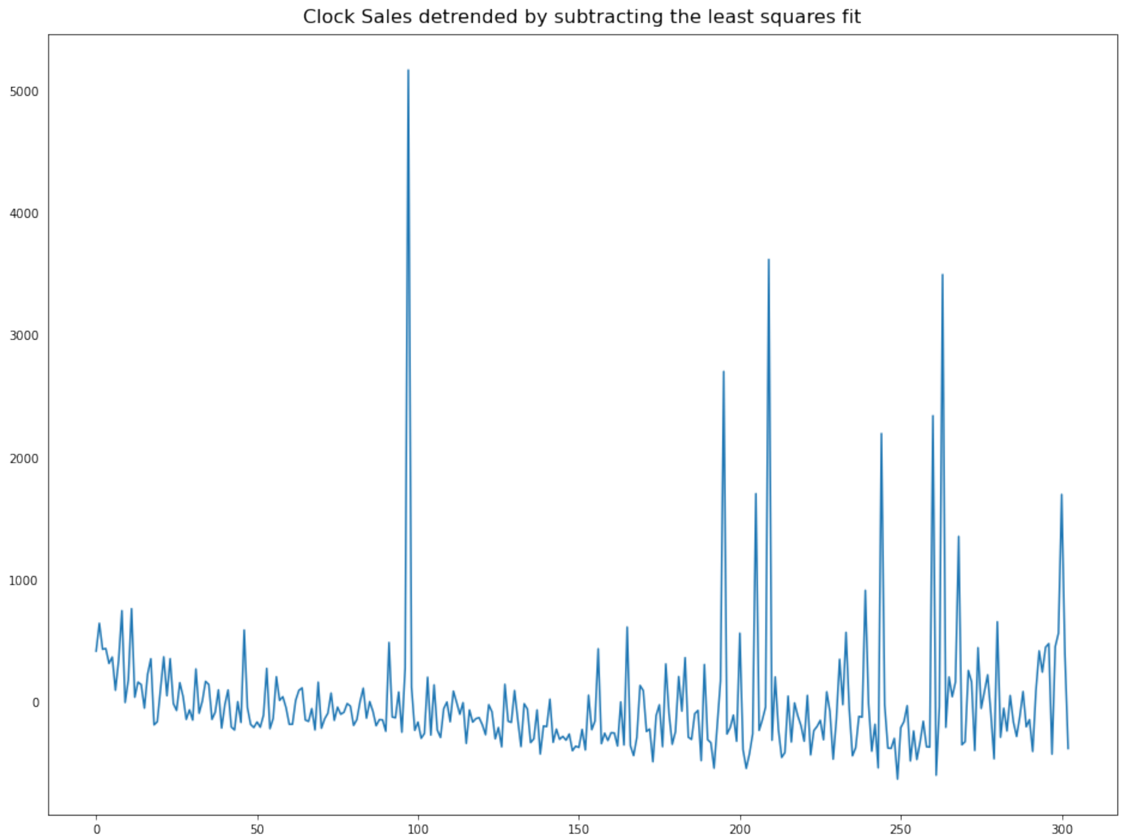




0.1 If we look at the residuals of the additive decomposition closely, it has some pattern left over.

0.2 The multiplicative decomposition, looks quite random which is good. So ideally, multiplicative decomposition should be preferred for this particular series.

[667]:
```
# Detrend

from scipy import signal
detrended = signal.detrend(clock2['Clock_Sales'].values)
plt.plot(detrended)
plt.title('Clock Sales detrended by subtracting the least squares fit',␣
    ↪fontsize=16)
plt.show()
```



0.3 The graph loooks the same as when it was first plotted suggesting no trend

[668]:
```python
# Dickey-Fuller Test
from statsmodels.tsa.stattools import adfuller

#perform augmented Dickey-Fuller test
X = clock2['Clock_Sales'].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: -4.091994
p-value: 0.000998
Critical Values:
        1%: -3.453
        5%: -2.871
        10%: -2.572
```

0.4 **Running the example prints the test statistic value of -4. The more negative this statistic, the more likely we are to reject the null hypothesis (we have a stationary dataset).**

0.5 **This suggests that we can reject the null hypothesis with a significance level of less than 1% (i.e. a low probability that the result is a statistical fluke).**

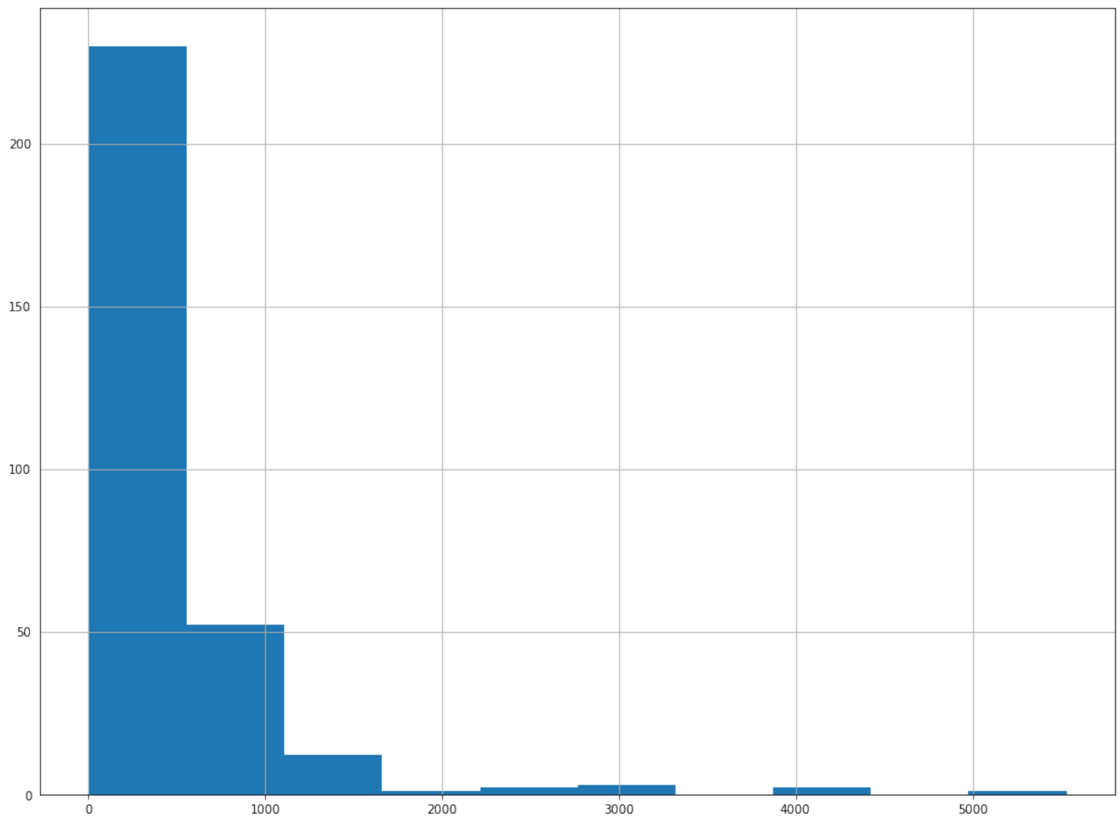
0.6 **Rejecting the null hypothesis means that the process has no unit root, and in turn that the time series is stationary or does not have time-dependent structure.**

0.7 **Stationarity of the time-series data: The stationarity of the data can be found using adfuller class of statsmodels.tsa.stattools module. The value of p-value is used to determine whether there is stationarity. If the value is less than 0.05, the stationarity exists.**

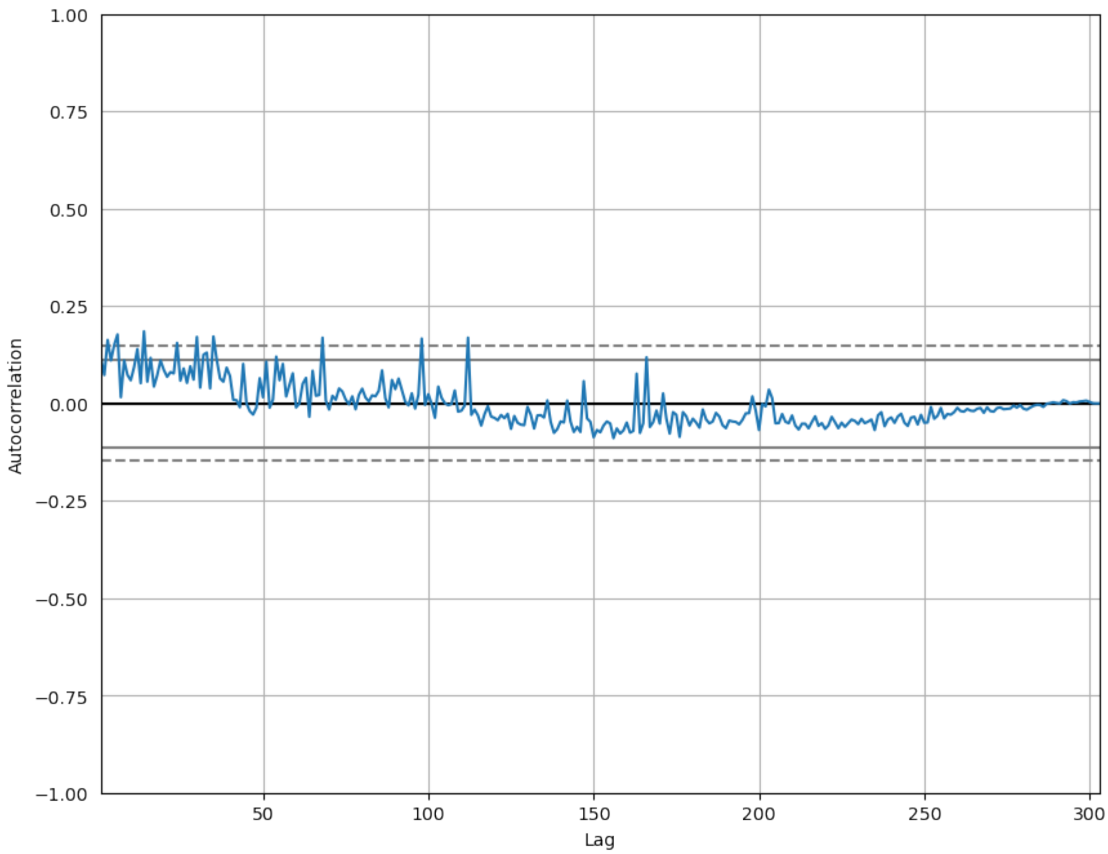
[669]:
```
# Dickey-Fuller Test

from statsmodels.tsa.stattools import adfuller

# Run the test

df_stationarityTest = adfuller(clock2['Clock_Sales'], autolag='AIC')

# Check the value of p-value

print("P-value: ", df_stationarityTest[1])
```

```
P-value:  0.0009984591482737
```

[670]:
```
clock2['Clock_Sales'].hist()
plt.show()
```

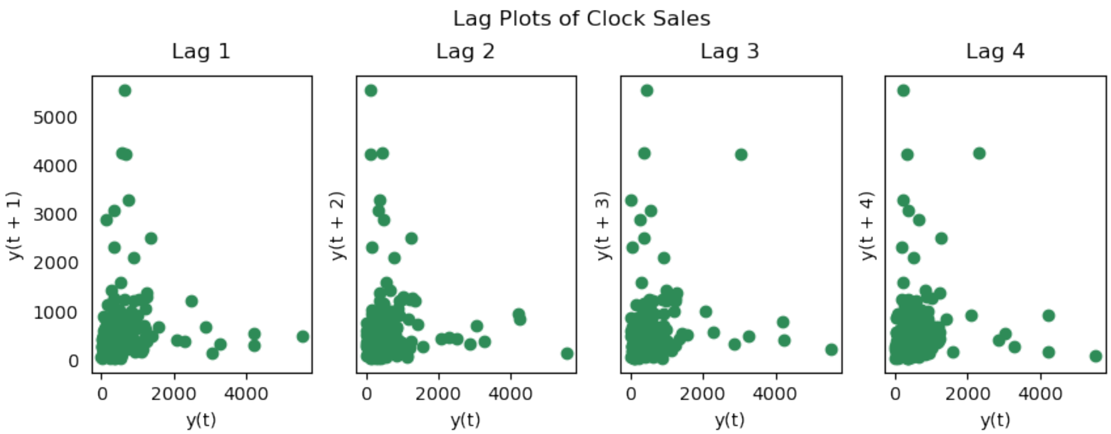


[671]:
```
# Test for seasonality
from pandas.plotting import autocorrelation_plot

# Draw Plot
plt.rcParams.update({'figure.figsize':(10,8), 'figure.dpi':100})
autocorrelation_plot(clock2['Clock_Sales'].tolist())
plt.show()
```



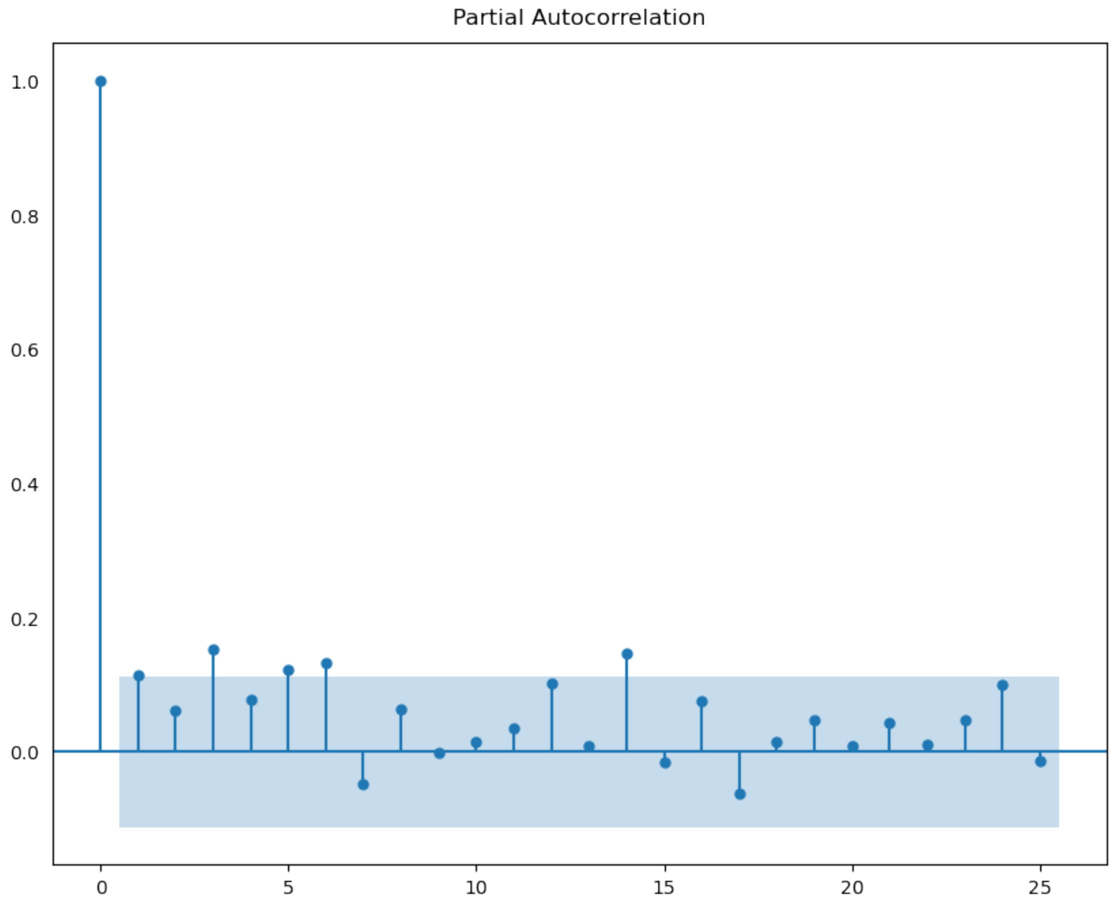
0.8 If partial autocorrelation values are close to 0, then values between observations and lagged observations are not correlated with one another. Inversely, partial autocorrelations with values close to 1 or -1 indicate that there exists strong positive or negative correlations between the lagged observations of the time series.

[672]:
```python
# Lag Plots

from pandas.plotting import lag_plot
plt.rcParams.update({'ytick.left' : False, 'axes.titlepad':10})

# Plot
fig, axes = plt.subplots(1, 4, figsize=(10,3), sharex=True, sharey=True,␣
↪dpi=100)
for i, ax in enumerate(axes.flatten()[:4]):
    lag_plot(clock2['Clock_Sales'], lag=i+1, ax=ax, c='seagreen')
    ax.set_title('Lag ' + str(i+1))

fig.suptitle('Lag Plots of Clock Sales', y=1.05)
plt.show()
```



0.9 A Lag plot is a scatter plot of a time series against a lag of itself. It is normally used to check for autocorrelation. If there is any pattern existing in the series, the series is autocorrelated. If there is no such pattern, the series is likely to be random white noise.

[673]:
```python
from statsmodels.graphics.tsaplots import plot_pacf

pacf = plot_pacf(clock2['Clock_Sales'], lags=25)
```



0.10 The above plot can be used to determine the order of AR model. You may note that a correlation value up to order 3 is high enough. Thus, we will train the AR model of order 3.

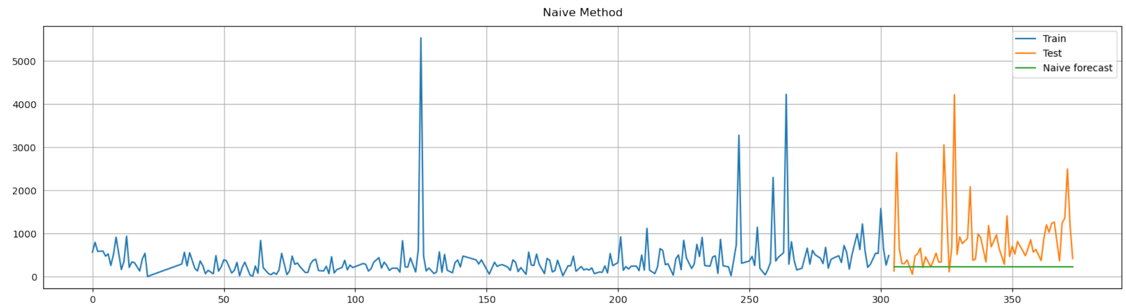
0.10.1 Naive Forecast Model

[674]:
```
# Split Train / Test

train_length = 243
train = clock2[0:train_length]
test = clock2[train_length:]
print(len(train))
print('')
print(len(test))
```

243

60

[675]:
```
# Naive Forecast

naive = test.copy()
naive['naive_forecast'] = train['Clock_Sales'][train_length-1]

plt.figure(figsize=(20,5))
plt.grid()
plt.plot(train['Clock_Sales'], label='Train')
plt.plot(test['Clock_Sales'], label='Test')
plt.plot(naive['naive_forecast'], label='Naive forecast')
plt.legend(loc='best')
plt.title('Naive Method')
plt.show()
```



[676]:
```
n_rmse = np.sqrt(mean_squared_error(test['Clock_Sales'],␣
↪naive['naive_forecast'])).round(2)
n_mape = np.round(np.mean(np.abs(test['Clock_Sales']-naive['naive_forecast'])/
↪test['Clock_Sales'])*100,2)

results = pd.DataFrame({'Method':['Naive method'], 'MAPE': [mape], 'RMSE':␣
↪[rmse]})
results = results[['Method', 'RMSE', 'MAPE']]
results
```

[676]:

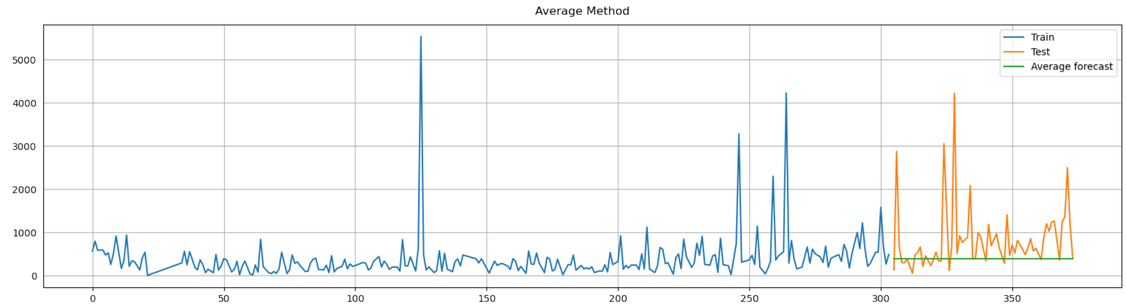
	Method	RMSE	MAPE
0	Naive method	868.59	58.44

Per the graph naive method is not suitable for data with high variability

0.10.2 Simple Average

[677]:
```
simple_average = test.copy()
simple_average['avg_forecast'] = train['Clock_Sales'].mean()

plt.figure(figsize=(20,5))
```

```
plt.grid()
plt.plot(train['Clock_Sales'], label='Train')
plt.plot(test['Clock_Sales'], label='Test')
plt.plot(simple_average['avg_forecast'], label='Average forecast')
plt.legend(loc='best')
plt.title('Average Method')
plt.show()
```



[678]:
```
sa_rmse = np.sqrt(mean_squared_error(test['Clock_Sales'],␣
 ↪simple_average['avg_forecast'])).round(2)
sa_mape = np.round(np.mean(np.
 ↪abs(test['Clock_Sales']-simple_average['avg_forecast'])/
 ↪test['Clock_Sales'])*100,2)

results = pd.DataFrame({'Method':['Average method'], 'MAPE': [mape], 'RMSE':␣
 ↪[rmse]})
results = results[['Method', 'RMSE', 'MAPE']]
results
```

[678]:

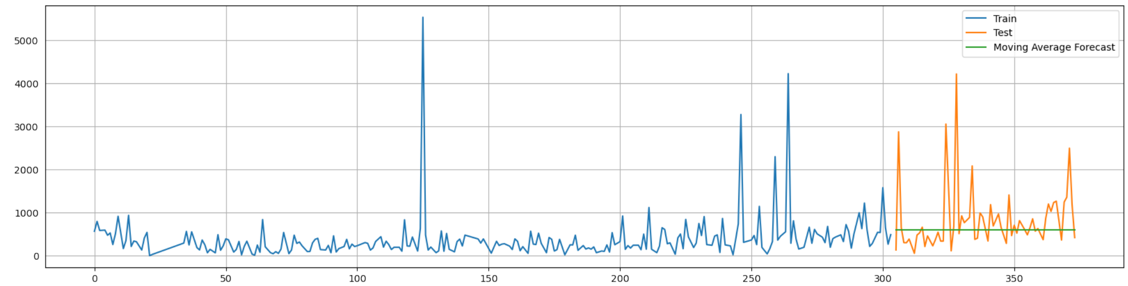
	Method	RMSE	MAPE
0	Average method	868.59	58.44

This model did improve our score, it seems the average of our data is pretty consistent.

0.10.3 Moving Average (60 Day)

[679]:
```
moving_avg = test.copy()
moving_avg['moving_avg_forecast'] = train['Clock_Sales'].rolling(60).mean().
 ↪iloc[-1]

plt.figure(figsize=(20,5))
plt.grid()
plt.plot(train['Clock_Sales'], label='Train')
plt.plot(test['Clock_Sales'], label='Test')
plt.plot(moving_avg['moving_avg_forecast'], label='Moving Average Forecast')
```

```
plt.legend(loc='best')
plt.show()
```



[680]:
```
ma_rmse = np.sqrt(mean_squared_error(test['Clock_Sales'],␣
↪moving_avg['moving_avg_forecast'])).round(2)
ma_mape = np.round(np.mean(np.
↪abs(test['Clock_Sales']-moving_avg['moving_avg_forecast'])/
↪test['Clock_Sales'])*100,2)

results = pd.DataFrame({'Method':['Moving Average method'], 'MAPE': [mape],␣
↪'RMSE': [rmse]})
results = results[['Method', 'RMSE', 'MAPE']]
results
```

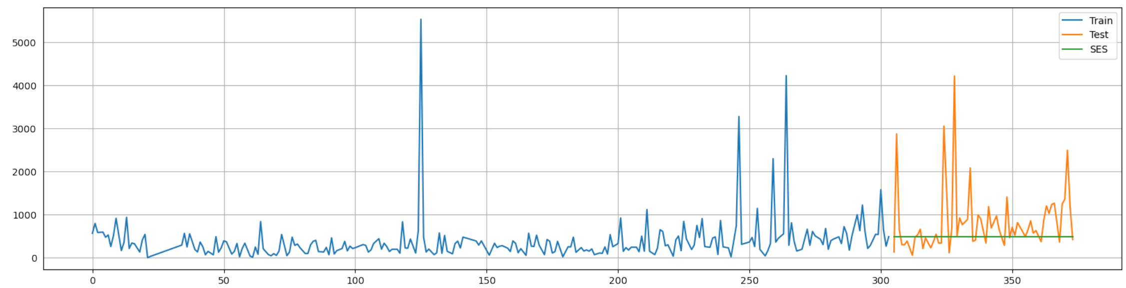
[680]:

	Method	RMSE	MAPE
0	Moving Average method	868.59	58.44

Interestingly enough this model did not improve our results after choosing the last 60 days. We could adjust the window and see if that improves our results.

0.10.4 Simple Exponential Smoothing

[681]:
```
ses = test.copy()
ses_fit = SimpleExpSmoothing(np.asarray(train['Clock_Sales'])).
↪fit(smoothing_level=0.6,optimized=False)
ses['SES'] = ses_fit.forecast(len(test))

plt.figure(figsize=(20,5))
plt.grid()
plt.plot(train['Clock_Sales'], label='Train')
plt.plot(test['Clock_Sales'], label='Test')
plt.plot(ses['SES'], label='SES')
plt.legend(loc='best')
plt.show()
```



[682]:
```
se_rmse = np.sqrt(mean_squared_error(test['Clock_Sales'], ses['SES'])).round(2)
se_mape = np.round(np.mean(np.abs(test['Clock_Sales']-ses['SES'])/
 ↪test['Clock_Sales'])*100,2)

results = pd.DataFrame({'Method':['Simple Exponential Smoothing method'],␣
 ↪'MAPE': [mape], 'RMSE': [rmse]})
results = results[['Method', 'RMSE', 'MAPE']]
results
```

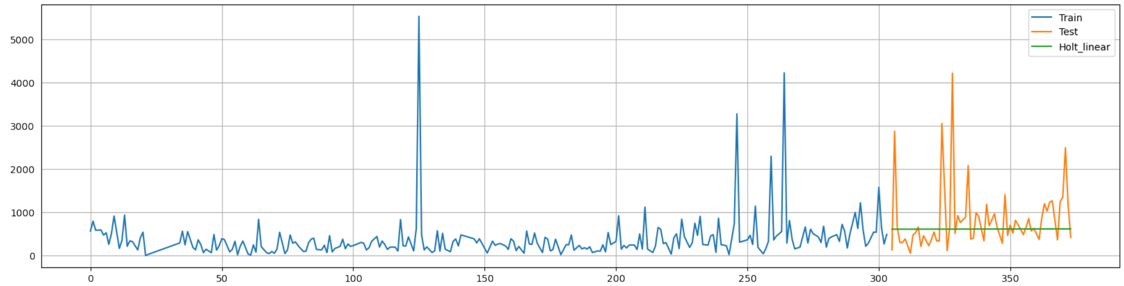
[682]:

	Method	RMSE	MAPE
0	Simple Exponential Smoothing method	868.59	58.44

So far the second best model after simple average. We can tune to alpha from 0.6 to another number to see if it helps improve the model.

0.10.5 Holt Method

[683]:
```
holt = test.copy()
holt_fit = Holt(np.asarray(train['Clock_Sales'])).fit(smoothing_level = 0.3,␣
 ↪smoothing_slope = 0.1)
holt['Holt_linear'] = holt_fit.forecast(len(test))

plt.figure(figsize=(20,5))
plt.grid()
plt.plot(train['Clock_Sales'], label='Train')
plt.plot(test['Clock_Sales'], label='Test')
plt.plot(holt['Holt_linear'], label='Holt_linear')
plt.legend(loc='best')
plt.show()
```



[684]:
```
hl_rmse = np.sqrt(mean_squared_error(test['Clock_Sales'], holt['Holt_linear'])).
↪round(2)
hl_mape = np.round(np.mean(np.abs(test['Clock_Sales']-holt['Holt_linear'])/
↪test['Clock_Sales'])*100,2)

results = pd.DataFrame({'Method':['Holt Linear method'], 'MAPE': [mape], 'RMSE':
↪ [rmse]})
results = results[['Method', 'RMSE', 'MAPE']]
results
```

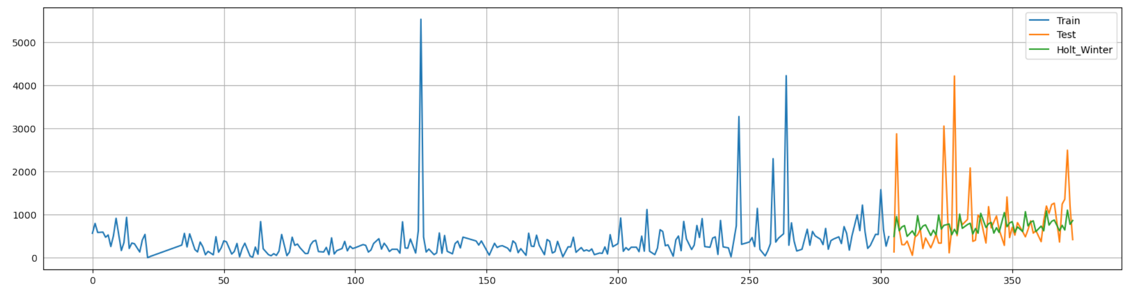
[684]:

	Method	RMSE	MAPE
0	Holt Linear method	868.59	58.44

Results were not very good on the first run, model can be tuned to see if there's improvement

0.10.6 Holt Winter's Method

[685]:
```
hw = test.copy()
hw_fit = ExponentialSmoothing(np.asarray(train['Clock_Sales'])␣
↪,seasonal_periods=7 ,trend='add', seasonal='add',).fit()
hw['Holt_Winter'] = hw_fit.forecast(len(test))

plt.figure(figsize=(20,5))
plt.grid()
plt.plot( train['Clock_Sales'], label='Train')
plt.plot(test['Clock_Sales'], label='Test')
plt.plot(hw['Holt_Winter'], label='Holt_Winter')
plt.legend(loc='best')
plt.show()
```



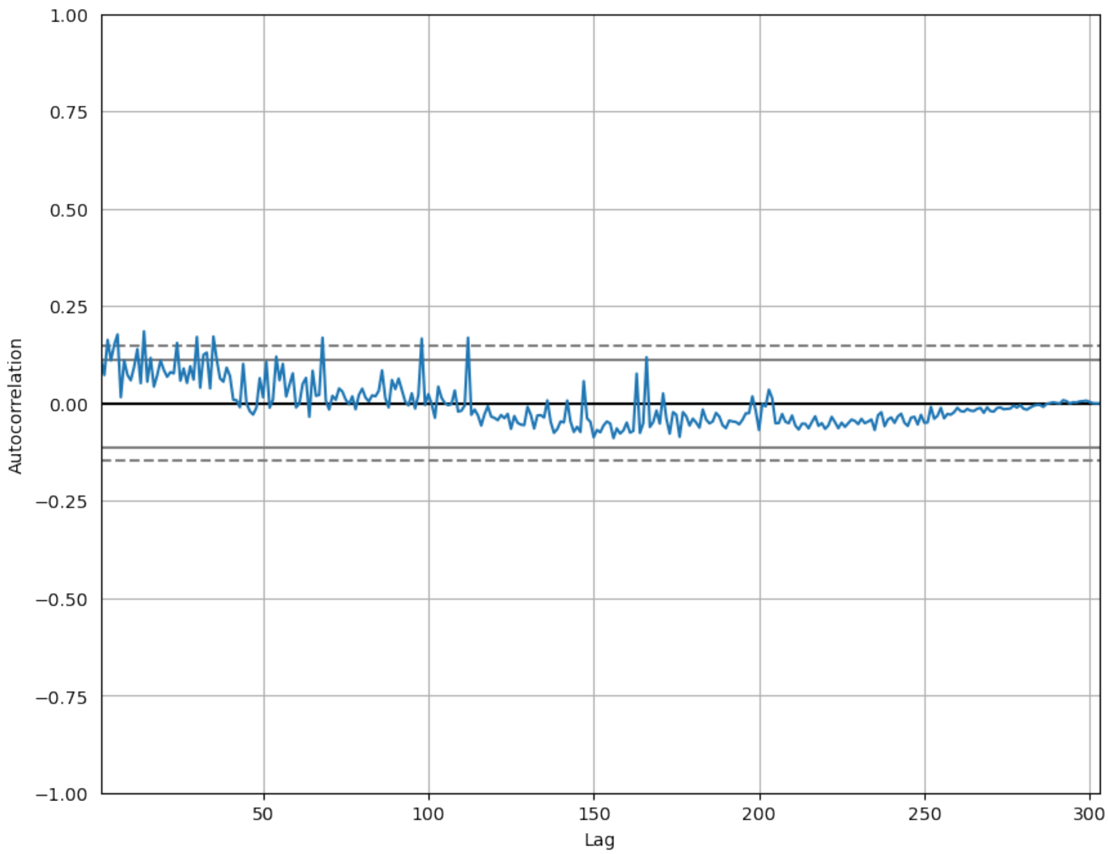
[686]:
```
hw_rmse = np.sqrt(mean_squared_error(test['Clock_Sales'], hw['Holt_Winter'])).
 ↪round(2)
hw_mape = np.round(np.mean(np.abs(test['Clock_Sales']-hw['Holt_Winter'])/
 ↪test['Clock_Sales'])*100,2)

results = pd.DataFrame({'Method':['Holt Winters method'], 'MAPE': [mape],␣
 ↪'RMSE': [rmse]})
results = results[['Method', 'RMSE', 'MAPE']]
results
```

[686]:

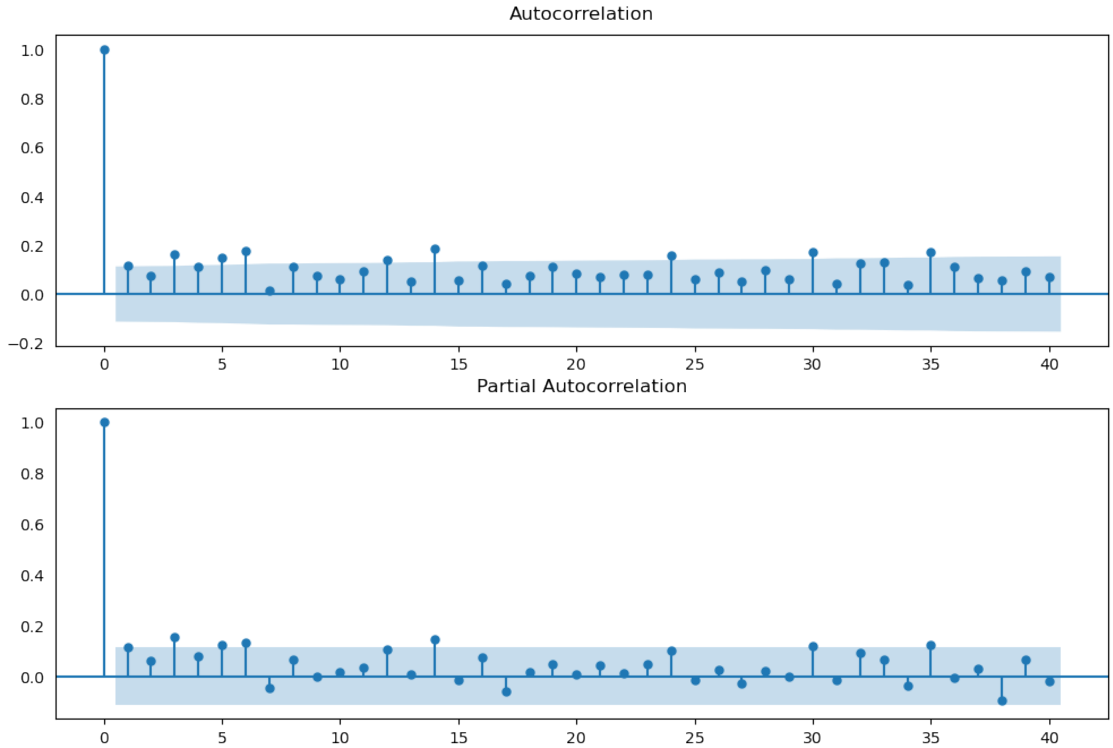
	Method	RMSE	MAPE
0	Holt Winters method	868.59	58.44

0.10.7 Arima Model

[687]:
```
autocorrelation_plot(clock2['Clock_Sales'])
plt.show()
```



[688]:
```
fig = plt.figure(figsize=(12,8))
ax1 = fig.add_subplot(211)
fig = sm.graphics.tsa.plot_acf(clock2['Clock_Sales'],lags=40,ax=ax1)
ax2 = fig.add_subplot(212)
fig = sm.graphics.tsa.plot_pacf(clock2['Clock_Sales'],lags=40,ax=ax2)
```

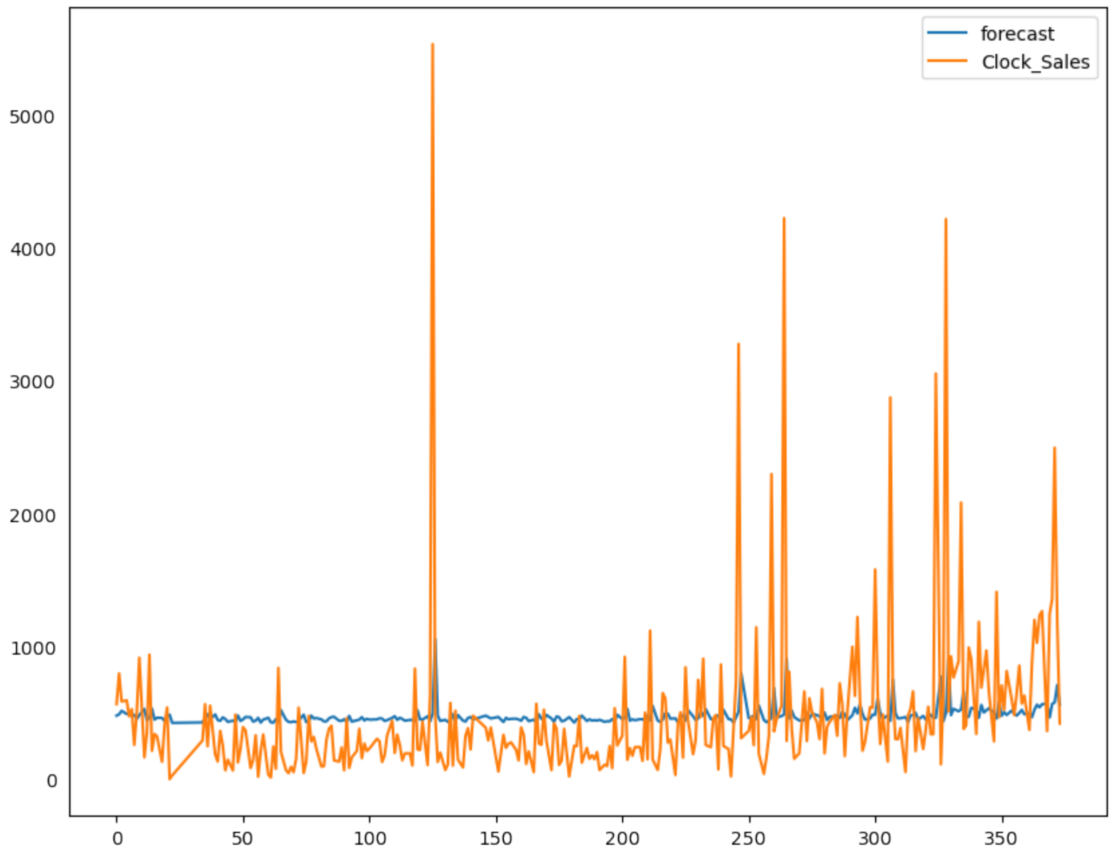


[689]:
```python
from statsmodels.tsa.arima_model import ARIMA

model = ARIMA(clock2['Clock_Sales'], order = (1,0,0))
model_fit = model.fit(disp=0)
model_fit.summary()
```

[689]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                              ARMA Model Results
==============================================================================
Dep. Variable:            Clock_Sales   No. Observations:                  303
Model:                     ARMA(1, 0)   Log Likelihood               -2368.473
Method:                       css-mle   S.D. of innovations            600.512
Date:                Tue, 29 Nov 2022   AIC                           4742.946
Time:                        21:22:27   BIC                           4754.087
Sample:                             0   HQIC                          4747.403

================================================================================
=====
                        coef    std err          z      P>|z|      [0.025
0.975]
--------------------------------------------------------------------------------
-----
```

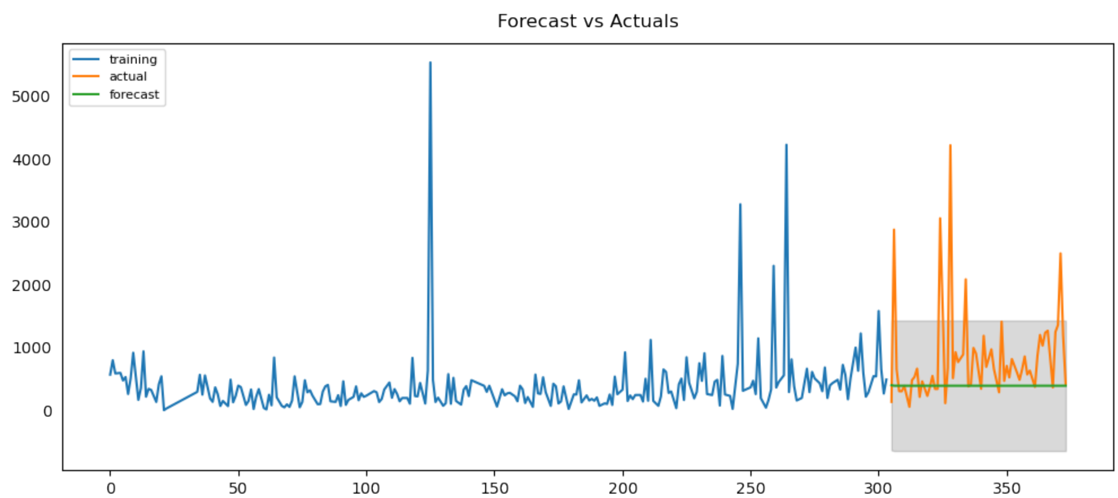
```
const                481.2924      38.920      12.366       0.000      405.011
557.574
ar.L1.Clock_Sales      0.1140       0.057       2.000       0.045        0.002
0.226
                                    Roots
=============================================================================
                  Real          Imaginary           Modulus         Frequency
-----------------------------------------------------------------------------
AR.1            8.7734           +0.0000j            8.7734            0.0000
-----------------------------------------------------------------------------
"""
```

[690]:
```python
model_fit.plot_predict(dynamic=False)
plt.show()
```



[691]:
```python
# Create Training and Test For Arima
train_length = 243

train_sales = clock2.Clock_Sales[0:train_length]
test_sales = clock2.Clock_Sales[train_length:]
```

[692]:
```
#  Build Model
model = ARIMA(train_sales, order=(1, 0, 0))
fitted = model.fit(disp=0)

# Forecast using 95% confidence interval
fc, se, conf = fitted.forecast(60, alpha=0.05)

# Make as pandas series
fc_series = pd.Series(fc, index=test_sales.index)
lower_series = pd.Series(conf[:, 0], index=test_sales.index)
upper_series = pd.Series(conf[:, 1], index=test_sales.index)

# Plot
plt.figure(figsize=(12,5), dpi=100)
plt.plot(train_sales, label='training')
plt.plot(test_sales, label='actual')
plt.plot(fc_series, label='forecast')
plt.fill_between(lower_series.index, lower_series, upper_series, color='k',␣
 ↪alpha=.15)
plt.title('Forecast vs Actuals')
plt.legend(loc='upper left', fontsize=8)
plt.show()
```



[693]:
```
# Calculating RMSE and MAPE

arima_rmse = np.sqrt(mean_squared_error(test_sales, fc_series)).round(2)
arima_mape = np.round(np.mean(np.abs(test_sales - fc_series)/test_sales)*100,2)

results = pd.DataFrame({'Method':['ARIMA method'], 'MAPE': [mape], 'RMSE':␣
 ↪[rmse]})
```

```
results = results[['Method', 'RMSE', 'MAPE']]
results
```

[693]:

	Method	RMSE	MAPE
0	ARIMA method	868.59	58.44

[695]:

```
# Table Results

Table = PrettyTable(["Model","MAPE", "RMSE"])
Table.add_row(["Naive", n_mape, n_rmse])
Table.add_row(["Simple Average", sa_mape, sa_rmse])
Table.add_row(["Moving Average", ma_mape, ma_rmse])
Table.add_row(["Simple Exponential", se_mape, se_rmse])
Table.add_row(["Holt Linear", hl_mape, hl_rmse])
Table.add_row(["Holt Winter", hw_mape, hw_rmse])
Table.add_row(["ARIMA", arima_mape, arima_rmse])
print("Time Series Model Performance Sorted by MAPE")
Table.sortby = "MAPE"
print(Table)
```

```
Time Series Model Performance Sorted by MAPE
+--------------------+-------+--------+
|       Model        |  MAPE |  RMSE  |
+--------------------+-------+--------+
|   Simple Average   | 58.35 | 868.61 |
|       ARIMA        | 58.44 | 868.59 |
| Simple Exponential | 63.33 | 819.63 |
|       Naive        | 65.53 | 962.45 |
|   Moving Average   | 74.06 | 778.11 |
|    Holt Linear     | 74.44 | 776.13 |
|    Holt Winter     | 83.05 | 729.28 |
+--------------------+-------+--------+
```